

WebbotLib  
09 October 2010

## Contents

<b>Introduction</b>	<b>5</b>
<b>Installing the library</b>	<b>8</b>
<b>Getting started with AVRStudio</b>	<b>9</b>
<b>My first program</b>	<b>12</b>
<b>Photovore</b>	<b>14</b>
<b>Header Files</b>	
a2d.h	18
actuators.h	23
buffer.h	25
core.h	28
color.h	31
device.h	33
eeprom	34
errors.h	40
i2cBus.h	42
iopin.h	53
led.h	60
segled.h	63
libdefs.h	71
motorPWM.h	76
pid.h	79
pinChange.h	82
pwm.h	85
rprintf.h	88
scheduler.h	96
servos.h	98
spi.h	105
spisw.h	114
spiUart.h	115
switch.h	116
timer.h	117
tone.h	133
uart.h	137
uartsw.h	147
sys	149
DC Motors	171
Stepper	189
Sensors	203
Displays	275
Cameras	286

Storage	300
Servos	322
Controller	332
Gait	340
Text2Speech/Text2Speech.h	348
Maths	352
Audio/SOMO14D.h	368
<b>Frequently Asked Questions</b>	<b>374</b>

## Overview

WebbotLib is a 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.



## Introduction

*"Another library? It's gotta add something special or else I won't be using it - I only just got familiar with the last one."* Quite Right! Neither would I.

*"Man - look at the size of this manual! It's big and big = complicated"* I would rather say that the manual is 'comprehensive'. It covers enough detail for those who are adding new functionality to it who need to understand the detail - but, for the novice, we also provide some much easier to use functions. So don't think that you need to understand all of this manual. For example: the timer.h module contains a lot of low level stuff which is probably only of interest to people writing new motor drivers for the library. But if all you want to do is use an existing motor driver then you don't need to understand timer.h at all. Consequently: for each header file I have attempted to categorise each function as to whether it's an advanced function or not. Newbies can skip the 'advanced functions'. Of course my decision as to what is advanced or not is somewhat arbitrary. I have also made the decision to release ALL detail to ALL people - newbie or not. The other alternative would be to have a separate small manual for newbies and a larger one for more advanced folk - but where do you draw the line?

Why have I invested an enormous amount of my own time to come up with a new library? Well it's really down to my own frustrations with AVRlib (the competition?) as well as my experience with the kind of programming questions posted to the Society of Robots (SoR) forum.

AVRlib tries to be 'all things to all AVRs' and I am sure that this is true but I am not going to put it down. But the result is a library that is still very much tied into the hardware meaning that you (the programmer) need to have some understanding of each chip or board that uses it. This is fine for 'experienced' developers who can invest the time in understanding datasheets of both the processor (which are 100s of pages long) and of the board they are using (ie what pins from the processor can actually be used). But for newbies it is rather unhelpful.

### **A compromise**

In order to try and be more removed from the underlying hardware this library concentrates on the following AVRs:



### Chip

ATMega1280

ATMega128

ATMega168

ATMega2560

ATMega2561

ATMega328P

ATMega32

ATMega640

ATMega644

ATMega8

This covers most set ups for the Society of Robots. If you want an extra chip added then let me know - and I will try to add support. Note that 8k chips like the ATMega8 have limited program space and so can only cope with small programs and may be dropped in the near future.

## Available Ports

Here's a little example. Let's assume you are creating a program for the Axon which uses the ATMega640 processor. The processor has 8 i/o pins on Port L and so AVRlib will let you create a program that uses them. Only problem is that the Axon board doesn't actually have any header pins for Port L but you don't know that since your program will compile ok. With **this** library the program will not compile. Saves you time and hassle.

## I/O Pins

Other libraries give you a myriad of different ways just to set an output pin high (see iopin.h for more info). This may mean that it caters for your own particular coding preference but it creates code that becomes unreadable - especially if you have more than one developer each having their own preferred way of doing things. So it's flexible but chaotic.



The heart of all libraries (AVRlib and this one) is the WinAVR library. WinAVR is very low level and should be thought of like a device driver. Programmers should be shielded from it as far as possible otherwise they can do strange things. WinAVR defines lots of things like ports, registers and stuff and various bit definitions. But the two aren't linked together in any way. So using most libraries I can write code that is valid but complete nonsense e.g. `sbi(PORTD,MUX1);` where both parameters are valid but in combination they are not - the compiler doesn't know this and you can only debug the program by checking the datasheets. My library tries to rectify this (e.g. the above example will not compile).

My library goes further by disallowing all of the other ways of setting I/O pins. It may sound restrictive but the reason is to get all I/O to go through a central place. This would mean that it would be easy to log all I/O pin changes: either to a PC as AVR scenario files or to the SoR'scope. Dead easy to just turn it on for testing and off for production - no coding needed!

### **Standalone - no more copying files into your own project**

WinAVR is a nice library - it just defines stuff in 'h' files and you can include as many of them as you like but it won't generate any code unless you reference something it has defined. Also: because it is just a bunch of 'h' files then you can point your makefile, or AVRStudio, to the base folder of the library and that's it. AVRlib is a bit more temperamental because it has some C files as well as the H files. Most newbies just end up copying the C files into their own project just to get it to compile. If you now download a new version of AVRlib then you have real problems. The compiler is probably picking up the 'h' files from the new AVRlib but is using the old 'c' files that you copied into the project. So now you have to recopy the new C files into every project.

I have tried to solve this by providing pre-compiled libraries for each of the covered devices. Each library will work irrespective of the processor speed. e.g. there is only one library for ATmega168 and this will work for 1MHz, 8Mhz and 20Mhz clock speeds. Examples are given later as to how you instruct the compiler to use the correct library.

### **Portability across outputs**

If you are still unconvinced then here is another reason for this library. Let's say you've made your robot and written the code. Now you want to change some of the servos for DC motors controlled via a UART on a motor controller board such as the Sabretooth. Big change? For some libraries then 'Yes' but not with my library.



## Installing the library

The library is supplied as a zipped file and you will need to use a program such as WinZip to extract the individual files. Note that the file contains a sub-directory structure and so you must make sure that this structure is maintained when the files are extracted. So if your extractor has an option such as 'Use Folder Names?' then make sure it is ticked.

You can extract the files to any folder you want. But for the sake of this guide I will assume that you have extracted to a folder called C:\WebbotLib. If you choose another location then replace C:\WebbotLib in these instructions with your own folder.

Having extracted the files you will see that there are several files of the form **libWebbot-\*.a**. These are the compiled libraries and there will be one for each of the devices supported. So for example: libWebbot-ATmega640.a is the library for the ATmega640.

The remaining files are all the header files (they end in .H) for inclusion in your own project. The only exception is 'version.h' which contains no code but just shows the version of the library that you have got installed.

Thats it - all installed.

If you want to make life easier for yourself then I suggest you download Project Designer from my website <http://webbot.org.uk>

This application makes it much easier to write your own programs because it generates all of the setup and initialisation code for the devices you use as well as showing you what wires you need to connect to each device.

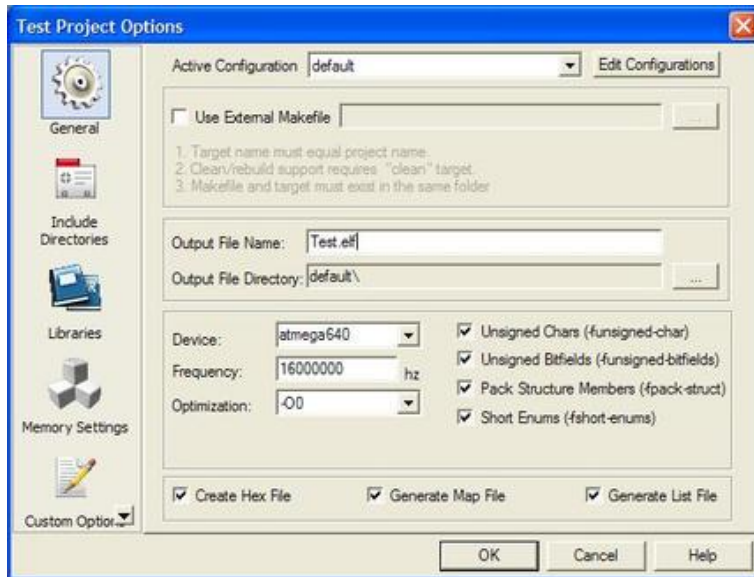




## Getting started with AVRStudio

Create your new project with AVR Studio as normal. If you don't know how to do that then read the AVR Studio manual.

In order to use the library you need to tell AVR Studio where the library has been installed. From the 'Project' menu option choose the 'Configuration Options'. This will open up a new window.



You need to make sure that the 'Device' and 'Frequency' fields contain the correct values for your board. So for example: for an Axon the device should be 'atmega640' and the frequency should be '16000000'. If you forget to set these to the correct values then you will get an error when trying to compile the program.

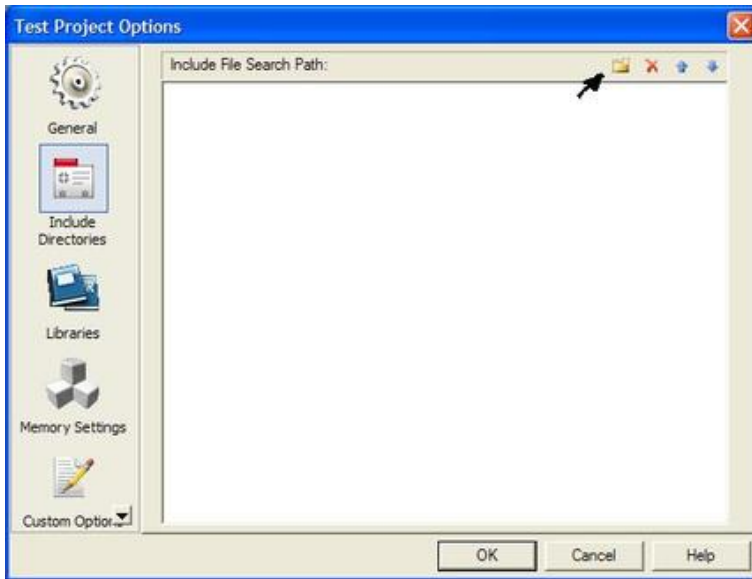
The next steps may be new to you so **please read carefully.**

When using libraries many people copy the files that they want out of the library directory and paste them into their own project directory. This is very bad practise

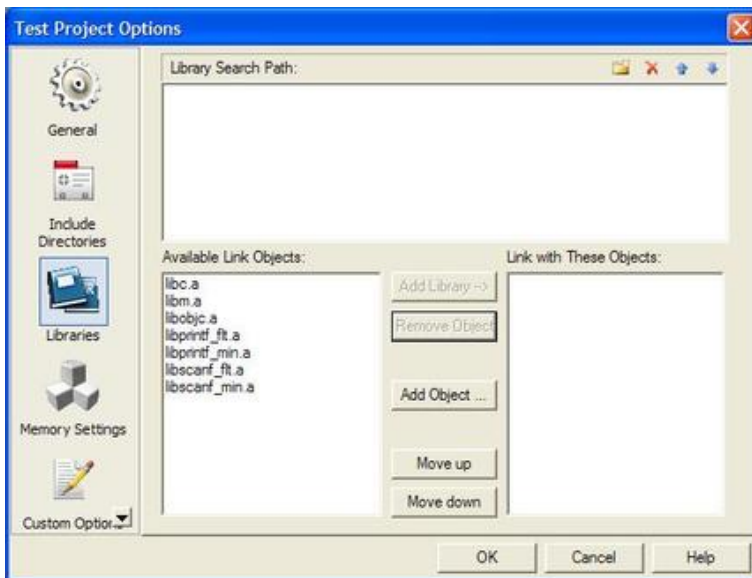
and you should avoid doing it. Otherwise: if you download a new version of the library then you need to copy the files to all of your projects. As well as taking up unnecessary disk space it quickly becomes unmanageable. So here is how to avoid it.

Click on the 'Include Directories' button on the left hand side of the window.





You will now see something like this ie the list of included directories is blank. So now we tell AVR Studio how to find the header files for my library by adding the installation directory to the list. So click on the 'new' button at the top right of the window. This will allow you to type in a directory name such as 'C:\WebbotLib'. Alternatively: you can click the '...' button at the right hand side of the line so that you can browse your computer to locate the correct directory. Having done that we move on to the next step by clicking the 'Libraries' button on the left hand side of the window.

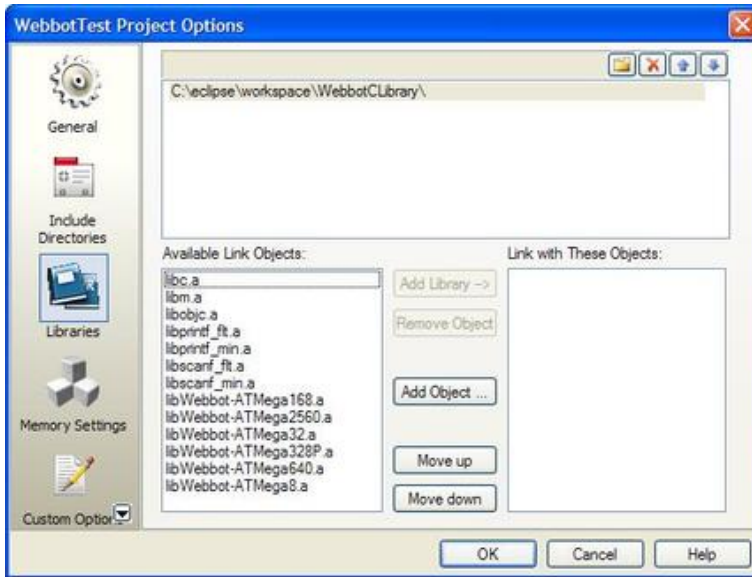


This window allows you to specify the precompiled libraries you want to add to your project. The main windows has three lists:-

1. The top list allows you to specify the directories where your precompiled libraries are stored.
2. The bottom left window shows all of the precompiled libraries that AVR Studio knows about and they all start with 'lib' and end in '.a'.
3. The bottom right window shows the list of libraries that you want to include and will be blank by default.

In the top window: click on the new button and insert the installation directory (just as you did earlier). Assuming that you have selected the correct directory then the bottom left window should now also show the WebbotLib libraries for each processor as follows -





Select the appropriate library for your processor. The one that you use should be the same one as the 'Device' you specified in the very first window. So for the Axon you should use 'libWebbot-ATMega640.a'. Copy it to the bottom right window by clicking the 'Add Library->' button.

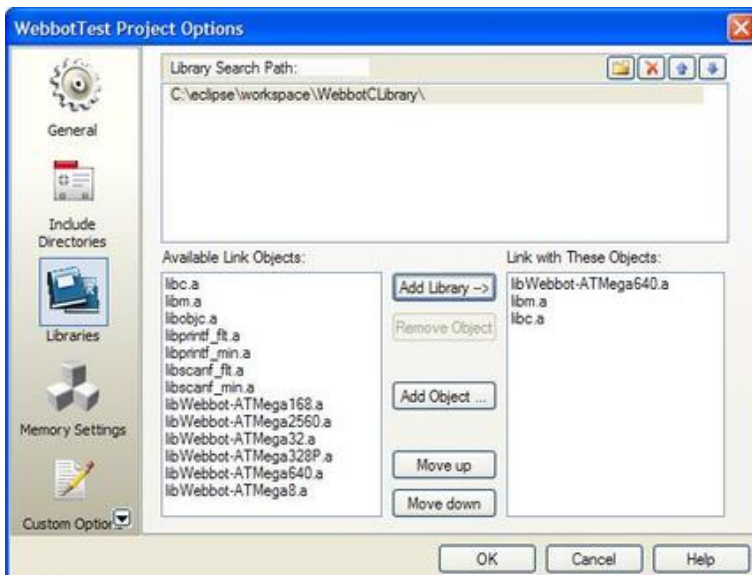
Next you should do the same thing to copy 'libm.a' and then 'libc.a' but do not copy any of the other libraries (unless you are an advanced user and you know that they are needed).

The bottom right window should list the libraries in the following order:-

1. Your chosen libWebbot-ATMegaxxx.a
2. libm.a
3. libc.a

NB If they are shown in a different order then you can use the 'Move Up' and 'Move Down' buttons to change the order. **FAILURE TO PUT THEM IN THIS ORDER WILL CAUSE COMPILE ERRORS.**

Your window should now look like this:-



Ok we are all done so you can click the OK button.



## My first program

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

Every program you write that uses my library will be of the same format:-

```
// Place any #define statements here before you include ANY other files

// You must ALWAYS specify the board you are using
// These are all in the 'sys' folder e.g.
#include "sys/axon.h" // I am using an Axon

// Now include any other files that are needed here
#include "uart.h"
#include "rprintf.h"

// Now create any global variables such as motors, servos, sensors etc

// This routine is called once only and allows you to do set up the hardware
// Dont use any 'clock' functions here - use 'delay' functions instead
void appInitHardware(void){
    // Set UART0 to 19200 baud
    uartInit(UART0, 19200);
    // Tell rprintf to output to UART0
    rprintfInit(&uart0SendByte);
}
// This routine is called once to allow you to set up any other variables in your
program
// You can use 'clock' function here.
// The loopStart parameter has the current clock value in µS
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0; // dont pause after
}

// This routine is called repeatedly - its your main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    rprintf("Hello world\n");
    return 1000000; // wait for 1 second before calling me again. 1000000us = 1
second
}
```

Now obviously that program doesn't actually do anything except send 'Hello World' every 1 second over UART0 but it should compile. If not then you have probably failed to set up your environment correctly.

The first change you will notice is that your program doesn't have a 'main' function. This function actually exists in the library and is responsible for calling 'appInitHardware', 'appInitSoftware' and 'appControl'.



You will also notice that 'appControl', unlike the usual 'main', actually exits - ie it is not surrounded by the traditional:-

```
while(1){  
... Keep doing stuff for ever ...  
}
```

The reason being that the routine is passed in several useful variables:-

1. 'loopCount' is an incrementing number which will eventually wrap around back to zero.
2. 'loopStart' is the current clock in  $\mu$ S.

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 20000 will mean that your appControl will be called every 20ms. Obviously if your code takes longer than that to run, say 25ms, then it will be called again almost immediately.

So WebbotLib spends some of its time running your program 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.



## Photovore

This is an implementation of the Society of Robots 'Photovore' program but using this library instead.

We will use PWM to control the modified servos and so, for the Axon, we will use ports E3 and E4 that are connected to the PWM outputs of Timer 3. Just make sure that your servos can cope with the voltage on those pins.

```
#include "sys/axon.h"
#include "servos.h"
#include "a2d.h"

// Define two light sensors connected to ADC channels 0 and 1
#define sensorLeft ADC_NUMBER_TO_CHANNEL(0)
#define sensorRight ADC_NUMBER_TO_CHANNEL(1)

// Define two servos
SERVO left = MAKE_SERVO(FALSE, E3,1500, 500);
SERVO right = MAKE_SERVO(TRUE , E4,1500, 500);

// Create the list - remember to place an & at the
// start of each servo name
SERVO_LIST servos[] = {&left,&right};

// Create a driver for the list of servos
SERVO_DRIVER bank1 = MAKE_SERVO_DRIVER(servos);

//the larger this number, the more likely your robot will drive straight
#define threshold 8
```

```
// In my initialising code - pass the list of servos to control
void appInitHardware(void){
    // Initialise the servo controller
    servoPWMInit(&bank1);
    // Give each servo a start value of 'stop'
    act_setSpeed(&left, 0);
    act_setSpeed(&right,0);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
```



```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    uint8_t lightLeft = a2dConvert8bit(sensorLeft);
    uint8_t lightRight = a2dConvert8bit(sensorRight);
    if(lightLeft > lightRight && (lightLeft - lightRight) > threshold){
        // go left
        act_setSpeed(&left,DRIVE_SPEED_MIN);
        act_setSpeed(&right,DRIVE_SPEED_MAX);
    }else if(lightRight > lightLeft && (lightRight - lightLeft) > threshold){
        // go right
        act_setSpeed(&left,DRIVE_SPEED_MAX);
        act_setSpeed(&right,DRIVE_SPEED_MIN);
    }else{
        // Go forwards
        act_setSpeed(&left,DRIVE_SPEED_MAX);
        act_setSpeed(&right,DRIVE_SPEED_MAX);
    }
    return 20000; // wait for 20ms to stop crazy oscillations
}
```

Servos all re-act somewhat differently, compounded with how accurately you have centred them, so I suggest you read the comments in *"servos.h"* (see page 98) for *MAKE\_SERVO* about changing the parameters to the *MAKE\_SERVO* calls above to adjust its behaviour to match your servos.

Now let's make a big change. Instead of servos we decide to use DC motors attached to a Sabertooth motor controller from Dimension Engineering which is accessed via a UART using 'simplified serial mode 3' mode at 19200 baud; and the board is set up as address 128 - although this is not used in this mode.



```
#include "sys/axon.h"
#include "Motors/DimensionEngineering/Sabertooth.h"
#include "a2d.h"

// Define two light sensors connected to ADC channels 0 and 1
#define sensorLeft ADC_NUMBER_TO_CHANNEL(0)
#define sensorRight ADC_NUMBER_TO_CHANNEL(1)

// Define two motors
SABERTOOTH_MOTOR left = MAKE_SABERTOOTH_MOTOR(FALSE, 128,1);
SABERTOOTH_MOTOR right = MAKE_SABERTOOTH_MOTOR(TRUE , 128,2);

// Create the list - remember to place an & at the
// start of each servo name
SABERTOOTH_MOTOR_LIST motors[] = {&left,&right};

// Create a driver for the list of motors
SABERTOOTH_DRIVER driver = MAKE_SABERTOOTH_DRIVER(motors, UART0, 19200, SIMPLE);

//the larger this number, the more likely your robot will drive straight
#define threshold 8
```





```
// In my initialising code - pass the list of motors to control
void appInitHardware(void){
    // Initialise the motor controller
    sabertoothInit(&driver);

    // Give each servo a start value of 'stop'
    act_setSpeed(&left, 0);
    act_setSpeed(&right,0);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    uint8_t lightLeft = a2dConvert8bit(sensorLeft);
    uint8_t lightRight = a2dConvert8bit(sensorRight);
    if(lightLeft > lightRight && (lightLeft - lightRight) > threshold){
        // go left
        act_setSpeed(&left,DRIVE_SPEED_MIN);
        act_setSpeed(&right,DRIVE_SPEED_MAX);
    }else if(lightRight > lightLeft && (lightRight - lightLeft) > threshold){
        // go right
        act_setSpeed(&left,DRIVE_SPEED_MAX);
        act_setSpeed(&right,DRIVE_SPEED_MIN);
    }else{
        // Go forwards
        act_setSpeed(&left,DRIVE_SPEED_MAX);
        act_setSpeed(&right,DRIVE_SPEED_MAX);
    }
    return 20000; // wait for 20ms to stop crazy oscillations
}
```

You will see that my 'appControl' function has not changed at all!



## a2d.h

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.

In order to make your code more portable then you should reference each channel by its 'name' as defined in this file. e.g.:-

```
ADC_CH_ADC0
ADC_CH_ADC1
ADC_CH_ADC2
ADC_CH_ADC3
ADC_CH_ADC4
ADC_CH_ADC5
ADC_CH_ADC6
ADC_CH_ADC7
etc
```

This list also contains any other hardware supported options such as the difference between different channels. However: these options are not often used and so I have also created shortcut names for the individual channels called: ADC0, ADC1, ADC2 etc.

By using these names (rather than just numbers like 0, 1, 2 etc) then the compiler will generate an error if the target processor does not support that channel.

Note that the channel numbers are not sequential numbers so you cannot go through the channels with a 'for' loop. Channels 0 to 7 are one sequence of numbers but channels 8 to 15 are another sequence.

This code will NOT work as you might expect:-

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

You must use the ADC\_NUMBER\_TO\_CHANNEL macro to convert a 'number' into a valid 'channel'. So the code should be written as:-

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

Whenever you read a channel the library will automatically initialise the ADC system so you don't need to call 'a2dInit' yourself. It is only there so that code converted from AVRlib doesn't generate any errors.



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. If you want you can change this by using #define commands before you include this file. For example, to use a prescaler of 8 you would write:

```
#define ADC_DEFAULT_PRESCALE ADC_PRESCALE_DIV8
#include "a2d.h"
```

Alternatively you can change the values at runtime with the a2dSetPrescaler and a2dSetReference commands.

## Standard Function Summary

ADC_CHANNEL	<a href="#">ADC NUMBER TO CHANNEL</a> This macro can be used to convert a number into a channel.
uint8_t	<a href="#">a2dConvert8bit(ADC_CHANNEL channel)</a> Read the value from an ADC channel and return it as a single byte in the range 0-255NB The parameter should either be one of the constant values such as ADC_CH_ADC3 or you can convert a number into a channel by using the ADC_NUMBER_TO_CHANNEL macro.
uint16_t	<a href="#">a2dConvert10bit(ADC_CHANNEL channel)</a> Returns the value in the range 0-1023 from an ADC channel.
uint8_t	<a href="#">NUM_ADC_CHANNELS</a> Returns the number of ADC channels on this device.
uint8_t	<a href="#">a2dReadPercent(ADC_CHANNEL channel)</a> Returns the value of the ADC channel as a percentage ie 0 to 100.
uint16_t	<a href="#">a2dReadMv(ADC_CHANNEL channel)</a> Returns the value of an ADC_CHANNEL in milli-volts.

## Advanced Function Summary

	<a href="#">a2dOff()</a> This will turn off the ADC unit which will save power.
	<a href="#">a2dSetPrescaler(uint8_t prescale)</a> Change the prescaler.
	<a href="#">a2dSetReference(uint8_t ref)</a> Set the reference voltage for the ADC unit.
	<a href="#">a2dInit()</a> This will turn on the ADC unit.



---

## Advanced Function Summary

uint16_t
----------

<a href="#">a2dGetAVcc(void)</a>
----------------------------------

Returns the AVcc voltage in milli volts ie 5 volts would be returned at 5000.
---

---

## Standard Function Detail

### ADC\_NUMBER\_TO\_CHANNEL

ADC\_CHANNEL ADC\_NUMBER\_TO\_CHANNEL

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

For example to read all of the channels could be done like this:

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

---

### 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

**NB The parameter should either be one of the constant values such as ADC\_CH\_ADC3 or you can convert a number into a channel by using the ADC\_NUMBER\_TO\_CHANNEL macro.**

So to read channel 3 you could either write:

```
uint8_t value = a2dConvert8bit(ADC_CH_ADC3);
```

or

```
uint8_t value = a2dConvert8bit( ADC_NUMBER_TO_CHANNEL(3) );
```

But you should **never** write:

```
uint8_t value = a2dConvert8bit(3);
```

This will definitely cause problems for channels 8 onwards.

---

### a2dConvert10bit

uint16\_t a2dConvert10bit(ADC\_CHANNEL channel)

Returns the value in the range 0-1023 from an ADC channel.



**NB The parameter should either be one of the constant values such as ADC\_CH\_ADC3 or you can convert a number into a channel by using the ADC\_NUMBER\_TO\_CHANNEL macro.**

So to read channel 3 you could either write:

```
uint16_t value = a2dConvert10bit(ADC_CH_ADC3);
```

or

```
uint16_t value = a2dConvert10bit( ADC_NUMBER_TO_CHANNEL(3));
```

But you should **never** write:

```
uint16_t value = a2dConvert10bit(3);
```

This will definitely cause problems for channels 8 onwards.

---

## NUM\_ADC\_CHANNELS

uint8\_t NUM\_ADC\_CHANNELS

Returns the number of ADC channels on this device.

---

## a2dReadPercent

uint8\_t a2dReadPercent(ADC\_CHANNEL channel)

Returns the value of the ADC channel as a percentage ie 0 to 100.

---

## 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.

---

## Advanced Function Detail

### a2dOff

a2dOff()

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

Before reading any more values you should call a2dInit();

---

### a2dSetPrescaler

a2dSetPrescaler(uint8\_t prescale)

Change the prescaler. Possible values are:-

---



```
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.

---

## a2dSetReference

`a2dSetReference(uint8_t ref)`

Set the reference voltage for the ADC unit. The parameter should be one of the following:

```
ADC_REFERENCE_AREF
ADC_REFERENCE_AVCC
ADC_REFERENCE_INTERNAL
```

**NB - This function has been deprecated and AVCC is always used.**

---

## a2dInit

`a2dInit()`

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()`.

---

## a2dGetAVcc

`uint16_t a2dGetAVcc(void)`

Returns the AVcc voltage in milli volts ie 5 volts would be returned at 5000.

---



## actuators.h

Contains constants, datatypes, and common commands for actuators eg: servos and motors.

This library tries to standardise on how we deal with these devices.

For example we introduce the concept of 'drive speed' so that we can use the same code to drive motors and modified servos. Theoretically this means we can swap between any actuator without changing the code. A drive speed of 0 means 'stop' regardless of whether you are using a motor or a modified servo. DRIVE\_SPEED\_MIN means go full speed in reverse and DRIVE\_SPEED\_MAX means go full speed forwards. Obviously the actual speed the robot moves at will depend on the abilities of the actual motor/servo as well as the wheel diameter.

For unmodified servos then DRIVE\_SPEED\_MIN and DRIVE\_SPEED\_MAX are the maximum movements of the servo and DRIVE\_SPEED\_CENTER is the center position.

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.

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.

### Standard Function Summary

	<a href="#"><u>act_setSpeed( ACTUATOR_COMMON* act, DRIVE_SPEED speed)</u></a> Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX.
DRIVE_SPEED	<a href="#"><u>act_getSpeed(const ACTUATOR_COMMON* act)</u></a> Returns the last value from 'act_setSpeed'.
	<a href="#"><u>act_setConnected( ACTUATOR_COMMON* act,boolean connected)</u></a> Connect or disconnect the actuator from the drive system.



---

## Standard Function Summary

boolean
---------

<a href="#"><code>act isConnected(const ACTUATOR COMMON* act)</code></a>
--

Test if an actuator is connected.
-----------------------------------

boolean
---------

<a href="#"><code>act isInverted(const ACTUATOR COMMON* act)</code></a>
---

Test if an actuator is inverted.
----------------------------------

---

## Standard Function Detail

### **act\_setSpeed**

`act_setSpeed(__ACTUATOR_COMMON* act, DRIVE_SPEED speed)`

Set the required speed to a value between DRIVE\_SPEED\_MIN and DRIVE\_SPEED\_MAX.

If the actuator has been disconnected then nothing will happen until it is reconnected.

---

### **act\_getSpeed**

`DRIVE_SPEED act_getSpeed(const __ACTUATOR_COMMON* act)`

Returns the last value from 'act\_setSpeed'.

NB This is not the actual speed that the actuator is moving at - we have no way of knowing that without an encoder!

---

### **act\_setConnected**

`act_setConnected(__ACTUATOR_COMMON* act, boolean connected)`

Connect or disconnect the actuator from the drive system. The second parameter should be TRUE to connect, or FALSE to disconnect.

When an actuator is disconnected it stops receiving commands.

---

### **act\_isConnected**

`boolean act_isConnected(const __ACTUATOR_COMMON* act)`

Test if an actuator is connected. Returns TRUE if it is, and FALSE if it is not.

---

### **act\_isInverted**

`boolean act_isInverted(const __ACTUATOR_COMMON* act)`

Test if an actuator is inverted. Returns TRUE if it is, FALSE if it is not.

You cannot change the inverted state except when creating the actuator.





## 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

	<a href="#"><u>bufferInit(cBuffer* buffer, uint8_t *start, size_t size)</u></a> Before the buffer can be used it must be initialised by this function.
boolean	<a href="#"><u>bufferGet(cBuffer* buffer, uint8_t *rtn)</u></a> Gets the next byte from the buffer.
boolean	<a href="#"><u>bufferPut(cBuffer* buffer, uint8_t data)</u></a> Writes the byte into the buffer.
size_t	<a href="#"><u>bufferFreeSpace(const cBuffer* buffer)</u></a> Returns the number of bytes available in the buffer.
size_t	<a href="#"><u>bufferBytesUsed(const cBuffer* buffer)</u></a> Return the number of bytes currently stored in the buffer.

### Advanced Function Summary

	<a href="#"><u>bufferDump(cBuffer* buffer, size_t numbytes)</u></a> Deletes numbytes from the front of the buffer.
uint8_t	<a href="#"><u>bufferGetAtIndex(const cBuffer* buffer, size_t index)</u></a> 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.
	<a href="#"><u>bufferFlush(cBuffer* buffer)</u></a> Removes all data from the buffer
boolean	<a href="#"><u>bufferIsFull(const cBuffer* buffer)</u></a> Returns TRUE if the buffer is full or FALSE if not.

### Standard Function Detail

#### bufferInit

```
bufferInit(cBuffer* buffer, uint8_t *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.



## 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.



## core.h

This file is always included by the system file (SYS/\*.h) that you use.

It defines some standard macros and datatypes and also provides the main entry point for your program.

### Standard Function Summary

Never returns

#### [main\(void\)](#)

This is the main entry point of your program and it will help to set up your program ready for use.

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 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\(int16\\_t value, int16\\_t minVal, int16\\_t maxVal, uint16\\_t minRtn, uint16\\_t maxRtn\)](#)

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.

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.

### Standard Function Detail

#### main

Never returns `main(void)`

This is the main entry point of your program and it will help to set up your program ready for use.

- It will configure your machine by calling 'configure\_ports' in the system file you have included. This will set up the I/O pins into a known state ready for use.



- Next it will call the 'void applnitHardware(void)' function in your program. This is your opportunity to do some of that 'one time set up' of the hardware devices. If you don't need to set anything up then the function must still be there but can have no lines of code.
  - It will initialise all of the available timers to known states and try to select an unused timer for the system clock.
  - Then it will call the 'applnitSoftware' function in your program. This allows you to initialise any variables in your program.
  - Finally it will go into a loop that keeps calling the appControl function in your application with a few parameters. The loop count is an incremental number (ie it increments each time) and the loop start is the number of microseconds since the board was powered on. NB both of these values will eventually wrap around back to 0.
- 

## 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 = interpolate(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 = interpolate(theValue, 0, 2048, 300, 800);
newValue = CLAMP(newValue,300,800); // limit the answer to be in the range
300 to 800
```

---

## 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.



## 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 'colorSetRGB', 'colorSetYUV' commands. Or you may assign it a value from another colour value using the 'color2rgb', 'color2yuv' commands.

### Standard Function Summary

COLOR_RGB*	<a href="#">color2rgb(const COLOR * src, COLOR* dest)</a> Copies a colour and converts it to RGB if required.
COLOR_YUV*	<a href="#">color2yuv(const COLOR * src, COLOR* dest)</a> Copies a colour and converts it to YUV if required.
	<a href="#">colorSetRGB(COLOR* color, uint8 t r, uint8 t g, uint8 t b)</a> Assign an RGB value to colour.
	<a href="#">colorSetYUV(COLOR* color, uint8 t y, uint8 t u, uint8 t v)</a> Assign a YUV value to a colour.
	<a href="#">colorDump(const COLOR* color)</a> Dump a colour value to the current rprintf destination.
boolean	<a href="#">colorEquals(const COLOR* c1, const COLOR* c2)</a> Test if two colours are the same.

### Standard Function Detail

#### color2rgb

```
COLOR_RGB* color2rgb(const COLOR * src, COLOR* dest)
```

Copies a colour and converts it to RGB if required.

The first parameter points to an existing colour which may be in any colour space. The second parameter points to a colour variable which will be assigned the RGB value.



## color2yuv

`COLOR_YUV* color2yuv(const COLOR * src, COLOR* dest)`

Copies a colour and converts it to YUV if required.

The first parameter points to an existing colour which may be in any colour space. The second parameter points to a colour variable which will be assigned the YUV value.

---

## colorSetRGB

`colorSetRGB(COLOR* color, uint8_t r, uint8_t g, uint8_t b )`

Assign an RGB value to colour.

The first parameter is the address of the colour variable and the remaining parameters specify the RGB values in the range 0 to 255.

So to set a colour variable to 'red' then use:

```
COLOR red;
colorSetRGB(&red, 255, 0, 0);
```

---

## colorSetYUV

`colorSetYUV(COLOR* color, uint8_t y, uint8_t u, uint8_t v )`

Assign a YUV value to a colour.

The first parameter is the address of the colour variable and the remaining parameters specify the YUV values in the range 0 to 255.

Example:

```
COLOR c;
colorSetYUV(&c, 0, 127, 127); // Assign it yuv = 0,127,127
```

---

## colorDump

`colorDump(const COLOR* color)`

Dump a colour value to the current rprintf destination.

---

## colorEquals

`boolean colorEquals(const COLOR* c1, const COLOR* c2)`

Test if two colours are the same.

The two parameters are the addresses of colours. The function will return TRUE if, and only if, the two colours are of the same colour space (ie both RGB or both YUV) and all of the colour band values are the same.

---





## device.h

Defines items such as timers, uarts, IO pins, A2D channels etc that are available on this device. It is included automatically once you select a system and so should **never** be included directly within your own code.

This information is used by the rest of the library and most of the information is made available via more friendly functions.



## eeeprom

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 300)

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!

## Standard Function Summary

<code>uint8_t</code>	<a href="#"><code>eeeprom_read_byte (const uint8_t * p)</code></a> Read a byte from EEPROM.
	<a href="#"><code>eeeprom_write_byte (uint8_t * p, uint8_t value)</code></a> Writes a byte to EEPROM.
<code>uint16_t</code>	<a href="#"><code>eeeprom_read_word (const uint16_t * p)</code></a> Read a word (ie a 16 bit number) from EEPROM.
	<a href="#"><code>eeeprom_write_word (uint16_t * p, uint16_t value)</code></a> Writes a word (ie a 16 bit number) to EEPROM.
<code>uint32_t</code>	<a href="#"><code>eeeprom_read_dword (const uint32_t * p)</code></a> Read a double word (ie a 32 bit number) from EEPROM.
	<a href="#"><code>eeeprom_write_dword (uint32_t * p, uint32_t value)</code></a> Writes a double word (ie a 32 bit number) to EEPROM.
	<a href="#"><code>eeeprom_read_block (void * dst, const void * src, size_t n)</code></a> Read a block of data from EEPROM into standard memory.
	<a href="#"><code>eeeprom_write_block (const void * src, void * dst, size_t n)</code></a> Writes a block of data to EEPROM from standard memory.

## Standard Function Detail

### **eeeprom\_read\_byte**

```
uint8_t eeeprom_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 = eeeprom_read_byte(&myVar);
```

### **eeeprom\_write\_byte**

```
eeeprom_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.

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.

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.

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

	<a href="#">setError(ERROR_CODE err)</a> Indicate that an error has happened.
ERROR_CODE	<a href="#">getError()</a> Returns the current error code or 0 if there is no error.
	<a href="#">setErrorLog(Writer log)</a> 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);
```



## i2cBus.h

Provides support for communicating with other I2C slave devices/sensors.

NB if you are using a sensor, or other device, that is supported directly by this library then you don't need to understand any of this. This library 'just does it' for you. So this module is only really of interest to people who are trying to use low level I2C functions. If your device is not listed then you can create a generic slave device using `MAKE_GENERIC_I2C_DEVICE` which just requires the unique number of the device.

Here is a somewhat simplistic view of I2C to save you reading the data sheet.

The 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!).

An I2C conversation, just like the telephone, requires you to call a 'number', say something, listen to the answer, and then hang up. A more complex conversation means doing this several times. Obviously each 'number' is unique. How do I know the 'number/address'? Well: check the data sheet for the device. Some devices only have one fixed address (meaning you can only connect one of them at most) whereas others allow you to change the address either via software or via switches.

The person who is making the call is called the 'master' and the person receiving the call is a 'slave'. The current implementation only supports one master on the same bus and you can only use a given bus either as a master or as a slave but not both.

### I2C Master

To indicate that you are the one and only master on the bus. You will need to do the following:

1. Create a list of the slave devices that are connected to the bus
2. Define the bus using `MAKE_I2C_HARDWARE_BUS` or `MAKE_I2C_SOFTWARE_BUS` providing the list (ie the 'telephone directory' of all the slave devices that are connected to it.
3. Call `i2cBusInit` to initialise the bus ready for use

At this point you may get an error if the devices don't have unique addresses.

Note that you can either use the built-in hardware I2C bus or you can defined as many software I2C busses as you like. You might want to do this, say, because not all of the slave devices have a unique number. So you can split the devices across several busses to make sure that each bus has a unique set of numbers.



The I2C sub-system is set up to use 100kHz as the communication speed. If you wish to change this then you can call `i2cBusSetBitRate` in `aplnitSoftware` - ie after you have initialised the bus.

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).

Warning: if your code talks to an I2C device but you have not connected the device then there is a possibility that the library can hang! So if you don't have the device connected then remove any communications with the device.

Once the bus has been initialised you can start talking to the devices. Most slave devices have a list of 'registers' which can be read and/or written to (check the datasheet of the device). In this case you can just use the library functions to read/write the registers.

If your device needs more complex communications then the receive, send and transfer calls may do what you need but, as a last resort, you can use the low-level start, get, put, stop functions to code the communications yourself.

## I2C\_SLAVE

If your processor is an I2C slave (ie it only accepts in-coming calls) then you can only use the I2C hardware bus - there is no software option.

Implementing a slave is quite easy. You just have to call `i2CSlaveInit` to initialise the bus and declare your own address (telephone number). You then register a callback handler which is called when an incoming message is received, and another which is called to send back a response. Quite what these handlers do, and the format of the messages, is up to you !

### Standard Function Summary

	<a href="#"><u>MAKE I2C HARDWARE BUS(I2C_DEVICE_LIST* devices)</u></a> Create a hardware I2C bus and specify the list of devices that are attached.
	<a href="#"><u>MAKE I2C SOFTWARE BUS(I2C_DEVICE_LIST* devices, const IOPin* scl, const IOPin* sda)</u></a> Create a software I2C bus and specify the list of devices that are attached as well as the IO pins to use for the SCL and SDA wires.
	<a href="#"><u>i2cBusInit(I2C_ABSTRACT_BUS* bus)</u></a> Initialise a hardware or software I2C bus to make it ready for use.



## Standard Function Summary

boolean	<a href="#">i2cMasterReadRegisters(const I2C_DEVICE* device, uint8_t startReg, size_t numBytes, uint8_t* response)</a> A complete communication session to read a number of sequential registers from the device.
boolean	<a href="#">i2cMasterWriteRegisters(const I2C_DEVICE* device, uint8_t startReg, size_t numBytes, const uint8_t* data)</a> Performs a complete communication session with a device to write to a series of registers.
boolean	<a href="#">i2cMasterWriteRegister(const I2C_DEVICE* device, uint8_t reg, uint8_t value)</a> A complete communication session to write a single byte value to a single I2C device register.
	<a href="#">i2cSlaveInit(uint8_t deviceAddr, boolean generalCall, cBuffer* rx, cBuffer* tx)</a> Initialise the bus as a slave.
	<a href="#">i2cSlaveSetReceiveHandler(void (*i2cSlaveRx_func)(cBuffer* data))</a> Register a callback routine that will be called once the slave has received a message.
	<a href="#">i2cSlaveSetTransmitHandler(void (*i2cSlaveTx_func)(cBuffer* data))</a> Register a callback routine that will be called to write the response bytes.

## Advanced Function Summary

	<a href="#">MAKE_GENERIC_I2C_DEVICE(uint8_t i2cAddr)</a> Create a generic slave device with the given address.
	<a href="#">i2cBusSetBitRate(I2C_ABSTRACT_BUS* bus, uint16_t bitrateKHz)</a> Set the communication speed of the I2C interface to the value of (the parameter x 100 kHz).
boolean	<a href="#">i2cMasterSend(const I2C_DEVICE* device, size_t length, const uint8_t* data)</a> Performs a complete communication with a slave device to write a number of bytes.
boolean	<a href="#">i2cMasterSendWithPrefix(const I2C_DEVICE* device, size_t prefixLen, const uint8_t* prefix, size_t length, const uint8_t* data)</a> Performs a complete communication with a slave device to



## Advanced Function Summary

	write a number of bytes.
boolean	<a href="#"><code>i2cMasterReceive(const I2C_DEVICE* device, size_t length, uint8_t *data)</code></a> Performs a complete communication with a slave device to read a number of bytes.
boolean	<a href="#"><code>i2cMasterTransfer(const I2C_DEVICE* device, size_t wlen, const uint8_t *wdata, size_t rlen, uint8_t *rdata)</code></a> Performs a complete communication with a slave device to write a number of bytes and then read a number of bytes.
boolean	<a href="#"><code>i2cStart(const I2C_DEVICE* device, boolean writeMode)</code></a> A low level function to start communicating with a given slave device if you are writing your own I2C communications from scratch.
boolean	<a href="#"><code>i2cPut(const I2C_ABSTRACT_BUS* i2c, uint8_t b)</code></a> This is a low-level function to write a byte across the bus.
uint8_t	<a href="#"><code>i2cGet(const I2C_ABSTRACT_BUS* i2c, boolean isLastByte)</code></a> This is a low-level function to read a byte from the bus.
	<a href="#"><code>i2cStop(const I2C_ABSTRACT_BUS* i2c)</code></a> This is a low-level call to indicate that the communication session has finished (ie 'hang up the phone').

## Standard Function Detail

### MAKE\_I2C\_HARDWARE\_BUS

`MAKE_I2C_HARDWARE_BUS(I2C_DEVICE_LIST* devices)`

Create a hardware I2C bus and specify the list of devices that are attached.

Assuming that you have defined a CMPS03 and one of your own devices at 0xA0 by using:-

```
CMPS03_I2C compass = MAKE_CMPS03_I2C_At(0xC0);
GENERIC_I2C_DEVICE myDevice = MAKE_GENERIC_I2C_DEVICE(0xA0);
```

then the first step is to build a list of the devices that will be on the bus:-

```
static I2C_DEVICE_LIST i2c_list[] = {
    &compass.i2cInfo,
    &myDevice.i2cInfo
};
```

This is just a comma separated list of device entries where each entry has an '&' followed by the name of the device and ending with '.i2cInfo'.



Now that we have a list of devices we can create the hardware bus:

```
I2C_HARDWARE_BUS i2c = MAKE_I2C_HARDWARE_BUS(i2c_list);
```

The only remaining thing to do is to initialise the bus in `aplnitHardware` passing the address of the device list:

```
i2cBusInit(&i2c);
```

All done. You can now start talking to the devices.

---

## MAKE\_I2C\_SOFTWARE\_BUS

`MAKE_I2C_SOFTWARE_BUS(I2C_DEVICE_LIST* devices, const IOPin* scl, const IOPin* sda)`

Create a software I2C bus and specify the list of devices that are attached as well as the IO pins to use for the SCL and SDA wires. NB You MUST add a 4k7 pull up resistor from each pin to the regulated Vcc supply to the processor.

This function is almost identical to the hardware version but just with the addition of specifying the extra IO pins. But, for completeness, here is an example:

Assuming that you have defined a CMPS03 and one of your own devices at 0xA0 by using:-

```
CMPS03_I2C compass = MAKE_CMPS03_I2C_At(0xC0);  
GENERIC_I2C_DEVICE myDevice = MAKE_GENERIC_I2C_DEVICE(0xA0);
```

then the first step is to build a list of the devices that will be on the bus:-

```
static I2C_DEVICE_LIST i2c_list[] = {  
    &compass.i2cInfo,  
    &myDevice.i2cInfo  
};
```

This is just a comma separated list of device entries where each entry has an '&' followed by the name of the device and ending with '.i2cInfo'.

Now that we have a list of devices we can create the software bus using pins D0 for SCL and D1 for SDA :

```
I2C_SOFTWARE_BUS i2c = MAKE_I2C_SOFTWARE_BUS(i2c_list, D0, D1);
```

The only remaining thing to do is to initialise the bus in `aplnitHardware` passing the address of the device list:

```
i2cBusInit(&i2c);
```

All done. You can now start talking to the devices.

---



## i2cBusInit

`i2cBusInit(I2C_ABSTRACT_BUS* bus)`

Initialise a hardware or software I2C bus to make it ready for use.

This should be called from `applnInitHardware`. It may result in an error if the list of devices contains duplicate I2C addresses.

See the examples in "*MAKE\_I2C\_HARDWARE\_BUS(I2C\_DEVICE\_LIST\* devices)*" (see page 45) or "*MAKE\_I2C\_SOFTWARE\_BUS(I2C\_DEVICE\_LIST\* devices, const IOPin\* scl, const IOPin\* sda)*" (see page 46)

---

## i2cMasterReadRegisters

`boolean i2cMasterReadRegisters(const I2C_DEVICE* device, uint8_t startReg, size_t numBytes, uint8_t* response)`

A complete communication session to read a number of sequential registers from the device. Returns TRUE if successful or FALSE if there was an error.

The first parameter specifies the I2C device.

The second parameter specifies the first register to read from.

The third parameter is the number of consecutive registers to read from.

The last parameter specifies the list of data bytes to store the returned values.

For example: given a device at 0x32 and you want to read registers 10 and 11 then you would use:

```
I2C_GENERIC_DEVICE myDevice = MAKE_I2C_GENERIC_DEVICE(0x32);
uint8_t response[2];
i2cMasterReadRegisters(&myDevice.i2cInfo, 10, 2, response);
```

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:-

```
i2cMasterReadRegisters(&myDevice.i2cInfo, 10, sizeof(response), response);
```

---

## i2cMasterWriteRegisters

`boolean i2cMasterWriteRegisters(const I2C_DEVICE* device, uint8_t startReg, size_t numBytes, const uint8_t* data)`

Performs a complete communication session with a device to write to a series of registers.

Returns TRUE if successful or FALSE if there was an error.

The first parameter specifies the device.

---



The second parameter specifies the first register to write to.

The third parameter is the number of consecutive registers to write to.

The last parameter specifies the list of data bytes to be written out.

For example: given a device at 0x32 and you want to set registers 10 and 11 to the values 1 and 2 then you would use:

```
I2C_GENERIC_DEVICE myDevice = MAKE_I2C_GENERIC_DEVICE(0x32);
uint8_t data[] = {1,2};
i2cMasterWriteRegisters(&myDevice.i2cInfo, 10, 2, data);
```

Note how the C operator 'sizeof' can be used to make sure that the number of bytes being written is actually the same as the number of bytes in the 'data' array:-

```
i2cMasterWriteRegisters(&myDevice.i2cInfo, 10, sizeof(data), data);
```

---

## **i2cMasterWriteRegister**

`boolean i2cMasterWriteRegister(const I2C_DEVICE* device, uint8_t reg, uint8_t value)`

A complete communication session to write a single byte value to a single I2C device register. Returns TRUE if succeeded or FALSE if there was an error.

This is a simplified form of the `i2cMasterWriteRegisters(const I2C_DEVICE* device, uint8_t startReg, size_t numBytes, const uint8_t* data)` command if you are only writing to a single (one byte) register.

---

## **i2cSlaveInit**

`i2cSlaveInit(uint8_t deviceAddr, boolean generalCall, cBuffer* rx, cBuffer* tx)`

Initialise the bus as a slave.

This should be called from `aplnitHardware`.

The first parameter is the unique device address which should be an even number between 2 and 254.

The second parameter indicates whether you are interested in responding to general calls which can be broadcast to all the devices on the bus.

The last two parameters specify the data buffers used to store received data and the response data to be returned. These should be initialised to store the longest expected message. See "*buffer.h*" (see page 25)

---





## i2cSlaveSetReceiveHandler

`i2cSlaveSetReceiveHandler(void (*i2cSlaveRx_func)(cBuffer* data))`

Register a callback routine that will be called once the slave has received a message.

For example:

```
void myRxHandler(cBuffer* data){
... process the message in the buffer ...
}
```

You can then register the above routine to be called by writing:

```
i2cSlaveSetReceiveHandler(&myRxHandler);
```

---

## i2cSlaveSetTransmitHandler

`i2cSlaveSetTransmitHandler(void (*i2cSlaveTx_func)(cBuffer* data))`

Register a callback routine that will be called to write the response bytes.

For example:

```
void myTxHandler(cBuffer* data){
... write the response into the data buffer ...
}
```

You can then register the above routine to be called by writing:

```
i2cSlaveSetTransmitHandler(&myTxHandler);
```

---

## Advanced Function Detail

### MAKE\_GENERIC\_I2C\_DEVICE

`MAKE_GENERIC_I2C_DEVICE(uint8_t i2cAddr)`

Create a generic slave device with the given address.

This should be used when the device is not directly supported by this library. The only parameter is the I2C address of the device. For example: if the device has an address of 0xA0 then:

```
GENERIC_I2C_DEVICE myDevice = MAKE_GENERIC_I2C_DEVICE(0xA0);
```

---

## i2cBusSetBitRate

`i2cBusSetBitRate(I2C_ABSTRACT_BUS* bus, uint16_t bitrateKHz)`

Set the communication speed of the I2C interface to the value of (the parameter x 100 kHz).

By default this is set to 100Khz which is equivalent to calling this routine with a parameter of 100. Note that it is ignored by a software bus as it just goes as fast as it can.

---



## i2cMasterSend

```
boolean i2cMasterSend(const I2C_DEVICE* device, size_t length, const
uint8_t *data)
```

Performs a complete communication with a slave device to write a number of bytes.

The return value specifies if it was successful (TRUE) or not (FALSE).

This is the same as the following pseudo code:

```
// Connect in write mode
i2cStart( &myDevice, TRUE);
.. one or more i2cPut to write the data ...
// Hang up
i2cStop( myDevice.bus);
```

---

## i2cMasterSendWithPrefix

```
boolean i2cMasterSendWithPrefix(const I2C_DEVICE* device, size_t
prefixLen, const uint8_t* prefix, size_t length, const uint8_t* data)
```

Performs a complete communication with a slave device to write a number of bytes.

The return value specifies if it was successful (TRUE) or not (FALSE).

This is the same as the following pseudo code:

```
// Connect in write mode
i2cStart( &myDevice, TRUE);
.. one or more i2cPut to write the prefix data ...
.. one or more i2cPut to write the main data ...
// Hang up
i2cStop( myDevice.bus);
```

---

## i2cMasterReceive

```
boolean i2cMasterReceive(const I2C_DEVICE* device, size_t length,
uint8_t *data)
```

Performs a complete communication with a slave device to read a number of bytes.

The return value specifies if it was successful (TRUE) or not (FALSE).

This is the same as the following pseudo code:

```
// Connect in read mode
i2cStart( &myDevice, FALSE);
.. one or more i2cGet to read the data ...
// Hang up
i2cStop( myDevice.bus);
```

---



## i2cMasterTransfer

```
boolean i2cMasterTransfer(const I2C_DEVICE* device, size_t wlen, const
uint8_t *wdata, size_t rlen, uint8_t * rdata)
```

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).

This is the same as the following pseudo code:

```
// Connect in write mode
i2cStart( &myDevice, TRUE);
.. one or more i2cPut to write the data ...
// Connect in read mode
i2cStart( &myDevice, FALSE);
.. one or more i2cGet to read the data ...
// Hang up
i2cStop( myDevice.bus);
```

---

## i2cStart

```
boolean i2cStart(const I2C_DEVICE* device, boolean writeMode)
```

A low level function to start communicating with a given slave device if you are writing your own I2C communications from scratch.

The first parameter specifies the device to contact and the second parameter specifies if you are in write/talk mode (TRUE) or read/listen mode (FALSE).

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 the i2cGet (if you are in read/listen mode) or i2cPut (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 i2cStop.

---

## i2cPut

```
boolean i2cPut(const I2C_ABSTRACT_BUS* i2c, uint8_t b)
```

This is a low-level function to write a byte across the bus.

The first parameter specifies the bus you want to send the data across and the second parameter is the data byte to be sent.



The function returns TRUE if successful or FALSE if not.

Note that this can only be used after a successful call to `i2cStart` has been used to place the bus into write mode.

---

## **i2cGet**

```
uint8_t i2cGet(const I2C_ABSTRACT_BUS* i2c, boolean isLastByte)
```

This is a low-level function to read a byte from the bus.

The first parameter specifies the bus you want to read the data from and the second parameter specifies whether you will be reading more bytes (FALSE) or this is the last one (TRUE).

The function returns the data byte.

Note that this can only be used after a successful call to `i2cStart` has been used to place the bus into read mode.

---

## **i2cStop**

```
i2cStop(const I2C_ABSTRACT_BUS* i2c)
```

This is a low-level call to indicate that the communication session has finished (ie 'hang up the phone').

Once an `i2cStart` has been issued then you must terminate the call with an `i2cStop`.



## iopin.h

*"Oh no! Not another way to use I/O pins"*

The answer is *'Yes - but with good reason'*.

Other libraries don't really provide any help for managing the digital I/O (input/output) pins but leave it down to you to set/clear the correct bit in the correct port. Perhaps using:

```
sbi(PORTD,PD4);  
or  
PORTD |= BV(PD4);  
or  
PORTD |= BV(4);  
or  
PORTD |= 1<<4;  
or  
PORTD |= 1<<PD4;  
or  
PORTD |= 16;
```

They all do exactly the same thing on PORTD Pin4 - and there are lots of extra ways as well. So the mechanism is not very tight - too many choices. Yes it works - but if you suddenly want to move the same device to PORTB Pin 2 then it's all a bit of nightmare trying to find all the places to change.

Yes - you can simplify things by defining a set of macros of your own but now you are just adding to the confusion by having 'yet another way' and there is nothing to force you to use these new macros.

Existing libraries are unaware of the board you are using. For example: the processor on the Axon has lots of I/O pins that aren't connected at all on the Axon. So you can write a program that uses these pins only to realise your mistake and change to another pin. We prevent this by removing the definition of I/O pins that are redundant on the board you are using. The library provides a number of 'pre-built' boards but you can add extra ones for the custom boards you build.

But the biggest reason for change is traceability. This is impossible when a given I/O port may be switched on/off in lots of places. Removing all of this and making all I/O go through one function makes it easier to keep track of what's going on and be able to log any changes in appropriate ways. These ways may include:-

1. Output all I/O pin changes to the SoR'scope so you can a time graph on the PC
2. Output I/O changes to an LCD display connected to the robot
3. Setting a single breakpoint for all I/O changes

So for these reasons I've changed how you do I/O.



This file is automatically included from your system file and hence knows the pins it supports. For each pin we create a two character variable. The first character is the port and the second is the pin. So PORTD Pin 6 will be called D6. Having done that for each pin then the old variables for each port and pin (PORTB, DDRB, PINB etc) are deleted to stop you using them. Hence you have to go through the functions in this file.

## Standard Function Summary

	<a href="#"><u>pin_make_input(const IOPin* io, boolean pullup)</u></a> Convert an I/O pin into an input pin.
boolean	<a href="#"><u>pin_is_input(const IOPin* io)</u></a> Test if a given pin is an input pin.
	<a href="#"><u>pin_make_output(const IOPin* io, boolean val)</u></a> Convert an I/O pin into an output pin and set its initial state.
boolean	<a href="#"><u>pin_is_output(const IOPin* io)</u></a> Test if a given pin is an output pin.
	<a href="#"><u>pin_high(const IOPin* io)</u></a> Change an output pin to high.
	<a href="#"><u>pin_low(const IOPin* io)</u></a> Change an output pin to low.
	<a href="#"><u>pin_set(const IOPin* io, boolean val)</u></a> Allows you to set an output pin to high or low depending on the second parameter.
	<a href="#"><u>pin_toggle(const IOPin* io)</u></a> This will flip a given output pin.
boolean	<a href="#"><u>pin_is_high(const IOPin* io)</u></a> Test if a given pin is high.
boolean	<a href="#"><u>pin_is_low(const IOPin* io)</u></a> Test if a given pin is low.
	<a href="#"><u>pin_pulseOut(const IOPin* io, TICK_COUNT us, boolean activeHigh)</u></a> Emit a single pulse to the pin.
TICK_COUNT	<a href="#"><u>pin_pulseIn(const IOPin* io, boolean activeHigh)</u></a> Measure the duration of an incoming pulse.



## Standard Function Detail

### pin\_make\_input

`pin_make_input(const IOPin* io, boolean pullup)`

Convert an I/O pin into an input pin.

The 'pullup' parameter allows you to specify whether or not the pin should add a pull-up resistor. If in doubt then use FALSE.

So to change B6 to an input with no pull-up then you could write:-

```
pin_make_input(B6, FALSE);
```

or if you want to have a pull-up then:-

```
pin_make_input(B6, TRUE);
```

You can then test the input value on the pin using `pin_is_high` and/or `pin_is_low`.

---

### pin\_is\_input

`boolean pin_is_input(const IOPin* io)`

Test if a given pin is an input pin. Return TRUE if it is or FALSE if it is currently an output pin.

Example:

```
if(pin_is_input(B6)){
    // B6 is an input
}else{
    // B6 is an output
}
```

### pin\_make\_output

`pin_make_output(const IOPin* io, boolean val)`

Convert an I/O pin into an output pin and set its initial state.

The second parameter specifies the initial value of the output pin and should be TRUE if you want the output pin to be high, or FALSE if you want it to be low.

You can then change the output value on the pin using `pin_high`, `pin_low`, `pin_set`, `pin_toggle` and `pin_pulseOut`

---

### pin\_is\_output

`boolean pin_is_output(const IOPin* io)`

Test if a given pin is an output pin. Return TRUE if it is or FALSE if it is currently an input pin.

---



Example:

```
if(pin_is_output(B6)){
    // B6 is an output
}else{
    // B6 is an input
}
```

---

## **pin\_high**

`pin_high(const IOPin* io)`

Change an output pin to high.

For example to set the output pin B6 high then use:-

```
// Make B6 an output pin - you only need to do it once
// So if B6 is always an output pin then you may do this in appInitHardware
pin_make_output(B6,FALSE); // Start with a value of low
// Set B6 high
pin_high(B6);
```

If the pin is not currently configured as an output pin, ie you haven't called `pin_make_output` then calling `pin_high` will cause a runtime error to be indicated.

---

## **pin\_low**

`pin_low(const IOPin* io)`

Change an output pin to low.

For example to set the output pin B6 low then use:-

```
// Make B6 an output pin - you only need to do it once
// So if B6 is always an output pin then you may do this in appInitHardware
pin_make_output(B6, TRUE); // Initial value is high
// Set B6 low
pin_low(B6);
```

If the pin is not currently configured as an output pin, ie you haven't called `pin_make_output` then it will cause a runtime error to be indicated.

---

## **pin\_set**

`pin_set(const IOPin* io, boolean val)`

Allows you to set an output pin to high or low depending on the second parameter.

This is useful when the pin can be set high or low depending on the rest of your code. Normally you would use a boolean variable to store the result - ie TRUE if you want the pin to be set HIGH, or FALSE if you want it to be set LOW.

---





For example to change the output pin B6 then use:-

```
// Make B6 an output pin - you only need to do it once
// So if B6 is always an output pin then you may do this in appInitHardware
pin_make_output(B6, FALSE); // Start with a low value

// The variable used to note whether it should be set high or low
boolean setB6;

// Add your code here to change 'setB6' to TRUE or FALSE
...

// And then change the pin
pin_set(B6, setB6);
```

If the pin is not currently configured as an output pin, ie you haven't called `pin_make_output` then it will cause a runtime error to be indicated.

---

## **pin\_toggle**

```
pin_toggle(const IOPin* io)
```

This will flip a given output pin.

If it was low it becomes high, if it was high it becomes low.

For example to flip B6 then use:-

```
// Make B6 an output pin - you only need to do it once
// So if B6 is always an output pin then you may do this in appInitHardware
pin_make_output(B6, FALSE); // Start with B6 low
// Set B6 high
pin_high(B6);
// Toggle B6 - since it was high then it will become low
pin_toggle(B6);
// Toggle B6 - since it was low then it will go back to high
pin_toggle(B6);
```

If the pin is not currently configured as an output pin, ie you haven't called `pin_make_output` then it will cause a runtime error to be indicated.

The main advantage of this function is when sending out pulses via an output pin. Sometimes you want the pin to be high and send a low pulse. Other times the pin is low and you want to send a high pulse. So this can be achieved by toggling the pin, waiting for the required delay, and then toggling it back to its previous state.

Hence the following will set the pin low and then send a positive pulse:-



```
void appInitHardware(void){
    // Make B6 an output and set its normal state to low
    pin_make_output(B6, FALSE);
}

// Then in your main code you can send a pulse using
pin_toggle(B6);
delay_ms(10);
pin_toggle(B6);
```

However: there may be a scenario where the output pin needs to be high and a pulse is sent by setting the pin to low for a short duration. This could be done using:

```
void appInitHardware(void){
    // Make B6 an output and set its normal state to high
    pin_make_output(B6, TRUE);
}

// Then in your main code you can send a pulse using
pin_toggle(B6);
delay_ms(10);
pin_toggle(B6);
```

Note that the initial state is set within `appInitHardware` but the remaining code stays the same.

---

## **pin\_is\_high**

`boolean pin_is_high(const IOPin* io)`

Test if a given pin is high. This works for both input and output pins.

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.

So if you want your code to do something different depending on whether B6 is high or low then you could write:

```
if(pin_is_high(B6)){
    // B6 is high so do scenario 1
}else{
    // B6 is low so do scenario 2
}
```

---

## **pin\_is\_low**

`boolean pin_is_low(const IOPin* io)`

Test if a given pin is low. This works for both input and output pins.

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.

So if you want your code to do something different depending on whether B6 is high or low then you could write:

```
if(pin_is_low(B6)){
    // B6 is low so do scenario 1
}else{
    // B6 is high so do scenario 2
}
```

---

## pin\_pulseOut

`pin_pulseOut(const IOPin* io, TICK_COUNT us, boolean activeHigh)`

Emit a single pulse to the pin.

This will change the pin to an output pin and then send a pulse of the given number of microseconds.

On return the pin will be an output pin set low.

So to send a 10ms high pulse to B6 you write:-

```
pin_pulseOut(B6, 10000, TRUE);
```

and to send a 10ms low pulse to B6 you write:-

```
pin_pulseOut(B6, 10000, FALSE);
```

---

## pin\_pulseIn


`TICK_COUNT pin_pulseIn(const IOPin* io, boolean activeHigh)`

Measure the duration of an incoming pulse.


This will change the pin to an input and then measure and return the duration of an incoming pulse.

If the second parameter is TRUE then the duration is how long the pulse was set to high, otherwise it is how long it was set to low.

On return the pin will be set as an input and the returned value is in microseconds.

So if you want to measure an incoming pulse on B6 that is active high, ie  then you would write:-

```
TICK_COUNT us = pin_pulseIn(B6, TRUE);
```

But if you want to measure an incoming pulse on B6 that is active low, ie  then you would write:-

```
TICK_COUNT us = pin_pulseIn(B6, FALSE);
```



## led.h

Extends an output pin to cope with an LED.

Why bother? Well 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.

This file 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.

The file also allows you to register an 'in built' LED to use for indicating the current program status. When using a commercial board such as the Axon or the Roboduino then these boards have a built in LED and are automatically registered as the status LED. However: if you are just including a generic board such as sys/atmega168 then no status LED can be assumed. You are encouraged to add a status LED in your initialising code (applnitHardware) by calling 'statusLEDregister'.

Status LEDs are easier to use because you can use the statusLED\_on/off/set commands without worrying each time which pin it is connect to. This makes your code more portable from one board to another.

The main benefit of registering a status LED is that the library, and your own program, can indicate error conditions by flashing the status LED. See errors.h for more details.

For a more complex LED see segled.h

### Standard Function Summary

	<a href="#"><u>MAKE_LED(pin, activehigh)</u></a> Allows you to create a new LED.
	<a href="#"><u>LED_on(LED* led)</u></a> Turn the LED on.
	<a href="#"><u>LED_off(LED* led)</u></a> Turn the LED off.
	<a href="#"><u>LED_set(LED* led, boolean value)</u></a> Allows you to turn an LED on or off depending on a variable passed in 'value'.
	<a href="#"><u>statusLEDregister(const IOPin* pin, boolean activeHigh)</u></a> Register which pin the status LED is attached to.
	<a href="#"><u>statusLED_on()</u></a> Turn on the status LED



---

## Standard Function Summary

	<a href="#"><code>statusLED_off()</code></a>
	Turn off the status LED
	<a href="#"><code>statusLED_set(boolean value)</code></a>
	Turn the status LED on or off.

---

## Standard Function Detail

### MAKE\_LED

`MAKE_LED(pin, activehigh)`

Allows you to create a new LED. The pin parameter says which pin it is connected to and the activehigh parameter should be TRUE if the LED is lit by setting the output pin high or FALSE if not.

Example - to create an LED on B3 that is turned on when the output is high then:

```
#include "led.h"
LED myLED = MAKE_LED(B3,TRUE);
```

---

### LED\_on

`LED_on(LED* led)`

Turn the LED on.

```
LED myLED = MAKE_LED(B6,TRUE);
LED_on(&myLED);
```

---

### LED\_off

`LED_off(LED* led)`

Turn the LED off.

```
LED myLED = MAKE_LED(B6,TRUE);
LED_off(&myLED);
```

---

### LED\_set

`LED_set(LED* led, boolean value)`

Allows you to turn an LED on or off depending on a variable passed in 'value'.

If 'value' is FALSE then the LED is turned off, otherwise it is turned on.

---



## **statusLEDregister**

`statusLEDregister(const IOPin* pin, boolean activeHigh)`

Register which pin the status LED is attached to.

This is done for you automatically for boards like the Axon and Roboduino.

The first parameter specifies the IOPin and the second parameter indicates whether the LED lights when the output pin is set high or low.

---

## **statusLED\_on**

`statusLED_on()`

Turn on the status LED

---

## **statusLED\_off**

`statusLED_off()`

Turn off the status LED

---

## **statusLED\_set**

`statusLED_set(boolean value)`

Turn the status LED on or off.

---



## segled.h

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.

To create a single led we must create it: specifying the output pins for each segment. So, for example, in sys/Axon2.h we specify:

```

#include "../segled.h"
SEGLED led_display = MAKE_SEGLED(C3,C2,C0,D6,D7,C4,C5,null,FALSE);

```

There are two things to note here:

1. The first 8 parameters specify the output pins for the segments 'a' to 'h' and you will see that the value for 'h' is 'null' - this means that the decimal point (segment h) is not connected - because we are using elsewhere for the status LED.
2. The last parameter 'FALSE' denotes whether the segments are lit by a high output. Since we say 'FALSE' then each segment is lit by setting the output pin to low.

On an Axon II the LED display is built onto the board and so the above declaration is automatically done for you in axon2.h. If you have added an LED display yourself then you will need to define it yourself in a manner similar to above.

Ok - so we have an LED - but what can we do with it?

Well the first thing is that we can write a character to it using `segled_put_char(SEGLED* led, uint8_t byte)` This allows us to display an individual character on a specific LED. The supported characters are:

- A to Z
- a to z



- 0 to 9
- . (full stop or decimal point)
- - (minus sign or hyphen)
- {[ (open bracket)
- ]} (close bracket)
- Anything else is show as a space - ie all segments are off.

Whilst this is useful - we often want to display a whole message rather than a single character.

So if we had 'one or more' such devices then we could create a calculator style display ie one line of many digits.

This library allows you to do just that - and then scroll the message across the display. This is called a 'marquee' and Windows users may well have used the screen saver with that name/effect.

We have seen above how to define a single 'digit' using MAKE\_SEGLED but we could of course use additional output pins, and hardware, to add additional 'digits'.

So the first step to create a marquee is to specify the list of digits that are involved. Example: if we are only using the single on-board digit for the Axon II then this can be done via:

```
// Create list of 7 segment leds. There is only 1 on the Axon II
SEGLED_LIST _leds[] = {&led_display};
```

Now we create a 'marquee' to scroll messages across that list of digits.

```
// Create a marquee to scroll text along these leds
MAKE_WRITER(marquee_put_char); // Forward definition
MARQUEE marquee = MAKE_MARQUEE(_leds_, (TICK_COUNT)600000, (TICK_COUNT)2000000,
&marquee_put_char);
// Now define a Writer that can send data to the marquee
MAKE_WRITER(marquee_put_char){
    return marqueeSendByte(&marquee,byte);
}
```

The first parameter to the MAKE\_MARQUEE is the list of LED digits used to show a portion of the message.

The second parameter is the delay in microseconds (ie 250ms) for scrolling.

The third parameter is the delay in microseconds (ie 1 second) to pause at the end of the message before starting to show the message again. A value of 0 means that the message will only be shown once - a 'one-shot message'.

The final parameter is the address of a Writer routine which can be used to write data to this marquee.





That may sound complex - but dont worry - the axon2.h already defines all of the above for the built in LED digit.

For Axon II builds then note that the defaults are:

- Error codes continue to be flashed on the decimal point but are also marquee'd using the single digit display unless redirected by using `setErrorLog`
- The default `rprintf` destination is also to be marquee'd across the single digit display unless redirected by using `rprintfInit(Writer writer)`

The delays can also be modified at runtime. For example the values specified in the above example of `MAKE_MARQUEE` can be modified using:

- `marqueeSetCharDelay(MARQUEE* marquee, TICK_COUNT delay)`
- `marqueeSetEndDelay(MARQUEE* marquee, TICK_COUNT delay)`

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.

## Standard Function Summary

<code>void</code>	<a href="#"><code>segled_init(SEGLED* led)</code></a> Initialise the LED ie turn all segments off.
<code>uint8_t</code>	<a href="#"><code>segled_put_char(SEGLED* led, uint8_t byte)</code></a> Output a character to the LED display.
<code>void</code>	<a href="#"><code>segled_on(SEGLED* led, SEGLED_SEGMENT segment)</code></a> Turn on an individual segment of the LED display.
<code>void</code>	<a href="#"><code>segled_off(SEGLED* led, SEGLED_SEGMENT segment)</code></a> Turn off an individual segment of the LED display.
<code>void</code>	<a href="#"><code>segled_set(SEGLED* led, SEGLED_SEGMENT segment, boolean value)</code></a> Turn on or off an individual segment of the LED display.
<code>Writer</code>	<a href="#"><code>marqueeGetWriter(MARQUEE* marquee)</code></a> Returns a Writer that can be used to write to the marquee.

## Advanced Function Summary

	<a href="#"><code>marqueeStop(MARQUEE* marquee)</code></a> Stop an auto-repeating marquee message.
--	---



## Advanced Function Summary

boolean

[marqueelsActive\(const MARQUEE \\* marquee\)](#)

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.

[marqueeSetCharDelay\(MARQUEE\\* marquee, TICK\\_COUNT delay\)](#)

Change the duration (in microseconds) for scrolling characters in the marquee.

[marqueeSetEndDelay\(MARQUEE\\* marquee, TICK\\_COUNT delay\)](#)

Change the delay for an auto-repeating message or make it non-repeating by specifying a delay of 0.

## Standard Function Detail

### segled\_init

`void segled_init(SEGLED* led)`

Initialise the LED ie turn all segments off.

### segled\_put\_char

`uint8_t segled_put_char(SEGLED* led, uint8_t byte)`

Output a character to the LED display.

The return value is the same character.

The first parameter specifies the 8 segment LED to output the data to, and the second parameter is the character to be output.

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



NB If the LED display does not support the decimal point (segment 'h') then the '.' character will be shown by using segment 'd' instead. This is done so that numbers, with decimal points, can be shown with some indication as to where the decimal point should occur.

So we now have a single character display, well on the Axon II at least. But of course there is nothing to stop you adding hardware for additional ones: on the Axon II or any other board - you are only limited by output pins. The result is a calculator like display of one or more 'digits'.

Whilst it is sometimes useful to be able to set each 'digit' to its own value independently it is also useful to be able to use this calculator display to display a single message. Often this message is longer than the number of available 'digits'. To overcome this problem we use what is known as a 'marquee'. Windows users may well be familiar with 'marquee' as a screen saver but if you are not then it is also similar to those billboards - ie a scrolling message.

In order to create a marquee effect we need to tell the library about all the individual digits that can be used to display a portion of the message. This would be at least one but may be more.

Looking at axon2.h we can see how it is defined for a single 'digit':

```
// Define the on-board digit display - ignoring the decimal point used as the
status led
SEGLED led_display = MAKE_SEGLED(C3,C2,C0,D6,D7,C4,C5,null,FALSE);
```

```
// Create list of 7 segment leds. There is only 1 on the Axon II
SEGLED_LIST _leds_[] = {&led_display};
```

```
// Create a marquee to scroll text along these leds
MARQUEE marquee = MAKE_MARQUEE(_leds_, (TICK_COUNT)600000,
(TICK_COUNT)1000000);
```

led\_display is the definition of a single digit.

\_leds\_ is the list of 'digits' across the calculator display - there is only one

The MAKE\_MARQUEE command specifies the list of \_leds\_ to use, the TICK\_COUNT in microseconds to use when scrolling (600ms), and the TICK\_COUNT to pause before repeating the message (1 second).

Since the Axon II has a built-in LED digit then any error codes are automatically defaulted to be displayed as a marquee using this digit unless subsequently redirected by you by calling setErrorLog.



## segled\_on

```
void segled_on(SEGLED* led, SEGLED_SEGMENT segment)
```

Turn on an individual segment of the LED display.

The first parameter is the address of the display.

The second parameter is the segment and can be any of the following:

```
SEGMENT_A  
SEGMENT_B  
SEGMENT_C  
SEGMENT_D  
SEGMENT_E  
SEGMENT_F  
SEGMENT_G  
SEGMENT_H
```

On the Axon II the on board LED display is called 'led\_display'. So to turn on segment E then we write:

```
segled_on(&led_display, SEGMENT_E);
```

---

## segled\_off

```
void segled_off(SEGLED* led, SEGLED_SEGMENT segment)
```

Turn off an individual segment of the LED display.

The first parameter is the address of the display.

The second parameter is the segment and can be any of the following:

```
SEGMENT_A  
SEGMENT_B  
SEGMENT_C  
SEGMENT_D  
SEGMENT_E  
SEGMENT_F  
SEGMENT_G  
SEGMENT_H
```

On the Axon II the on board LED display is called 'led\_display'. So to turn off segment E then we write:

```
segled_off(&led_display, SEGMENT_E);
```

---

## segled\_set

```
void segled_set(SEGLED* led, SEGLED_SEGMENT segment, boolean value)
```

Turn on or off an individual segment of the LED display.



The first parameter is the address of the display.

The second parameter is the segment and can be any of the following:

```
SEGMENT_A
SEGMENT_B
SEGMENT_C
SEGMENT_D
SEGMENT_E
SEGMENT_F
SEGMENT_G
SEGMENT_H
```

The last parameter is a boolean value. If this is TRUE then the segment is turned on, otherwise it is turned off.

On the Axon II the on board LED display is called 'led\_display'. So to turn off segment E then we could write:

```
segled_set(&led_display, SEGMENT_E, FALSE);
```

---

## marqueeGetWriter

Writer marqueeGetWriter(MARQUEE\* marquee)

Returns a Writer that can be used to write to the marquee.

The returned writer is normally passed to rprintfInit to redirect rprintf to the marquee.

For example on the Axon II:-

```
rprintfInit( marqueeGetWriter(&marquee) );
```

will redirect all rprintf statements to output to the onboard LED.

**Advanced Function Detail**

## marqueeStop

marqueeStop(MARQUEE\* marquee)

Stop an auto-repeating marquee message.

If there is an existing message being scrolled repetitively then this will cause it to be stopped in a clean fashion. ie not necessarily immediately but will at least guarantee that it will not start from the beginning again.

---

## marqueeIsActive

boolean marqueeIsActive(const MARQUEE \* marquee)

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.

---

### **marqueeSetCharDelay**

`marqueeSetCharDelay(MARQUEE* marquee, TICK_COUNT delay)`

Change the duration (in microseconds) for scrolling characters in the marquee.

Note that this may not take effect until the next message scroll.

---

### **marqueeSetEndDelay**

`marqueeSetEndDelay(MARQUEE* marquee, TICK_COUNT delay)`

Change the delay for an auto-repeating message or make it non-repeating by specifying a delay of 0.

Note that this may not take effect immediately.



## libdefs.h

Defines various macros and datatypes that are used in the rest of the library.

### Standard Function Summary

	<a href="#"><u>MIN(a,b)</u></a> This macro takes two numbers as its arguments and will return the smallest one ie the one closest to negative infinity.
	<a href="#"><u>MAX(a,b)</u></a> This macro takes two numbers as its arguments and will return the largest one ie the one closest to positive infinity.
	<a href="#"><u>ABS(x)</u></a> This macro takes one number as its arguments and will return the absolute value.
	<a href="#"><u>PI</u></a> Gives the value of the constant PI.
	<a href="#"><u>CLAMP(val, min, max)</u></a> A macro to clamp a value to be within a given range.
<code>uint16_t</code>	<a href="#"><u>get_uint16(const void* buf,size_t offset)</u></a> Returns a 16 bit integer from a buffer.
<code>uint32_t</code>	<a href="#"><u>get_uint32(const void* buf,size_t offset)</u></a> Returns a 32 bit integer from a buffer.
	<a href="#"><u>set_uint16(const void* buf,size_t offset,uint16_t data)</u></a> Stores a 16 bit integer into a buffer.
	<a href="#"><u>set_uint32(const void* buf,size_t offset,uint32_t data)</u></a> Stores a 32 bit integer into a buffer.

### Advanced Function Summary

	<a href="#"><u>CRITICAL_SECTION</u></a> Bracket a section of code that should not be interrupted.
	<a href="#"><u>CRITICAL_SECTION_START</u></a> Deprecated
	<a href="#"><u>CRITICAL_SECTION_END</u></a> 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 '+'.

### PI

PI

Gives the value of the constant PI.

### 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\_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 thorough to most significant byte.

The parameters specify a memory array, a byte offset into that array, and the value to be written.

---

<h2><b>Advanced Function Detail</b></h2>
--

## **CRITICAL\_SECTION**

`CRITICAL_SECTION`

Bracket a section of code that should not be interrupted.

Since most ATMegs 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 interrupt 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  $1 \times 256 + 255 = 511$ . 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.



## motorPWM.h

Controls DC motors using pulse width modulation (PWM).

This file is a 'catch all' in case your specific motor driver is not listed in "*DC Motors*" (see page 888)

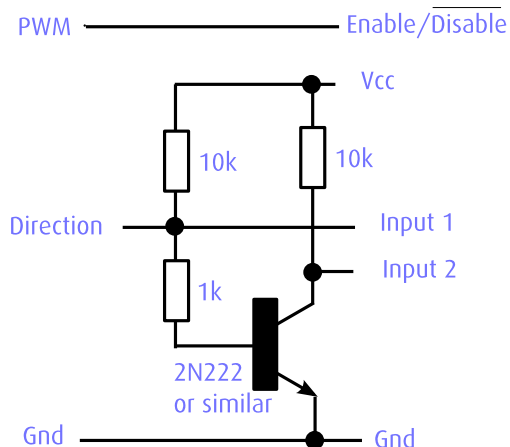
This module assumes that your motor driver has three inputs which can be used to achieve the following states:-

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

### Two Pin Operation

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 to 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](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 microcontroller provides.



If you attempt to use pins that are not valid: say because they are not PWM pins, or the timer doesn't provide hardware PWM then error codes will be set on the status LED.

It's very easy to use: you just give it a list of motors and that's it.

Note that you cannot connect a DC Motor directly to the microprocessor board - as the current requirements are too big and you will fry the chip. So the outputs from the board will need to be fed into a motor controller board.

Here is an example of the code in your application:-

```
#include "motorPWM.h"
// Define two motors
MOTOR left = MAKE_MOTOR(FALSE, B1,B4);
MOTOR right = MAKE_MOTOR(TRUE , B2,B5);
// Create the list - remember to place an & at the
// start of each motor name
MOTOR_LIST motors[] = {&left,&right};
// Create a driver for the list of motors
MOTOR_DRIVER bank1 = MAKE_MOTOR_DRIVER(motors);
// In my initialising code - pass the list of motors to control
void appInitHardware(void){
    // Initialise the servo controller
    motorPWMInit(&bank1);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Give each servo a start value - make them go full ahead
    act_setSpeed(&left,DRIVE_SPEED_MAX);
    act_setSpeed(&right,DRIVE_SPEED_MAX);
    return 0;
}
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    .. nothing to do. But in reality I would use act_speed to change the speed of
    each motor ...
    return 0;
}
```

## Three Pin Operation

If you don't want to add the additional circuitry then you can still use this module but you will need to use 3 I/O pins rather than two.

The only difference to the code above is to replace the MAKE\_MOTOR calls with MAKE\_MOTOR\_3\_PIN instead:-



```
MOTOR left = MAKE_MOTOR_3_PIN(FALSE, B1,B4,B3);
MOTOR right = MAKE_MOTOR_3_PIN(TRUE , B2,B5,B6);
```

## Standard Function Summary

	<a href="#">motorPWMInit(MOTOR_DRIVER* driver)</a> Tells the motor driver what motors it controls.
	<a href="#">MAKE_MOTOR(inverted, motorPin, directionPin)</a> Creates a DC motor.
	<a href="#">MAKE_MOTOR_3_PIN(inverted, pwmPin, input1,input2)</a> Creates a DC motor.

## Standard Function Detail

### motorPWMInit

`motorPWMInit(MOTOR_DRIVER* driver)`

Tells the motor driver what motors it controls.

### MAKE\_MOTOR

`MAKE_MOTOR(inverted, motorPin, directionPin)`

Creates a DC motor.

The inverted parameter specifies whether the direction of the motor should be reversed: TRUE or FALSE.

The second parameter specifies an IOPin that can produce hardware PWM.

The final parameter specifies any IOPin that is used to specify the rotation direction.

### MAKE\_MOTOR\_3\_PIN

`MAKE_MOTOR_3_PIN(inverted, pwmPin, input1,input2)`

Creates a DC motor.

The inverted parameter specifies whether the direction of the motor should be reversed: TRUE or FALSE.

The second parameter specifies an IOPin that can produce hardware PWM.

The final two parameters specify the IOPins that are used to control both sides of the H-Bridge to control the rotation direction.



## 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](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\(PID\\* pid,float setPoint\)](#)

Specify the goal, or 'set point' for the PID.

float

[pidSetActual\(PID\\* pid,float actual\)](#)

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 or ATmega32. The ATmega168, ATmega328P, 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

	<a href="#"><code>pin_change_attach(const IOPin* io, PinChangeCallback callback, void* data)</code></a> Attach a callback routine.
	<a href="#"><code>pin_change_detach(const IOPin* io)</code></a> 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.



## pwm.h

Provides a generic way of generating a hardware PWM signal.

This can generate a hardware PWM signal on a timer output pin - check the sys/\*.h file for your board to see which pins are available. Hardware PWM is useful as it can be used either as a fallback if your specific motor driver is not currently supported or for miscellaneous tasks such as dimming LEDs.

PWM has two main requirements:

1. The frequency: ie the 'off time' + the 'on time'
2. The duty cycle: the percentage of time 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  $(16\text{MHz} / 1024) / 256 = 61\text{Hz}$ . So if you want really low frequencies then you would be better choosing a 16 bit timer instead.

Because these calculations are quite complex then you may want to make the initialisation routines return the 'actual' value into a variable (by using the last parameter). You can then decide if the actual value is close enough to the value you wanted or not.

When using the 'init' functions there are two 'short cuts' that can be used for the required frequency:-



PWM\_SLOWEST - Will set up the slowest possible PWM cycle time. Useful for things like blinking LEDs.

PWM\_FASTEST - Use the fastest possible PWM cycle time. Useful for dimming LEDs, speech, digital-to-analogue conversion etc.

## Standard Function Summary

boolean

[pwmInitHertz\(const IOPin\\* pin, uint32\\_t hertz, DUTY\\_CYCLE duty, uint32\\_t\\* actualHertz\)](#)

Initialise PWM using the given frequency in Hertz.

boolean

[pwmInitDeciHertz\(const IOPin\\* pin, uint32\\_t deciHertz, DUTY\\_CYCLE duty, uint32\\_t\\* actualDeciHertz\)](#)

Initialise PWM using the given frequency in DeciHertz - tenths of a Hertz - ie 1 = 0.1.

[pwmSetDutyCycle\(const IOPin\\* pin, DUTY\\_CYCLE duty\)](#)

Sets the PWM duty cycle on a pin that has already been initialised successfully.

DUTY\_CYCLE

[pwmGetDutyCycle\(const IOPin\\* pin\)](#)

Returns the current duty cycle for an IO pin that has already been set up for PWM.

[pwmOff\(const IOPin\\* pin\)](#)

Stop sending the PWM output to a given pin.

## Standard Function Detail

### pwmInitHertz

```
boolean pwmInitHertz(const IOPin* pin, uint32_t hertz, DUTY_CYCLE duty,
uint32_t* actualHertz)
```

Initialise PWM using the given frequency in Hertz.

The function will return FALSE if there is a problem - for example if you are trying to use a non-PWM pin or it will return TRUE if everything is ok.

Parameters:

pin - The IOPin to use. Check the sys/\*.h file to see what PWM timer output pins are available

hertz - The frequency

duty - The initial setting for the duty cycle (a value between 0 and 100).



actualHertz - The address of an uint32\_t variable in your calling code. This variable will be filled in, on return, with the actual frequency that will be used. If you are not interested in this value then you can use **null** for this parameter.

---

### **pwmInitDeciHertz**

```
boolean pwmInitDeciHertz(const IOPin* pin, uint32_t deciHertz,  
DUTY_CYCLE duty, uint32_t* actualDeciHertz)
```

Initialise PWM using the given frequency in DeciHertz - tenths of a Hertz - ie 1 = 0.1Hz, 10=1Hz etc.

This function is similar to the other 'init' function but is mainly used to set a frequency of less than 1 Hertz.

The function will return FALSE if there is a problem - for example if your are trying to use a non-PWM pin or it will return TRUE if everything is ok.

Parameters:

pin - The IOPin to use. Check the sys/\*.h file to see what PWM timer output pins are available

deciHertz - The frequency in tenths of a Hertz

duty - The initial setting for the duty cycle (a value between 0 and 100).

actualDeciHertz - The address of an uint32\_t variable in your calling code. This variable will be filled in, on return, with the actual frequency, in DeciHertz, that will be used. If you are not interested in this value then you can use **null** for this parameter.

---

### **pwmSetDutyCycle**

```
pwmSetDutyCycle(const IOPin* pin, DUTY_CYCLE duty)
```

Sets the PWM duty cycle on a pin that has already been initialised successfully.

The duty cycle is a percentage, 0 to 100.

---

### **pwmGetDutyCycle**

```
DUTY_CYCLE pwmGetDutyCycle(const IOPin* pin)
```

Returns the current duty cycle for an IO pin that has already been set up for PWM.

---

### **pwmOff**

```
pwmOff(const IOPin* pin)
```

Stop sending the PWM output to a given pin.

---



## 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). Although most microprocessors can include other libraries to cope with these more complex 'real' numbers it does so at the cost of increasing the size of your program.

So this library allows you to set some 'switches' at the top of your main program to suggest what kind of variables should be allowed.

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 - before it is used. This is done using the rprintfInit function and it is important that you call rprintfInit, in your initialisation code, before you start calling this library otherwise the output will just be discarded.

### Standard Function Summary

Writer	<a href="#">rprintfInit(Writer writer)</a> Tells rprintf where to send its output to.
	<a href="#">rprintfChar(unsigned char c)</a> Output a single character.
	<a href="#">rprintfCharN(unsigned char c, size_t repeat)</a> Print a character a given number of times.
	<a href="#">rprintfStr(char str[])</a> Print a string that is stored in RAM.
	<a href="#">rprintfProgStr(const prog_char str[])</a> Output a constant string stored in program memory.
	<a href="#">rprintfProgStrM(string)</a> Output a fixed string from program memory.
	<a href="#">rprintfCRLF()</a> Move the output to a new line.
	<a href="#">rprintfu04(unsigned char data)</a> Output the low 4 bits of 'data' as one hexadecimal character





## Standard Function Summary

	<a href="#"><u>rprintfu08(unsigned char data)</u></a> Output the parameter as two hexadecimal characters.
	<a href="#"><u>rprintfu16(unsigned short data)</u></a> Output the parameter as 4 hexadecimal digits.
	<a href="#"><u>rprintfu32(unsigned long data)</u></a> Output the parameter as an 8 digit hexadecimal number.
	<a href="#"><u>rprintfNum(char base, char numDigits, boolean isSigned, char padchar, long n)</u></a> Allows you to print a number in any base using different padding characters.
	<a href="#"><u>rprintf(format, args...)</u></a> Output a formatted string to the output.

## Advanced Function Summary

char	<a href="#"><u>hexchar(char x)</u></a> Convert binary to hexadecimal digit.
	<a href="#"><u>rprintfFloat(char numDigits, double x)</u></a> Prints a floating point number.
	<a href="#"><u>rprintfMemoryDump(const void* data, size_t offset, size_t len)</u></a> 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.

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 sent 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 ...
```

---

## 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(char str[])("Hello World");
```

This is not ideal since the text 'Hello World' will actually eat into our valuable RAM space.

---

## **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)`

Output 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, else 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.

There are two versions of this routine which depend on whether or not you have declared `PRINTF_COMPLEX` prior to including this file. If you have not (the default) then the only formatting characters are `%d`, `%u`, `%x`, `%s` and `%c`. Here are some examples:-

```
#include "rprintf.h"
rprintf("Character=%c Decimal number=%d String=%s", 'A', 1234, "Help");
```

This will output "Character=A Decimal number=1234 String=Help"

If you have declared `PRINTF_COMPLEX` then it will include some additional code which allows: `%d`, `%ld`, `%u`, `%lu`, `%o`, `%x`, `%c` and `%s` as well as the width, precision and padding modifiers from the C specification. Here are some examples:-

```
#define RPRINTF_COMPLEX
#include "rprintf.h"
int16_t num1 = 1234; // a standard 'signed int'
int32_t num2 = 1234567; // a standard 'long int'
int16_t width = 6;
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'
- %lo Expects an 'unsigned long' or 'uint32\_t'
- %x Expects an 'uint8\_t' or an 'uint16\_t'
- %lx Expects an 'uint32\_t'
- %% Expects no parameter and just outputs the '%' character
- %c Expects a 'char', 'signed char', 'int8\_t', 'int16\_t', 'int'

## 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. This is currently used to flash the status LED when an error occurs. 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 servos. However it is fine for less time critical applications.

Since the status LED is flashed using this code to signify an error then this code is always automatically included.

The scheduler can only queue up a maximum number of jobs. The default is currently 4 jobs but you can change this by defining a value for SCHEDULER\_MAX\_JOBS prior to including your system file.

Example: if we are using the Axon but want the scheduler to cope with 8 jobs then:-

```
#define SCHEDULER_MAX_JOBS 8
#include "sys/axon.h"
```

### Advanced Function Summary

[scheduleJob\(SchedulerCallback callback, SchedulerData data, TICK\\_COUNT start, TICK\\_COUNT delay\)](#)

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 the 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 1000ms after the previous call back **started** whereas the second will start the callback routine 1000ms after the previous call back has **finished**.

By default the scheduler can cope with a maximum of 4 queued jobs. However you can change this by defining SCHEDULER\_MAX\_JOBS at the top of your program eg.

```
// Allow up to 16 scheduled jobs
#define SCHEDULER_MAX_JOBS 16
```



## **servos.h**

This is a servo driver for controlling servos that are plugged directly into your board and uses either software PWM, or hardware PWM, to send commands to each servo. If you are using a commercial servo controller then refer to "Servos" (see page 322) .

If it is software based then each servo can be connected to **any** I/O pin. However it also means that its not as precise as a hardware PWM solution, When running in an application that has other interrupts enabled then an unmodified servo may judder slightly as the pulse widths will not be exact. However: this juddering is not noticeable with a modified servo. For software PWM you should call 'servosInit'.

If it is hardware based then each servo must be connected to an IOPin that is the PWM output for a 16 bit timer and you must call 'servoPWMInit'.

Most servos require a pulse that is between 1ms and 2ms to control them. This pulse needs to be sent every 20ms or so - some servos are more tolerant and may be happy with every 40ms. Given the longest pulse time of 2ms then this code could drive anywhere between 10 (20ms/2ms) or 20 (40ms/2ms) servos depending on the repeat pulse timing required by the servos.

So the choice is to either use this file to control a large number of servos versus using hardware PWM to control 2 or 3 servos with better accuracy. Both methods will require one Timer. Of course you can mix and match the approaches e.g. use software PWM to control modified servos (as their speed settings aren't that accurate) and hardware PWM to control any more precise unmodified servos.,

It's very easy to use: you just give it a list of servos and an available 16 bit timer and that's it. Whether you are using hardware or software PWM then the pulses are sent out automatically in the background.

Once initialised then each servo can be controlled individually using the actuator commands. See "*actuators.h*" (see page 23) . As discussed in that section, a servo is handled the same way whether it is a modified servo or an unmodified servo. For an unmodified servo then just remember that the act\_SetSpeed function sets the position of the servo rather than the speed.

Here is an example of the code in your application:-



```

#include "servos.h"
// Define two servos
SERVO left = MAKE_SERVO(FALSE, D1,1500, 500);
SERVO right = MAKE_SERVO(TRUE , D2,1500, 500);

// Create the list - remember to place an & at the
// start of each servo name
SERVO_LIST servos[] = {&left,&right};

// Create a driver for the list of servos
SERVO_DRIVER bank1 = MAKE_SERVO_DRIVER(servos);

// In my initialising code - pass the list of servos to control
void appInitHardware(void){
    // Initialise the servo controller using software PWM
    servosInit(&bank1, TIMER1);
// OR use this line instead for hardware PWM
    servoPWMInit(&bank1);
}
// Initialise the servos
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Give each servo a start value - make them go full ahead
    act_setSpeed(&left,DRIVE_SPEED_MAX);
    act_setSpeed(&right,DRIVE_SPEED_MAX);
    return 0;
}

// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    .. nothing to do. But in reality I would use act_speed to change the speed of
    each servo ...
    return 0; // no delay
}

```

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.

## Standard Function Summary

[MAKE\\_SERVO\(inverted, iopin, center, range\)](#)

Define a servo.



## Standard Function Summary

	<a href="#"><code>servosInit(SERVO_DRIVER* driver, const Timer* timer)</code></a> Drive the list of servos using software PWM.
	<a href="#"><code>servoPWMInit(SERVO_DRIVER* driver)</code></a> Tells the servo driver what servos it controls.
	<a href="#"><code>servoSerialInit(SERVO_DRIVER* driver, UART* uart, BAUD_RATE baud, SERVO_PROTOCOL protocol)</code></a> Initialise a bank of servos being driven by serial servo controller.
	<a href="#"><code>servosSetConnected(const SERVO_DRIVER* driver, boolean connect)</code></a> Connect or disconnect a group of servos.
	<a href="#"><code>servosSetSpeed(const SERVO_DRIVER* driver, DRIVE_SPEED speed)</code></a> Set the speed for a group of servos.
	<a href="#"><code>servosCenter(SERVO_LIST* const servos, UART* uart)</code></a> 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.

## Advanced Function Summary

	<a href="#"><code>servoSetConfig(SERVO* servo, uint16 t center, uint16 t range)</code></a> Changes the servo centre position and range at runtime.
--	---

## Standard Function Detail

### MAKE\_SERVO

`MAKE_SERVO(inverted, iopin, center, range)`

Define a servo.

The parameters are:

`inverted` - TRUE or FALSE  
`iopin` - The processor pin it is connected to. e.g. B5 or D2 etc  
`center` - The center point of the servo. See below.  
`range` - The range on either side of center. It should never be greater than 'center'. See below

For example: if you have a differential drive robot with servo motors on D2 and D3 then we could define them in our code as follows:-



```
SERVO left = MAKE_SERVO(FALSE, D2, 1500, 500);  
SERVO right = MAKE_SERVO(TRUE, D3, 1500, 500);
```

Having defined the servos then we can set the speed etc with other functions in this file. However: the servo will not start moving until it has been attached to some other code that decides how to drive it - ie is it hardware PWM or software PWM or some other method.

Assuming that the servo is attached to a driver then what do those 'center' and 'range' parameters mean? The 'center' parameter is a number and indicates the pulse width (in microseconds) required to move the servo to its center position. For most servos this would be '1500' ie 1.5ms. The 'range' parameter says how much the pulse width changes either side of center'. For most servos this would be '500' ie 0.5ms - meaning the pulse width varies between 1ms and 2ms.

In practise you will need to fiddle with these values for each of your servos as they may not all be setup in the same way. Assuming you have modified a servo for continuous rotation then here's the best way to set these values.

1. Set the center=1500 and the range=0
2. Run the program
3. The servos should be stationary. If not then adjust the center value until they stop moving
4. Now change the 'range' value to 500 and change your program to set both motors to `DRIVE_SPEED_MAX`
5. The motors should now turn in the same direction. If not then you've probably forgotten to invert one of the servos.
6. If your servos dont rotate at the same speed then try decreasing the range value of the fastest servo until they do.

---

## **servosInit**

```
servosInit(SERVO_DRIVER* driver, const Timer* timer)
```

Drive the list of servos using software PWM.

The servos parameter is the list of servos that it controls.

The Timer parameter is ANY available 16 bit timer. Look at the 'sys/\*.h' for your board and locate any 16 bit timer - it doesn't matter if it has an IO Pin output or not as the timer is only used to generate internal interrupts to drive the servos. The library will use the first two channels of the timer.

If in doubt then use `TIMER1` as this works on most devices.

---

## **servoPWMInit**

```
servoPWMInit(SERVO_DRIVER* driver)
```

Tells the servo driver what servos it controls.



Since this uses hardware PWM to drive each servo then there is a restriction as to what IO pins the servos can be connected to.

Look at the timer list in your 'sys/\*.h'. Your servos must be connected to one of the 16 bit timer channels that has an IO pin to connect it to. Use of other pins will cause a runtime error.

---

### **servoSerialInit**

```
servoSerialInit(SERVO_DRIVER* driver, UART* uart, BAUD_RATE baud,  
SERVO_PROTOCOL protocol)
```

Initialise a bank of servos being driven by serial servo controller.

See "*Generic/Serial Servo Controller*" (see page 330) for an example of how to use it.

---

### **servosSetConnected**

```
servosSetConnected(const SERVO_DRIVER* driver, boolean connect)
```

Connect or disconnect a group of servos.

This is the equivalent of calling `act_isConnected(const __ACTUATOR_COMMON* act)` for each servo that is attached to the driver.

The second parameter should be TRUE to 'connect' the servos and FALSE to 'disconnect' the servos.

Two shorthand forms exist:

```
servosConnect(driver); // To start send pulses to the servos  
servosDisconnect(driver); // To stop sending pulses to the servos
```

---

### **servosSetSpeed**

```
servosSetSpeed(const SERVO_DRIVER* driver, DRIVE_SPEED speed)
```

Set the speed for a group of servos.

This is the equivalent of calling `act_setSpeed(__ACTUATOR_COMMON* act, DRIVE_SPEED speed)` for each servo that is attached to the driver so that all the servos have the same speed.

---

### **servosCenter**

```
servosCenter(SERVO_LIST* const servos, UART* uart)
```

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.

---



All of the servos should be set up and configured as described elsewhere in this manual. You should then call this function from your appControl main loop.

Here is an example of a program for the AxonII that uses this function:-

```
#include <sys/Axon2.h>
```

```
// Connect to terminal program via this uart at 115200 baud
#define THE_UART UART3
```

```
#include <servos.h>
#include <_uart_common.h>
```

```
// Define servos for the left side of my Brat humanoid
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 bank1_list[] = {&servo1,&servo2,&servo3};
SERVO_DRIVER bank1 = MAKE_SERVO_DRIVER(bank1_list);
```

```
// Define servos for the right side of my Brat humanoid
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 bank2_list[] = {&servo4,&servo5,&servo6};
SERVO_DRIVER bank2 = MAKE_SERVO_DRIVER(bank2_list);
```

```
// Create a list of ALL servos across ALL banks
// The ordering in this list is the ordering used by the terminal program
SERVO_LIST all[] = {&servo1,&servo2,&servo3,&servo4,&servo5,&servo6};
```

```
//initialize hardware, ie servos, UART, etc.
void appInitHardware(void){
    servosInit(&bank1, TIMER1);
    servosInit(&bank2, TIMER3);

    //setup UART on USB at the required baud rate
    uartInit(THE_UART, 115200);
}
```



```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
```

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    servosCenter(all,THE_UART); // 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 your code and enter these values in the servo definitions.

## Advanced Function Detail

### **servoSetConfig**

```
servoSetConfig(SERVO* servo, uint16_t center, uint16_t range)
```

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 your MAKE\_SERVO constructor.

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:

```
servoSetConfig(&servo1, 1450, 475);
```





## **spi.h**

Routines for handling communications with other devices by using the hardware SPI interface.

The SPI interface comes in two parts:

1. Decide how you want to implement the bus
2. Define the slave devices you want to be attached to that bus

The bus can be implemented using the SPI hardware interface built into the device by using the MISO, MOSI and SCLK pins by using the MAKE\_SPI macro described later.

Sometimes the hardware SPI bus may not be suitable or available and so, for alternatives, then look at *"spisw.h"* (see page 114) and *"spiUart.h"* (see page 115) for other techniques. Note that a software SPI bus will always be the slowest solution at runtime.

When you define slave devices connected to the bus then the format will remain the same and so whichever technique you use for defining the SPI bus, via their respective MAKE commands, then the rest of the code remains the same.

Also note that if you are using the hardware SPI interface then this may also be used by an SPI 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 the chip select (CS) line to the +ve power supply.

Okay - so lets assume you have chosen an appropriate method to create the SPI bus. So the next thing we need to do is create the slave devices that are attached to the bus so that we can talk to them.

Some slave devices, like sdCards and eeproms, are supported directly by WebbotLib and have their own MAKE commands. However: you may have an unsupported device. In which case you can create a default driver for it using MAKE\_SPI\_GENERIC\_DEVICE (see later).

Repeat the above step to create all of the devices controlled by SPI.

Now you need to group them all together into a list using an SPI\_DEVICE\_LIST.

Part of the reason for this list is to make sure that only one device on the same bus is 'active' at any one time. This stops things misbehaving or going 'pop'.

### **Example**

So lets assume that we want to create a device that isn't directly supported by WebbotLib and that we use pin B1 to select the chip. We could do it as follows:-



```
#include "spi.h"
SPI_GENERIC_DEVICE device = MAKE_SPI_GENERIC_DEVICE( SPI_MODE_0,
SPI_DATA_ORDER_MSB, B1, 0xFF);
// Now wrap all the devices into a list
SPI_DEVICE_LIST devices[] = { spiGetAbstractDevice(&device) };
SPI bus = MAKE_SPI(devices);
```

Then in your initialisation code you need to initialise the interface using spiBusInit.

When using the library to talk to a specific device then it will make sure that all other devices are disabled and the target device is enabled.

Since the hardware pins are dictated by your hardware then you may need to check your datasheets to see which pins provide the hardware bus - MISO, MOSI, and SCK.

Here are some examples:

### **Baby Orangutan168:**



### **Axon and Axon II:**

The bus is available on B3, B2 and B1 respectively. Unfortunately these pins have not been bought out to header pins other than the ISP programmer 3x2 pins. So you will need to use the pins from that connector. This will mean constantly swapping your cable with the hardware programmer cable. Life can be made easier by using the boot loader to program the device over the USB/UART1 connection which frees the ISP connector for your own use. Alternatively use software SPI on the I/O pins which suit you.



## Generic 28 pin ATmega8 or ATmega168 board

* 1	28	*
* 2	27	*
* 3	26	*
* 4	25	*
* 5	24	*
* 6	23	*
* 7	22	*
* 8	21	*
* 9	20	*
* 10	19	* SCK/PB5
* 11	18	* MISO/PB4
* 12	17	* MOSI/PB3
* 13	16	*
* 14	15	*

### Standard Function Summary

	<a href="#"><u>MAKE SPI(SPI_DEVICE LIST* devices)</u></a> Create a hardware SPI bus for the specified devices.
	<a href="#"><u>MAKE SPI GENERIC_DEVICE(SPI_MODE mode, SPI_DATA_ORDER order, const IOPin* chipSelect, uint8_t fillerByte)</u></a> Creates a generic SPI slave device.
	<a href="#"><u>spiBusInit(SPI_ABSTRACT_BUS* spi, boolean master)</u></a> Initialise an SPI interface as a master or as a slave.
void	<a href="#"><u>spiDeviceSelect(SPI_DEVICE* device, boolean active)</u></a> Select, or de-select, the device you are communicating with.
void	<a href="#"><u>spiDeviceSendByte(const SPI_ABSTRACT_DEVICE* device, uint8_t data)</u></a> Sends a single byte.
void	<a href="#"><u>spiDeviceSendWord(const SPI_ABSTRACT_DEVICE* device, uint16_t data)</u></a> Sends a 16 bit (2 byte) number.
uint8_t	<a href="#"><u>spiDeviceXferByte(const SPI_ABSTRACT_DEVICE* device, uint8_t data)</u></a> Sends a byte over the SPI interface and returns a response.
uint16_t	<a href="#"><u>spiDeviceXferWord(const SPI_ABSTRACT_DEVICE* device, uint16_t data)</u></a> Sends a 16 bit value and receives a 16 bit word from the SPI interface.



## Standard Function Summary

<code>uint8_t</code>	<a href="#"><code>spiDeviceReceiveByte(const SPI_ABSTRACT_DEVICE* device)</code></a> Receives a byte from the SPI interface.
<code>uint16_t</code>	<a href="#"><code>spiDeviceReceiveWord(const SPI_ABSTRACT_DEVICE* device)</code></a> Receives a 16 bit word from the SPI interface.
<code>void</code>	<a href="#"><code>spiDeviceReceiveBytes(const SPI_ABSTRACT_DEVICE* device, void* buff, size_t size)</code></a> Receives a series of bytes from the SPI interface.
<code>void</code>	<a href="#"><code>spiDeviceSendBytes(const SPI_ABSTRACT_DEVICE* device, const void* buff, size_t size)</code></a> Sends a series of bytes to the SPI interface.

## Advanced Function Summary

	<a href="#"><code>spiBusOff(SPI_ABSTRACT_BUS* bus)</code></a> Turns off the SPI interface.
	<a href="#"><code>spiBusSetMode(SPI_ABSTRACT_BUS* bus, SPI_MODE mode)</code></a> Set the operating mode of the SPI bus.
<code>SPI_MODE</code>	<a href="#"><code>spiBusGetMode(const SPI_ABSTRACT_BUS* bus)</code></a> Returns the current operating mode of the SPI bus.
	<a href="#"><code>spiBusSetDataOrder(SPI_ABSTRACT_BUS* bus, SPI_DATA_ORDER order)</code></a> Sets the bit order that will be used to send bytes over the bus.
<code>SPI_DATA_ORDER</code>	<a href="#"><code>spiBusGetDataOrder(const SPI_ABSTRACT_BUS* bus)</code></a> Returns the current bit ordering of the bus.
	<a href="#"><code>spiBusSetClock(SPI_ABSTRACT_BUS* bus, SPI_CLOCK clock)</code></a> Set the clock speed of the bus.
<code>SPI_CLOCK</code>	<a href="#"><code>spiBusGetClock(SPI_ABSTRACT_BUS* bus)</code></a> Returns the current clock speed of the bus.

## Standard Function Detail

### MAKE\_SPI

`MAKE_SPI(SPI_DEVICE_LIST* devices)`

Create a hardware SPI bus for the specified devices.



## **MAKE\_SPI\_GENERIC\_DEVICE**

```
MAKE_SPI_GENERIC_DEVICE(SPI_MODE mode, SPI_DATA_ORDER order, const  
IOPin* chipSelect, uint8_t fillerByte)
```

Creates a generic SPI slave device.

If WebbotLib doesn't add direct support for your SPI device then use this call to create it. You will then be able to send / receive data to / from it.

The parameters allow you to select the communication protocol and you will need to check the data sheet for your device to find the best settings.

The chipSelect parameter allows you to specify the output pin used to select the device. This is normally connected to the Chip Select (CS) pin.

The 'fillerByte' parameter specifies the data byte that is sent - when only a response is required. The most common value for this parameter is 0xFF.

---

## **spiBusInit**

```
spiBusInit(SPI_ABSTRACT_BUS* spi, boolean master)
```

Initialise an SPI interface as a master or as a slave.

Set the 'master' parameter to TRUE for a master or 'FALSE' for a slave.

---

## **spiDeviceSelect**

```
void spiDeviceSelect(SPI_DEVICE* device, boolean active)
```

Select, or de-select, the device you are communicating with.

The first parameter is the address of the device and the second parameter should be TRUE to select it, or FALSE to de-select it. Note that if you select a device using this call then it will automatically make sure that any other devices are de-selected first - thus ensuring that only one device can be active at a time.

All of the calls that send, receive or transfer data will automatically call this function to select the device.

---

## **spiDeviceSendByte**

```
void spiDeviceSendByte(const SPI_ABSTRACT_DEVICE* device, uint8_t data)
```

Sends a single byte.

This will send a byte and ignore the response.



### **spiDeviceSendWord**

```
void spiDeviceSendWord(const SPI_ABSTRACT_DEVICE* device, uint16_t data)
```

Sends a 16 bit (2 byte) number.

Note that this will send the most significant byte first followed by the least significant byte.

---

### **spiDeviceXferByte**

```
uint8_t spiDeviceXferByte(const SPI_ABSTRACT_DEVICE* device, uint8_t data)
```

Sends a byte over the SPI interface and returns a response.

This will send the data byte and return any response.

---

### **spiDeviceXferWord**

```
uint16_t spiDeviceXferWord(const SPI_ABSTRACT_DEVICE* device, uint16_t data)
```

Sends a 16 bit value and receives a 16 bit word from the SPI interface.

The values are assumed to be high byte then low byte.

---

### **spiDeviceReceiveByte**

```
uint8_t spiDeviceReceiveByte(const SPI_ABSTRACT_DEVICE* device)
```

Receives a byte from the SPI interface.

This will send the 'fillerByte' and return any response.

---

### **spiDeviceReceiveWord**

```
uint16_t spiDeviceReceiveWord(const SPI_ABSTRACT_DEVICE* device)
```

Receives a 16 bit word from the SPI interface.

This will send the 'fillerByte' twice. The returned value is assumed to be high byte then low byte.

---

### **spiDeviceReceiveBytes**

```
void spiDeviceReceiveBytes(const SPI_ABSTRACT_DEVICE* device, void* buff, size_t size)
```

Receives a series of bytes from the SPI interface.

Read the specified number of bytes into a buffer.

---



## **spiDeviceSendBytes**

```
void spiDeviceSendBytes(const SPI_ABSTRACT_DEVICE* device, const void* buff, size_t size)
```

Sends a series of bytes to the SPI interface.

Writes the specified number of bytes from a buffer.

---

### **Advanced Function Detail**

---

## **spiBusOff**

```
spiBusOff(SPI_ABSTRACT_BUS* bus)
```

Turns off the SPI interface.

Turning off the SPI interface may be done to save power. To turn it back on again later then you must issue another call to spilnit.

---

## **spiBusSetMode**

```
spiBusSetMode(SPI_ABSTRACT_BUS* bus, SPI_MODE mode)
```

Set the operating mode of the SPI bus.

The mode parameter must be one of the following values:

```
SPI_MODE_0
SPI_MODE_1
SPI_MODE_2
SPI_MODE_3
```

NB When a slave device is activated then the mode of the bus will automatically be changed to the correct mode for the device. This means that the bus can contain slave devices that require different operating modes and the library automatically changes to the correct mode for the current device.

---

## **spiBusGetMode**

```
SPI_MODE spiBusGetMode(const SPI_ABSTRACT_BUS* bus)
```

Returns the current operating mode of the SPI bus.

The returned value will be one of the following values:

```
SPI_MODE_0
SPI_MODE_1
SPI_MODE_2
SPI_MODE_3
```



## **spiBusSetDataOrder**

`spiBusSetDataOrder(SPI_ABSTRACT_BUS* bus, SPI_DATA_ORDER order)`

Sets the bit order that will be used to send bytes over the bus.

The 'order' parameter must be one of the following values:-

```
SPI_DATA_ORDER_MSB  
SPI_DATA_ORDER_LSB
```

NB When a slave device is activated then the ordering of the bus will automatically be changed to the correct ordering for the device. This means that the bus can contain slave devices that require different bit ordering and the library automatically changes to the correct ordering for the current device.

---

## **spiBusGetDataOrder**

`SPI_DATA_ORDER spiBusGetDataOrder(const SPI_ABSTRACT_BUS* bus)`

Returns the current bit ordering of the bus.

The returned value will be one of the following values:-

```
SPI_DATA_ORDER_MSB  
SPI_DATA_ORDER_LSB
```

## **spiBusSetClock**

`spiBusSetClock(SPI_ABSTRACT_BUS* bus, SPI_CLOCK clock)`

Set the clock speed of the bus.

The 'clock' parameter must be one of the following values:

```
SPI_CLOCK_DIV2  
SPI_CLOCK_DIV4  
SPI_CLOCK_DIV8  
SPI_CLOCK_DIV16  
SPI_CLOCK_DIV32  
SPI_CLOCK_DIV64  
SPI_CLOCK_DIV128
```

The clock speed can be changed at any time but the preferred method is to set the required speed before initialising the bus with `spiBusInit`. Note that the same clock speed is used to talk to all of the slave devices - so you should set it to the speed required by the slowest device on the bus.

---

## **spiBusGetClock**

`SPI_CLOCK spiBusGetClock(SPI_ABSTRACT_BUS* bus)`

Returns the current clock speed of the bus.

---





The returned value will be one of the following values:

```
SPI_CLOCK_DIV2  
SPI_CLOCK_DIV4  
SPI_CLOCK_DIV8  
SPI_CLOCK_DIV16  
SPI_CLOCK_DIV32  
SPI_CLOCK_DIV64  
SPI_CLOCK_DIV128
```



## **spisw.h**

Allows you to simulate an SPI interface in software.

This is useful when your hardware doesn't provide hardware SPI or if you want additional SPI interfaces.

Note that this currently only supports master mode.

You can use any IO pins for the SPI bus (MOSI, MISO and SCLK).

Assuming we want to use the following pins: B1 for MOSI, B2 for MISO and B3 for SCLK then you can create the interface via:-

```
#include "spisw.h"
SPI_SW spi = MAKE_SW_SPI(device_list, B1, B2, B3);
```

Once defined you can use the same functions as described in "*spi.h*" (see page 105)



## **spiUart.h**

Allows you to simulate an SPI interface using a hardware UART that provides this ability.

This is useful when your hardware doesn't provide hardware SPI or if you want additional SPI interfaces.

Note that the hardware only supports master mode.

The ATmega8 and ATmega32 DO NOT provide this ability whereas the ATmega168, ATmega328P, ATmega640, ATmega2560 and ATmega2561 DO.

The transmit output of the UART corresponds to MOSI pin, the receive input of the UART corresponds to the MISO pin. The SCLK clock output comes from the XCK pin of the relevant UART.

To define UART2 to be used as an SPI master then:-

```
#include "spiUart.h"
SPI_UART spi = MAKE_UART_SPI(device_list, UART2);
```

Checking the datasheet for the ATmega640, or look at 'dev/ATmega640.h' in this library, then you will see that the clock pin for UART2 (XCK2) is on H2.

Once defined you can use the same functions as described in spi.h.



## switch.h

This module allows you to wrap an IOPin to represent a switch or push button.

The button should be connected between an IOPin and Ground.

Assuming that the switch is connected to B1 then:

```
SWITCH mySwitch = MAKE_SWITCH(B1); // create the switch
// In appInitHardware...
SWITCH_init(&mySwitch); // Initialise the switch
// In your main loop
The switch can then be read using the pressed/released functions.
```

### Standard Function Summary

void	<a href="#">SWITCH_init(SWITCH *sw)</a> Initialise a switch.
boolean	<a href="#">SWITCH_pressed(SWITCH* sw)</a> Test if a given switch has been pressed.
boolean	<a href="#">SWITCH_released(SWITCH* sw)</a> Test if a switch is 'released' ie 'not pressed'.

### Standard Function Detail

#### SWITCH\_init

```
void SWITCH_init(SWITCH *sw)
    Initialise a switch.
```

This is normally called from `appInitHardware` to make sure the given pin is an input pin and that the internal pullup resistor is enabled.

#### SWITCH\_pressed

```
boolean SWITCH_pressed(SWITCH* sw)
    Test if a given switch has been pressed.
```

#### SWITCH\_released

```
boolean SWITCH_released(SWITCH* sw)
    Test if a switch is 'released' ie 'not pressed'.
```



## timer.h

Defines Helper functions for the timers and compare channels.

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-mistify some of the above by providing a more intelligent API than other libraries.

Since your program must include a `sys/*.h` file then 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.

Each timer has a name such as `TIMER0`, `TIMER1`, `TIMER2` etc. Unlike other libraries we discourage you from reading values from any given timer in order to measure durations since the result is dependent on the clock speed and any prescaler. Instead we introduce the concept of an internal clock. This clock is not a 'time of day' clock in that it doesn't generate values that are guaranteed to increase. It will, after every 7 minutes, wrap around to zero but since I have never found a scenario where you need to measure incoming pulses any way near this value then it is not a problem. The clock is only defined after your 'app\_init' code completes - at which time the library attempts to create the clock from any timers that are unused by your code. As a consequence none of the 'clock' functions will work during 'app\_init' but the various 'delay' functions will.

I don't understand! What does that mean? Well in essence it means you should **never ever** do the following:-

```
uint32_t delay = timer0_get_overflow * 256 + TCNT0;
```

mainly because the result is linked to the CPU clock speed but is also 'full of bugs' caused by interrupts. Instead you should use the 'clock' functions in this library. So to measure how long something takes then:-

```
TICK_COUNT start = clockGetus();  
...do something...  
TICK_COUNT end = clockGetus();  
TICK_COUNT duration = end - start;
```

Various general delay routines are also defined to allow you to create delays in cycles (not recommended), microseconds and milliseconds. Generally all delays are not a good idea as the processor just idles and the whole system just grinds to a halt - it will still work but you should consider rewriting your code so that other things continue to happen.



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

TICK_COUNT	<a href="#"><u>clockGetus()</u></a> Get the current time in $\mu$ S.
boolean	<a href="#"><u>clockHasElapsed(TICK_COUNT usStart, TICK_COUNT usWait)</u></a> Test if a given duration has elapsed.
	<a href="#"><u>clockWaitms(TICK_COUNT ms)</u></a> Pause for the given number of milliseconds.
	<a href="#"><u>clockWaitus(TICK_COUNT us)</u></a> Pause for the given number of microseconds.
	<a href="#"><u>delay_cycles(uint32 t_cycles)</u></a> Pause for the given number of clock cycles.
	<a href="#"><u>delay_ms(uint32 t_ms)</u></a> Pause for the given number of ms.
	<a href="#"><u>delay_us(uint32 t_us)</u></a> Pause for the given number of $\mu$ s.
uint64_t	<a href="#"><u>MS TO CYCLES(ms)</u></a> Convert milliseconds into a number of cycles.
uint64_t	<a href="#"><u>US TO CYCLES(us)</u></a> Convert microseconds into a number of cycles.
	<a href="#"><u>timerOverflowAttach(const Timer* timer, TimerCallback callback, void* user_data)</u></a> Register an overflow callback.
	<a href="#"><u>timerOverflowDetach(const Timer* timer)</u></a> Remove any overflow callback for the given timer.
	<a href="#"><u>timerCaptureAttach(const Timer* timer, TimerCallback callback, void* user_data, boolean risingEdge)</u></a> Register a capture callback.
	<a href="#"><u>timerCaptureDetach(const Timer* timer)</u></a> Remove any capture callback for the given timer.



## Standard Function Summary

boolean	<a href="#"><u>timerSupportsCompare(const Timer* timer)</u></a> Does the timer have any compare units? Return TRUE if the timer has at least one compare unit.
uint8_t	<a href="#"><u>timerNumberOfCompareUnits(const Timer* timer)</u></a> Returns the number of compare units provided by this timer.
uint16_t	<a href="#"><u>timerGetPrescaler(const Timer* timer)</u></a> Returns the current clock prescaler value for this timer.
boolean	<a href="#"><u>timerIsInUse(const Timer* timer)</u></a> Test if a given timer is currently in use.
boolean	<a href="#"><u>timerIs16bit(const Timer* timer)</u></a> 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.
	<a href="#"><u>compareAttach(const TimerCompare* channel, TimerCompareCallback callback, uint16_t threshold, void* data)</u></a> Attach a callback function to a compare unit.
	<a href="#"><u>compareDetach(const TimerCompare* compare)</u></a> Removes any callback function from the given compare unit.
const TimerCompare*	<a href="#"><u>compareFromIOPin(const IOPin* pin)</u></a> Find the compare unit that can be used to toggle the specified IOPin.
const IOPin*	<a href="#"><u>compareGetPin(const TimerCompare* channel)</u></a> 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	<a href="#"><u>compareGetThreshold(const TimerCompare* channel)</u></a> Returns the threshold value for the given compare unit.
	<a href="#"><u>compareSetThreshold(const TimerCompare* channel, uint16_t threshold)</u></a> Changes the threshold value for the compare unit.
const Timer*	<a href="#"><u>compareGetTimer(const TimerCompare* compare)</u></a> Returns the Timer that this compare unit is associated with.
boolean	<a href="#"><u>compareIsInUse(const TimerCompare* channel)</u></a> Tests if a given compare unit is currently in use.
CHANNEL_MODE	<a href="#"><u>compareGetOutputMode(const TimerCompare* channel)</u></a> Find the mode used for toggling the PWM output pin of a



## Standard Function Summary

	<a href="#">compare channel when the compare threshold is met.</a>
	<a href="#">compareSetOutputMode(const TimerCompare* channel, CHANNEL_MODE mode)</a> Sets the output mode of the compare unit.
uint8_t	<a href="#">NUMBER OF TIMERS</a> A constant value representing the number of Timers in the microprocessor.

## Advanced Function Summary

boolean	<a href="#">clockHasElapsedGetOverflow(TICK_COUNT usStart, TICK_COUNT usWait, TICK_COUNT* overflow)</a> Similar to 'clockHasElapsed' but returns the number of $\mu$ S left to go if the duration has not elapsed, or the number of $\mu$ S we have exceeded the duration if it has elapsed.
uint32_t	<a href="#">ticks_per_ms(uint32_t ms, uint16_t prescale)</a> Returns the number of timer ticks that would be generated for the given number of milliseconds if the timer used the given prescaler.
	<a href="#">timerOverflowClearInterruptPending(const Timer* timer)</a> Clears the timers 'overflow interrupt pending' flag.
boolean	<a href="#">timerOverflowIsInterruptPending(const Timer* timer)</a> Test if an overflow interrupt is pending for the given timer.
boolean	<a href="#">timerIsCaptureSupported(const Timer* timer)</a> Returns TRUE if this timer supports Input Capture.
	<a href="#">timerCaptureClearInterruptPending(const Timer* timer)</a> Clears the timers 'capture interrupt pending' flag.
boolean	<a href="#">timerCaptureIsInterruptPending(const Timer* timer)</a> Test if a capture interrupt is pending for the given timer.
const IOPin*	<a href="#">timerGetCapturePin(const Timer* timer)</a> Returns the IOPin used for input capture.
const TimerCompare*	<a href="#">timerGetCompare(const Timer* timer, uint8_t channel)</a> Find the compare unit for a given timer.
TIMER_MODE	<a href="#">timerGetMode(const Timer* timer)</a> Returns the current mode for a timer.





## Advanced Function Summary

	<a href="#"><code>timerSetMode(const Timer* timer, TIMER_MODE mode)</code></a> Set the operating mode for the given timer.
boolean	<a href="#"><code>timerIsModeSupported(const Timer* timer, TIMER_MODE mode)</code></a> Test if a timer supports a given mode of operation.
uint16_t	<a href="#"><code>timerGetBestPrescaler(const Timer* timer, uint16_t repeat_ms)</code></a> Find the optimum prescaler value for the given timer in order to measure the given number of ms.
boolean	<a href="#"><code>timerSupportsPrescaler(const Timer* timer, uint16_t prescaler)</code></a> Does the timer support the given prescaler value?
uint16_t	<a href="#"><code>timerGetCounter(const Timer* timer)</code></a> Returns the current counter value for the timer.
	<a href="#"><code>timerSetPrescaler(const Timer* timer, uint16_t prescaler)</code></a> Set the clock prescaler value for the given timer.
uint16_t	<a href="#"><code>timerGetTOP(const Timer* timer)</code></a> Returns the current value of TOP for the given timer.
	<a href="#"><code>timerOff(const Timer* timer)</code></a> Turn off the specified timer.
	<a href="#"><code>compareClearInterruptPending(const TimerCompare* channel)</code></a> Clears the 'interrupt pending' flag for the compare unit.
boolean	<a href="#"><code>compareIsInterruptPending(const TimerCompare* channel)</code></a> Test if the compare unit has set the 'interrupt pending' flag.

## Standard Function Detail

### clockGetus

`TICK_COUNT clockGetus()`

Get the current time in  $\mu$ S.

Note that this number will wrap around so a later reading may give a smaller value.

This happens every 0xfffffff or 429,4967,295 microseconds ie every 429.5 seconds or every 7 minutes.

This means that the longest time difference you can sense by subtracting two values is about 7 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 7 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 7
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.

---

## **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  $\mu$ s.

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`

---

## **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 change 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 change 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 `compareDetach`.

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

### clockHasElapsedGetOverflow

`boolean clockHasElapsedGetOverflow(TICK_COUNT usStart, TICK_COUNT usWait, TICK_COUNT* overflow)`

Similar to 'clockHasElapsed' but returns the number of  $\mu$ S left to go if the duration has not elapsed, or the number of  $\mu$ S we have exceeded the duration if it 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'  $\mu$ S left  
}  
// overflow has the number of  $\mu$ S 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.

---

## ticks\_per\_ms

`uint32_t ticks_per_ms(uint32_t ms, uint16_t prescale)`

Returns the number of timer ticks that would be generated for the given number of milliseconds if the timer used the given prescaler.

---





This is normally used in conjunction with `timerGetBestPrescaler(const Timer* timer, uint16_t repeat_ms)`. You would use `timerGetBestPrescaler(const Timer* timer, uint16_t repeat_ms)` to find the optimum prescaler value for the timer and then use this value in `ticks_per_ms(uint32_t ms, uint16_t prescale)` to find the number of ticks that would be generated for the given number of ms.

---

### **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)
```

Turn 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.



## tone.h

Play a tone 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  $8\Omega$  speaker then the current will be  $5/8 = 625\text{mA}$  which is way in excess of the typical  $40\text{mA}$  that a typical pin can provide.

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

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

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.

To create and use a single tone player using timer 2 to pin B0 then:

```
#include <tone.h>
TONE_PLAYER tone1 = MAKE_TONE_PLAYER(TIMER2, B0);
```

Then initialise the player in apInitHardware:

```
toneInit( &tone1 );
```

You can now use the remaining calls to play/stop tones.

### Standard Function Summary

	<a href="#"><u>MAKE_TONE_PLAYER(const Timer* timer, const IOPin* pin)</u></a> Creates a new tone player.
--	---



## Standard Function Summary

	<a href="#"><code>toneInit(TONE_PLAYER* player)</code></a> Initialise the tone player.
	<a href="#"><code>tonePlay(TONE_PLAYER* player, uint16 t frequency, uint32 t durationMS )</code></a> Start playing the specified tone in the background.
	<a href="#"><code>toneStop(TONE_PLAYER* player)</code></a> Stop any current output.
boolean	<a href="#"><code>toneIsPlaying(const TONE_PLAYER* player)</code></a> Returns TRUE if a note is currently being played or FALSE if not.
	<a href="#"><code>tonePlayRTTTL(TONE_PLAYER* player, const prog_char tune[])</code></a> Play an entire tune stored in RTTTL format.

## Advanced Function Summary

	<a href="#"><code>toneAttach(TONE_PLAYER* player, ToneCallback callback)</code></a> Attach a callback routine to be called when the current note has finished playing.
	<a href="#"><code>toneDetach(TONE_PLAYER* player)</code></a> Remove any callback registered with this tone player.

## Standard Function Detail

### MAKE\_TONE\_PLAYER

`MAKE_TONE_PLAYER(const Timer* timer, const IOPin* pin)`

Creates a new tone player.

The first parameter specifies the timer which will be exclusively used by the tone generator.

The second parameter can be any output pin and is the pin where the frequency signal will be output.

### toneInit

`toneInit(TONE_PLAYER* player)`

Initialise the tone player.

This will configure the timer ready for use and make sure that the same timer is not being used for other things. It should be called from within your `aplnitHardware`.



## tonePlay

`tonePlay(TONE_PLAYER* player, uint16_t frequency, uint32_t durationMS )`

Start playing the specified tone in the background.

The first parameter specifies the TONE\_PLAYER you want to use.

The second parameter is the frequency of the tone. See `tone.h` for some shortcuts for musical notes. You can also add a pause by specifying a frequency of 0.

The final parameter is how long you want the tone to be played in milliseconds. A value of 0 will cause the note to be played continuously until either a new note is played or you call `toneStop`.

---

## toneStop

`toneStop(TONE_PLAYER* player)`

Stop any current output.

---

## toneIsPlaying

`boolean toneIsPlaying(const TONE_PLAYER* player)`

Returns TRUE if a note is currently being played or FALSE if not.

---

## tonePlayRTTTL

`tonePlayRTTTL(TONE_PLAYER* player, const prog_char tune[])`

Play an entire tune stored in RTTTL format.

The RTTTL format (Ring Tone Text Transfer Language) is a piece of text to represent a simple tune such as those used by hand held phones. See [http://en.wikipedia.org/wiki/Ring\\_Tone\\_Transfer\\_Language](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.

This command allows you to play such a ring tone whilst the rest of your robot code is running.

Here is an example for the Axon:-

```
#include <tone.h>
// Create the tone player using TIMER1 and outputting to A2
TONE_PLAYER tone = MAKE_TONE_PLAYER(TIMER1,A2);
```

Now initialise the tone library from `applInitHardware`:-

---



```
toneInit(&tone);
```

Now we can play a tune. But before we do so then we need to create the RTTTL data. The first thing to note in the following example is the use of the 'PROGMEM' keyword to make sure that we don't waste precious RAM memory. So here is an example that you can add at the top of your file:-

```
const char PROGMEM tune[] =  
"StarWars:d=4,o=5,b=45:32p,32f#,32f#,32f#,8b.,8f#.6,32e6,32d#6,32c#6,8b.6,16f#.6,32e6,3  
2d#6,32c#6,8b.6,16f#.6,32e6,32d#6,32e6,8c#.6,32f#,32f#,32f#,8b.,8f#.6,32e6,32d#6,32c#6,  
8b.6,16f#.6,32e6,32d#6,32c#6,8b.6,16f#.6,32e6,32d#6,32e6,8c#6";
```

To play the tune when the robot first starts up then we can add the following to `aplnitSoftware`:-

```
tonePlayRTTTL(&tone, tune);
```

## Advanced Function Detail

### toneAttach

```
toneAttach(TONE_PLAYER* player, ToneCallback callback)
```

Attach a callback routine to be called when the current note has finished playing.

Since the musical tones are played in the background then its useful to be able to add some code that is called automatically when the current note has finished so that you can queue up the next note. That's what this function is for!

The first parameter is the tone player you want to add the callback to and the second parameter is the address of your own routine.

Your routine should be defined with the following return and parameter signature:-

```
void myCallback(TONE_PLAYER* player){  
    ... the previous tone has finished ...  
    ... now do what you want to do - including tonePlay to start the next  
    note ...  
}
```

---

### toneDetach

```
toneDetach(TONE_PLAYER* player)
```

Remove any callback registered with this tone player.





## uart.h

Routines for handling serial communications with other devices by using built-in UART hardware.

If you run out of UARTs you can use `uartsw.h` 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 UARTs are not automatically initialised at start up and so you must initialise them by hand. The serial motor drivers will do this for you but when using `rprintf` debug commands over a UART then you must use `uartInit(UART* uart, BAUD_RATE baud)` in your initialisation code to set up the baud rate. Older versions of the library used to do this for you but has been removed mainly because it could cause random characters to be output to the UART due to power switch contact bounce.

By default the UARTs have no transmit and receive buffers. This means that when you try to transmit a byte then your program will pause until any previous byte has been sent. Received characters will be thrown away unless you use `uartAttach` to attach a callback routine that is called when the character is received.

Alternatively you can supply transmit and receive buffers by specifying `UART_RX_BUFFER_SIZE` and/or `UART_TX_BUFFER_SIZE` at the top of your program file (before you include the system file). If you specify a transmit buffer size then all transmissions will occur in the background and your program will not pause unless the transmit buffer fills up because you are adding characters faster than they can be sent. If you specify a receive buffer then all received data is queued up until you read it. If you fail to read the received data frequently enough then the receive buffer may overflow and characters will be lost.

To set up the UARTs with a transmit buffer of 80 bytes and a receive buffer of 20 bytes then you would add the following lines to the very top of your program:-

```
#define UART_TX_BUFFER_SIZE 80
#define UART_RX_BUFFER_SIZE 20
```

Most of the functions in this module require a reference to the UART you want to perform the action upon. Each uart has a name such as `UART0`, `UART1` etc. Alternatively, you can iterate through all of the available hardware uarts as follows:-



```
int i;
for(i=0;i<NUM_UARTS;i++){
  HW_UART* theUart = &Uarts[i];
  // You can now pass in 'theUart' as the reference
}
```

## Standard Function Summary

int	<a href="#">uartGetByte(UART* uart)</a> Returns the next byte from the receive buffer or -1 if there is no buffer or the buffer is empty.
uint8_t	<a href="#">uartSendByte(UART* uart, uint8_t txData)</a> Transmit a byte on the specified UART.
	<a href="#">uartSetBaudRate(UART* uart, BAUD_RATE baudrate)</a> Set the baud rate for the UART.
	<a href="#">uartFlushReceiveBuffer(UART* uart)</a> Discards any data in the receive buffer.
boolean	<a href="#">uartHasOverflowed(const UART* uart)</a> Test if the receive buffer has overflowed.
boolean	<a href="#">uartReceiveBufferIsEmpty(const UART* uart)</a> Tests if the receive buffer is empty.
boolean	<a href="#">uartIsBusy(UART* uart)</a> Returns FALSE if the UART has nothing left to send.
void	<a href="#">uartSendBuffer(UART* uart, const uint8_t* data, size_t count)</a> Send a block of bytes to the UART.
Reader	<a href="#">uartGetReader(const UART*)</a> Returns the Reader associated with this UART.
Writer	<a href="#">uartGetWriter(const UART*)</a> Returns the Writer for a given UART.

## Advanced Function Summary

	<a href="#">uartAttach(UART* uart, void (*rx_func)(unsigned char c))</a> Attaches a callback routine that is invoked when a character is received and the UART has no receive buffer.
	<a href="#">uartDetach(UART* uart)</a> Removes any existing receive callback from the UART.



## Advanced Function Summary

cBuffer*	<a href="#"><u>uartGetRxBuffer(const UART* uart)</u></a> Returns the receive buffer or null if there is no buffer.
cBuffer*	<a href="#"><u>uartGetTxBuffer(const UART* uart)</u></a> Returns the UARTs transmit buffer or null if there is none.
	<a href="#"><u>uartSetReceiveBuffer(UART* uart, cBuffer* buffer)</u></a> Changes the receive buffer.
	<a href="#"><u>uartSetTransmitBuffer(UART* uart, cBuffer * buffer)</u></a> Changes the transmit buffer.
	<a href="#"><u>uartInit(UART* uart, BAUD_RATE baud)</u></a> Re-initialise a UART after it has been turned off.
	<a href="#"><u>uartOff(UART* uart)</u></a> Disables the given UART once all pending transmissions have finished.
void	<a href="#"><u>uartReceivePollingMode(const HW_UART* uart,boolean polling)</u></a> Changes a hardware UART receive mode between interrupt driven and polling modes.
int	<a href="#"><u>uartPollByte(const HW_UART* uart)</u></a> Poll the hardware UART to see if a byte has been received.

## Standard Function Detail

### uartGetByte

```
int uartGetByte(UART* uart)
```

Returns the next byte from the receive buffer or -1 if there is no buffer or the buffer is empty.

Example:

```
int ch = uartGetByte(UART0);
if(ch!=-1){
  // We have got a character in 'ch'
}
```

There are shorthand forms when dealing with a specific uart. eg

uart0GetByte() is the same thing as uartGetByte(UART0) and

uart1GetByte() is the same thing as uartGetByte(UART1)



The shorthand forms do exactly the same thing but because they don't need to pass a parameter then they result in smaller code. However: they make it harder if you suddenly decide to change your code to receive via a different UART as you will have to search and replace all the shorthand forms. The longer form allows you to use a #define at the start of your program such as:

```
#define LISTEN UART0
```

then your code can pass 'LISTEN' to all of the UART commands. Changing to another uart just involves changing the 'LISTEN' line to use another uart.

---

## uartSendByte

```
uint8_t uartSendByte(UART* uart, uint8_t txData)
```

Transmit a byte on the specified UART.

If the UART has a transmit buffer then the byte will be added to the buffer and sent in the background. If the buffer is full then it will pause until there is available space in the buffer.

If the UART does not have a buffer then it will pause until any previous byte has been sent and then it will transmit this byte.

For example:

```
uartSendByte(UART0, 'A'); // send an A character
```

There are shorthand forms when dealing with a specific uart. eg

uart0SendByte(data) is the same thing as uartSendByte(UART0,data) and

uart1SendByte(data) is the same thing as uartSendByte(UART1,data) etc

The shorthand forms do exactly the same thing but because they don't need to pass a parameter then they result in smaller code. However: they make it harder if you suddenly decide to change your code to transmit via a different UART as you will have to search and replace all the shorthand forms. The longer form allows you to use a #define at the start of your program such as:

```
#define SEND UART0
```

then your code can pass 'SEND' to all of the UART commands. Changing to another uart just involves changing the 'SEND' line to use another uart.

The return value is the byte you have sent and this makes it easier to do things like build checksums for the bytes that are sent.

---



## uartSetBaudRate

`uartSetBaudRate(UART* uart, BAUD_RATE baudrate)`

Set the baud rate for the UART.

For example:

```
uartSetBaudRate(UART0, (BAUD_RATE)19200);
```

There is a 'magic' value for the baud rate called `BAUD_RATE_MAX` which will use the highest possible baud rate. This is equivalent to setting the baud rate divisors to 1. The actual baud rate achieved will depend on your clock speed.

---

## uartFlushReceiveBuffer

`uartFlushReceiveBuffer(UART* uart)`

Discards any data in the receive buffer.

This is useful if you have changed the baud rate from some earlier value and so any characters in the receive buffer may have been received incorrectly:-

```
uartSetBaudRate(UART0, (BAUD_RATE)19200);  
uartFlushReceiveBuffer(UART0);
```

## uartHasOverflowed

`boolean uartHasOverflowed(const UART* uart)`

Test if the receive buffer has overflowed.

Returns TRUE if it has overflowed, ie at least one character has been lost, or FALSE if not.

The receive buffer overflows when it is receiving characters faster than you are processing them. If you are receiving bursts of data then increase the size of the receive buffer. However: if you are receiving a constant stream of data then you either need to speed up your program to cope with the data or you need to reduce the baud rate.

---

## uartReceiveBufferIsEmpty

`boolean uartReceiveBufferIsEmpty(const UART* uart)`

Tests if the receive buffer is empty.

Return TRUE if it is empty (or there is no receive buffer), or FALSE if it contains at least one byte.

```
if(uartReceiveBufferIsEmpty(UART0)){  
    // We have no data to process. So do other things....  
}else{  
    // We have data to process so start reading it in  
}
```



## uartIsBusy

boolean uartIsBusy(UART\* uart)

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! Example:

```
if(!uartIsBusy(UART0)){  
    // UART has nothing to do  
    // so send your stuff out now  
}
```

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.

---

## uartSendBuffer

void uartSendBuffer(UART\* uart, const uint8\_t\* data, size\_t count)

Send a block of bytes to the UART.

Parameter 1 is the UART.

Parameter 2 is the address of the data to be sent.

Parameter 3 is the number of bytes to be sent starting at the address in parameter 2.

Example:-

```
uint8_t data[4];  
data[0] = 0x55;  
data[1] = 0x99;  
data[2] = 'A';  
data [3] = '\n';  
uartSendBuffer(UART1, data, 4);
```

---

## uartGetReader

Reader uartGetReader(const UART\*)

Returns the Reader associated with this UART.

A 'Reader' is a routine that can be called to get the next received character or -1 if there isn't one. The default behaviour, for a hardware UART, is to return the address of 'uartXGetByte' where 'X' is the UART number.

---



This has been done to make it easier to move from one UART to another. Previously you probably used a `#define` to identify the UART you used to connect to your PC. ie

```
#define PC_UART UART1
```

You then probably used calls to `uart1GetByte` to read incoming characters. This made it difficult to change to another UART as you then had to change multiple lines.

So the new way of doing this is to keep your `#define` for the UART and then change your read code to:

```
int ch = uartGetReader(PC_UART)();
```

---

## uartGetWriter

`Writer uartGetWriter(const UART*)`

Returns the Writer for a given UART.

A 'Writer' is a routine that can be called to send a character to the UART. The default behaviour, for a hardware UART, is to return the address of `'uartXSendByte'` where 'X' is the UART number.

This has been done to make it easier to move from one UART to another. Previously you probably used a `#define` to identify the UART you used to connect to your PC. ie

```
#define PC_UART UART1
```

You then probably used calls to `uart1SendByte` to write characters or you may have used `rprintfInit(&uart1SendByte)` to send `rprintf` output to the UART. This made it difficult to change to another UART as you then had to change multiple lines.

So the new way of doing this is to keep your `#define` for the UART and then change your write code to:

```
rprintfInit(uartGetWriter(PC_UART));
```

If you were to change the `#define` to use a different UART then the `rprintfInit` would automatically now use the correct routine to put characters to the new UART.

## Advanced Function Detail

### uartAttach

`uartAttach(UART* uart, void (*rx_func)(unsigned char c))`

Attaches a callback routine that is invoked when a character is received and the UART has no receive buffer.

For example you could have your own routine:



```
void myReceive(unsigned char data){  
    // You have just received a byte in the 'data' parameter  
}
```

You can now ask the UART0 to call it when a byte is received by writing:-

```
uartAttach(UART0, &myReceive);
```

Note the '&' character at the start of your function name.

Only one callback routine can be registered at any one time.

NB Your callback routine will never be called if the UART has a receive buffer.

---

## uartDetach

```
uartDetach(UART* uart)
```

Removes any existing receive callback from the UART.

---

## uartGetRxBuffer

```
cBuffer* uartGetRxBuffer(const UART* uart)
```

Returns the receive buffer or null if there is no buffer.

---

## uartGetTxBuffer

```
cBuffer* uartGetTxBuffer(const UART* uart)
```

Returns the UARTs transmit buffer or null if there is none.

---

## uartSetReceiveBuffer

```
uartSetReceiveBuffer(UART* uart, cBuffer* buffer)
```

Changes the receive buffer.

The only time you may want to use this is if you have multiple UARTs and you want them to have different receive buffer sizes. If this is what you want to do then don't define 'UART\_RX\_BUFFER\_SIZE' at the start of your program so that the UARTs don't have a receive buffer. Then initialise them by hand in your start up code:-

```
unsigned char myRxBuf[128];  
cBuffer myRxBuffer = MAKE_BUFFER(myRxBuf);  
uartSetReceiveBuffer(UART0, &myRxBuffer);
```

---

## uartSetTransmitBuffer

```
uartSetTransmitBuffer(UART* uart, cBuffer * buffer)
```

Changes the transmit buffer.





The only time you may want to use this is if you have multiple UARTs and you want them to have different transmit buffer sizes. If this is what you want to do then don't define 'UART\_TX\_BUFFER\_SIZE' at the start of your program so that the UARTs don't have a transmit buffer. Then initialise them by hand in your start up code:-

```
unsigned char myTxBuf[128];
cBuffer myTxBuffer = MAKE_BUFFER(myTxBuf);
uartSetTransmitBuffer(UART0,&myTxBuffer);
```

---

## uartInit

`uartInit(UART* uart, BAUD_RATE baud)`

Re-initialise a UART after it has been turned off.

---

## uartOff

`uartOff(UART* uart)`

Disables the given UART once all pending transmissions have finished.

---

## uartReceivePollingMode

`void uartReceivePollingMode(const HW_UART* uart,boolean polling)`

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 for 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\_START/END 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.

If you want to use this in your own code then here is a half duplex example for UART0 which expects a 10 byte response:

```
// Put uart into polling mode for receive
uartReceivePollingMode(UART0, 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(uartIsBusy(UART0)){
    breathe();
}
```

```
CRITICAL_SECTION_START; // Turn off hardware interrupts
while(replyPos < sizeof(reply)){
    int ch = uartPollByte(UART0);
    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;
    }
}
CRITICAL_SECTION_END; // Turn interrupts back on
```

---

## uartPollByte

```
int uartPollByte(const HW_UART* uart)
```

Poll the hardware UART to see if a byte has been received.

Returns the byte or -1 if nothing has been received yet.



## uartsw.h

Simulates a UART in software.

Exactly the same functions are available as for a hardware UART so refer to "*uart.h*" (see page 137) for the list of methods. This means we can interchange between hardware and software UARTs with very few code changes - just your initialisation code.

Each software UART requires a 16 bit Timer with two compare channels. If you want the uart to receive incoming data then the IO pin used for 'receive' must be the input capture pin associated with the timer. See the tables in the *sys/\*.h* for your board. If in doubt use timer number 1. The output pin, for the 'transmit wire', can be any available IO pin.

If you want to add buffers to the receive and/or transmit then you must create these buffers yourself. Here's an example for the ATmega168 that uses timer 1 (input capture is always on B0) and transmits on D1 and has an 80 byte buffer on both the receive and transmit:

```
#include "sys/atmega168.h"
#include "uartsw.h"
```

```
// Make receive buffer space
unsigned char rxArr[80];
```

```
// Create the receive buffer
cBuffer rxBuf = MAKE_BUFFER(rxArr);
```

```
// Make transmit buffer space
unsigned char txArr[80];
```

```
// Create the transmit buffer
cBuffer txBuf = MAKE_BUFFER(txArr);
```

```
// Create the software uart
SW_UART swUART = MAKE_SW_UART_BUFFERED( &rxBuf, &txBuf, 1, B0, D1, FALSE, null,
null);
```



```
// Create a 'define' in case we change its name
#define URT &swUART
```

```
// This routine is called once only and allows you to do any initialisation
// Dont use any 'clock' functions here - use 'delay' functions instead
void appInitHardware(void){
    // Initialise the UART to 4800 baud
    uartInit(URT, (BAUD_RATE)4800);
}
```

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
```

```
// This routine is called repeatedly - its your main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Do stuff with the UART
    return 0;
}
```

Note that you can pass 'null' for the receive pin (if your UART has no input) or 'null' for the transmit pin (if your UART has no output).




## sys

Configure WebbotLib by including the correct definition file for your board.

axon.h	150
axon2.h	152
babyOrangutan168.h	155
babyOrangutanB328.h	156
roboduino.h	157
atmega8.h	158
atmega168.h	159
atmega32.h	160
atmega328P.h	161
atmega1280.h	162
atmega2560.h	164
atmega2561.h	166
atmega128.h	167
atmega640.h	168
atmega644.h	170



## sys/axon.h

This is the system file you should include if you are developing for the Axon. 

It will make sure that your compiler settings are appropriate: ie ATmega640 running at 16MHz.

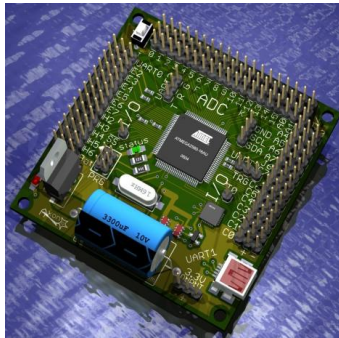
The following functions are available for the additional hardware on this board:-

`void statusLED_on()` - turn on the green status LED surface mounted on the board

`void statusLED_off()` - turn off the green status LED surface mounted on the board

`boolean button_pressed(void)` - has the surface mounted button on the board been pressed?

Note that the green status LED is used by this library for indicating any errors.



The following IOPins are available:

G5 , E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , H0 , H1 , H2 , H3 , H4 , H5 , H6 , B6 , H7 , D0 , D1 , D4 , D5 , D6 , D7 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , J0 , J1 , J2 , J3 , J4 , J5 , J6 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , J7 , K7 , K6 , K5 , K4 , K3 , K2 , K1 , K0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

**Note 1:** E0 and E1 are also used by UART0 for Rx and Tx - so if you aren't using UART0 then you can use them as I/O pins.

**Note 2:** H0 and H1 are also used by UART2 for Rx and Tx - so if you aren't using UART2 then you can use them as I/O pins.

**Note 3:** J0 and J1 are also used by UART3 for Rx and Tx - so if you aren't using UART3 then you can use them as I/O pins.

**Note 4:** G5 is connected to the button so don't make it into an output.

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	--
0	B	OC0B	8	G5
1	A	OC1A	16	--
1	B	OC1B	16	B6



Timer	Channel	Alias	Bits	IO Pin
1	C	OC1C	16	--
2	A	OC2A	8	--
2	B	OC2B	8	H6
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5
4	A	OC4A	16	H3
4	B	OC4B	16	H4
4	C	OC4C	16	H5
5	A	OC5A	16	--
5	B	OC5B	16	--
5	C	OC5C	16	--

**Note 1:** Do not use G5 as an output (PWM etc) since it is connected to the button. Doing so may blow up the processor!


**Note 2:** B6 is connected to the green status LED so if you use it as an output (PWM etc) then the error flashing feature of the library will be disabled.

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D4
2	ICP2	--
3	ICP3	E7
4	ICP4	--
5	ICP5	--



## sys/axon2.h

This is the system file you should include if you are developing for the Axon II. 

It will make sure that your compiler settings are appropriate: ie ATmega640 running at 16MHz.

The following functions are available for the additional hardware on this board:-

void statusLED\_on() - turn on the decimal point of the LED display

void statusLED\_off() - turn off the decimal point of the LED display

boolean button\_pressed(void) - has the surface mounted button on the board been pressed?

The on board display is also automatically defined as:

```
SEGLED led_display = MAKE_SEGLED(C3,C2,C0,D6,D7,C4,C5,null,FALSE);
```

This means you can turn individual segments of the display on or off as well as write a character to the display. See "*segled.h*" (see page 63) for a full set of the commands you can use. For example:

```
segled_put_char(&led_display, 'A');
```

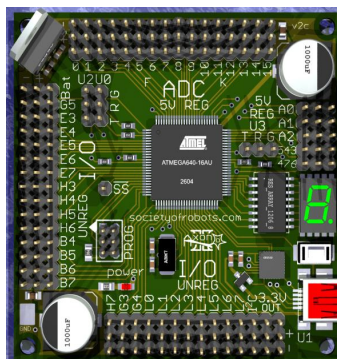
will write the character A to the display. Whereas:

```
segled_on(&led_display, SEGMENT_D);
```

will turn on segment D of the display - leaving the other segments untouched.

Note that the decimal point of the Axon II display (SEGMENT\_H) is used by this library for indicating any errors.

The library will also make the on board display available as a Marquee called 'marquee'- see "*segled.h*" (see page 63) . This allows you to use `rprintf` to output text to the display which is then scrolled across the display.





The following IOPins are available:

G5 , E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , H0 , H1 , H2 , H3 , H4 , H5 , H6 , B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , H7 , G3 , G4 , L0 , L1 , L2 , L3 , L4 , L5 , L6 , L7 , D0 , D1 , D4 , D5 , D6 , D7 , G0 , G1 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , J0 , J1 , J2 , J3 , J4 , J5 , J6 , G2 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , J7 , K7 , K6 , K5 , K4 , K3 , K2 , K1 , K0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

**Note 1:** E0 and E1 are also used by UART0 for Rx and Tx - so if you aren't using UART0 then you can use them as I/O pins.

**Note 2:** H0 and H1 are also used by UART2 for Rx and Tx - so if you aren't using UART2 then you can use them as I/O pins.

**Note 3:** J0 and J1 are also used by UART3 for Rx and Tx - so if you aren't using UART3 then you can use them as I/O pins.

**Note 4:** D5 is connected to the button so don't make it into an output.

**Note 5:** B0, B1, B2 and B3 are only available on the ISP connector

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B7
0	B	OC0B	8	G5
1	A	OC1A	16	B5
1	B	OC1B	16	B6
1	C	OC1C	16	B7
2	A	OC2A	8	B4
2	B	OC2B	8	H6
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5
4	A	OC4A	16	H3
4	B	OC4B	16	H4
4	C	OC4C	16	H5
5	A	OC5A	16	L3
5	B	OC5B	16	L4
5	C	OC5C	16	L5

**Note 1:** Note that B7 is used as the output pin for two of the timers. This means you can only use one of them at a time.

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D4



**Timer Alias IO Pin**

2	ICP2	--
3	ICP3	E7
4	ICP4	L0
5	ICP5	L1



## sys/babyOrangutan168.h

Defines a baby orangutan board using a 20MHz ATmega168.

If you need to use the on board motor controller then add

```
#define USE_ON_BOARD_MOTORS
```

to the top of your main program before including this file.

The motors will then be created and initialised automatically with the names Motor1 and Motor2.

The following IOPins are available:

D0 , D1 , D2 , D3 , D4 , D7 , B0 , B3 , B4 , B5 , C0 , C1 , C2 , C3 , C4 , C5

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	--
0	B	OC0B	8	--
1	A	OC1A	16	--
1	B	OC1B	16	--
2	A	OC2A	8	B3
2	B	OC2B	8	D3

**Note 1:** Timer 0 and Timer 1 are used to control the on-board motor controller and so the output pins are not available. However: if you are not going to use the motor controller then you can still use these timers for timing functions that do not utilise the output pins.

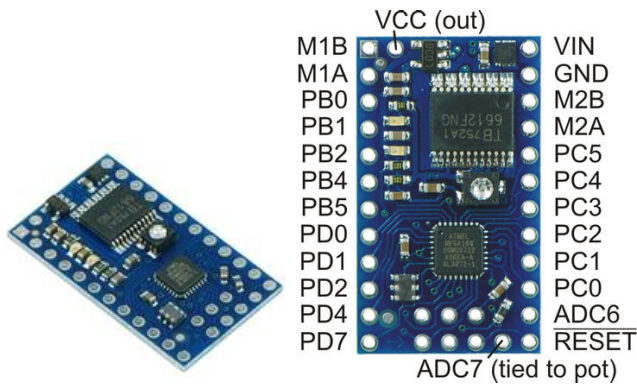
Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	B0
2	ICP2	--



## sys/babyOrangutanB328.h

Defines a baby orangutan board using a 20MHz ATmega328P.



If you need to use the on board motor controller then add

```
#define USE_ON_BOARD_MOTORS
```

to the top of your main program before including this file.

The motors will then be created and initialised automatically with the names Motor1 and Motor2.

The following IOPins are available:

D0 , D1 , D2 , D4 , D7 , B0 , B1 , B2 , B4 , B5 , C0 , C1 , C2 , C3 , C4 , C5

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	--
0	B	OC0B	8	--
1	A	OC1A	16	B1
1	B	OC1B	16	B2
2	A	OC2A	8	--
2	B	OC2B	8	--

**Note 1:** Timer 0 and Timer 1 are used to control the on-board motor controller and so the output pins are not available. However: if you are not going to use the motor controller then you can still use these timers for timing functions that do not utilise the output pins.

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	B0
2	ICP2	--



## sys/roboduino.h

The system file for the Roboduino.



This will assume an ATmega168 running at 16MHz.

The board provides 6 ADC channels on IOPins C0...C5 and 14 digital pins connected on D0...D7 and B0...B5.

A status LED is connected to digital pin 13 (B5) and is used by the library to flash any library error messages.

The following IOPins are available:

D0 , D1 , D2 , D3 , D4 , B6 , B7 , D5 , D6 , D7 , B0 , B1 , B2 , B3 , B4 , B5 , C0 , C1 , C2 , C3 , C4 , C5

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	D6
0	B	OC0B	8	D5
1	A	OC1A	16	B1
1	B	OC1B	16	B2
2	A	OC2A	8	B3
2	B	OC2B	8	D3

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	B0
2	ICP2	--



## sys/atmega8.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega8 but your actual board is not listed in the sys folder. ie it is for a generic ATmega8 board. This may also be because you have created your own ATmega8 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega8

NB Support for the ATmega8 may be deprecated in future releases since it has little memory and little ability compared to the pin-compatible ATmega168 for just a 'few cents more'. The main issue is the 8k program space. As our projects become more complex then we have to decide whether or not to use floating point maths (eg real numbers). This 'luxury' requires the floating point library which immediately adds a 'hit' of about 2.5kb of program space which for the ATmega8 is over a quarter of its total size. So here is the problem: do we continue to support 8k processors or do we make 'real numbers' be built in? Let me know what you think... My own thoughts are 'libraries should help the developer of complex applications' and so the ATmega8 should go the same way as 'Windows for Workgroups', 'MS/DOS' - it has seen its day but the future is different.

The following IOPins are available:

C6 , D0 , D1 , D2 , D3 , D4 , B6 , B7 , D5 , D6 , D7 , B0 , B1 , B2 , B3 , B4 , B5 , C0 , C1 , C2 , C3 , C4 , C5

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	--
0	B	OC0B	8	--
1	A	OC1A	16	B1
1	B	OC1B	16	B2
2	A	OC2A	8	--
2	B	OC2B	8	--

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	B0
2	ICP2	--



## sys/atmega168.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega168 but your actual board is not listed in the sys folder. ie it is for a generic 168 board. This may also be because you have created your own ATmega168 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega168

The following IOPins are available:

C6 , D0 , D1 , D2 , D3 , D4 , B6 , B7 , D5 , D6 , D7 , B0 , B1 , B2 , B3 , B4 , B5 , C0 , C1 , C2 , C3 , C4 , C5

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	D6
0	B	OC0B	8	D5
1	A	OC1A	16	B1
1	B	OC1B	16	B2
2	A	OC2A	8	B3
2	B	OC2B	8	D3

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	B0
2	ICP2	--



## sys/atmega32.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega32 but your actual board is not listed in the sys folder. ie it is for a generic 32 board. This may also be because you have created your own ATmega32 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega32

The following IOPins are available:

B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B3
1	A	OC1A	16	D5
1	B	OC1B	16	D4
2	A	OC2A	8	D7

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D6
2	ICP2	--





## sys/atmega328P.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega328P but your actual board is not listed in the sys folder. ie it is for a generic 328P board. This may also be because you have created your own ATmega328P based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega328P

The following IOPins are available:

C6 , D0 , D1 , D2 , D3 , D4 , B6 , B7 , D5 , D6 , D7 , B0 , B1 , B2 , B3 , B4 , B5 , C0 , C1 , C2 , C3 , C4 , C5

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	D6
0	B	OC0B	8	D5
1	A	OC1A	16	B1
1	B	OC1B	16	B2
2	A	OC2A	8	B3
2	B	OC2B	8	D3

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	B0
2	ICP2	--



## sys/atmega1280.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega1280 but your actual board is not listed in the sys folder. ie it is for a generic ATmega1280 board. This may also be because you have created your own ATmega1280 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega1280

The following IOPins are available:

G5 , E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , H0 , H1 , H2 , H3 , H4 , H5 , H6 , B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , H7 , G3 , G4 , L0 , L1 , L2 , L3 , L4 , L5 , L6 , L7 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , G0 , G1 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , J0 , J1 , J2 , J3 , J4 , J5 , J6 , G2 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , J7 , K7 , K6 , K5 , K4 , K3 , K2 , K1 , K0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B7
0	B	OC0B	8	G5
1	A	OC1A	16	B5
1	B	OC1B	16	B6
1	C	OC1C	16	B7
2	A	OC2A	8	B4
2	B	OC2B	8	H6
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5
4	A	OC4A	16	H3
4	B	OC4B	16	H4
4	C	OC4C	16	H5
5	A	OC5A	16	L3
5	B	OC5B	16	L4
5	C	OC5C	16	L5

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D4



**Timer Alias IO Pin**

2	ICP2	--
3	ICP3	E7
4	ICP4	L0
5	ICP5	L1



## sys/atmega2560.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega2560 but your actual board is not listed in the sys folder. ie it is for a generic ATmega2560 board. This may also be because you have created your own ATmega2560 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega2560

The following IOPins are available:

G5 , E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , H0 , H1 , H2 , H3 , H4 , H5 , H6 , B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , H7 , G3 , G4 , L0 , L1 , L2 , L3 , L4 , L5 , L6 , L7 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , G0 , G1 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , J0 , J1 , J2 , J3 , J4 , J5 , J6 , G2 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , J7 , K7 , K6 , K5 , K4 , K3 , K2 , K1 , K0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B7
0	B	OC0B	8	G5
1	A	OC1A	16	B5
1	B	OC1B	16	B6
1	C	OC1C	16	B7
2	A	OC2A	8	B4
2	B	OC2B	8	H6
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5
4	A	OC4A	16	H3
4	B	OC4B	16	H4
4	C	OC4C	16	H5
5	A	OC5A	16	L3
5	B	OC5B	16	L4
5	C	OC5C	16	L5

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D4



**Timer Alias IO Pin**

2	ICP2	--
3	ICP3	E7
4	ICP4	L0
5	ICP5	L1



## sys/atmega2561.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega2561 but your actual board is not listed in the sys folder. ie it is for a generic ATmega2561 board. This may also be because you have created your own ATmega2561 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega2561

The following IOPins are available:

G5 , E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , G3 , G4 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , G0 , G1 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , G2 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B7
0	B	OC0B	8	G5
1	A	OC1A	16	B5
1	B	OC1B	16	B6
1	C	OC1C	16	B7
2	A	OC2A	8	B4
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D4
2	ICP2	--
3	ICP3	--



## sys/atmega128.h

This is the system file you should include as a matter of last resort if you are developing for the ATmega128 but your actual board is not listed in the sys folder. ie it is for a generic ATmega128 board. This may also be because you have created your own ATmega128 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega128

The following IOPins are available:

E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , G3 , G4 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , G0 , G1 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , G2 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B4
1	A	OC1A	16	B5
1	B	OC1B	16	B6
1	C	OC1C	16	B7
2	A	OC2A	8	B7
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D4
2	ICP2	--
3	ICP3	E7



## sys/atmega640.h

This is the system file you should include if you are developing for a generic ATmega640 board.

It will make sure that your compiler settings are appropriate: ie ATmega640 running at 16MHz.

This is the system file you should include as a matter of last resort if you are developing for the ATmega640 but your actual board is not listed in the sys folder. ie it is for a generic ATmega640 board. This may also be because you have created your own ATmega640 based board.

Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega640

The following IOPins are available:

G5 , E0 , E1 , E2 , E3 , E4 , E5 , E6 , E7 , H0 , H1 , H2 , H3 , H4 , H5 , H6 , B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , H7 , G3 , G4 , L0 , L1 , L2 , L3 , L4 , L5 , L6 , L7 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , G0 , G1 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , J0 , J1 , J2 , J3 , J4 , J5 , J6 , G2 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0 , J7 , K7 , K6 , K5 , K4 , K3 , K2 , K1 , K0 , F7 , F6 , F5 , F4 , F3 , F2 , F1 , F0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B7
0	B	OC0B	8	G5
1	A	OC1A	16	B5
1	B	OC1B	16	B6
1	C	OC1C	16	B7
2	A	OC2A	8	B4
2	B	OC2B	8	H6
3	A	OC3A	16	E3
3	B	OC3B	16	E4
3	C	OC3C	16	E5
4	A	OC4A	16	H3
4	B	OC4B	16	H4
4	C	OC4C	16	H5
5	A	OC5A	16	L3
5	B	OC5B	16	L4
5	C	OC5C	16	L5





Table 2 - The following input capture pins are available on timers:

**Timer Alias IO Pin**

0	ICP0	--
1	ICP1	D4
2	ICP2	--
3	ICP3	E7
4	ICP4	L0
5	ICP5	L1



## sys/atmega644.h

This is the system file you should include if you are developing for a generic ATmega644 board.

It will make sure that your compiler settings are appropriate: ie ATmega644 running at 16MHz.

This is the system file you should include as a matter of last resort if you are developing for the ATmega644 but your actual board is not listed in the sys folder. ie it is for a generic ATmega644 board. This may also be because you have created your own ATmega644 based board. Consequently the library assumes that all processor pins have header pins.

It will make sure that your compiler settings are appropriate: ie ATmega644

The following IOPins are available:

B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 , D0 , D1 , D2 , D3 , D4 , D5 , D6 , D7 , C0 , C1 , C2 , C3 , C4 , C5 , C6 , C7 , A7 , A6 , A5 , A4 , A3 , A2 , A1 , A0

Table 1 - The following timer pins are available:

Timer	Channel	Alias	Bits	IO Pin
0	A	OC0A	8	B3
0	B	OC0B	8	B4
1	A	OC1A	16	D5
1	B	OC1B	16	D4
2	A	OC2A	8	D7
2	B	OC2B	8	D6

Table 2 - The following input capture pins are available on timers:

Timer	Alias	IO Pin
0	ICP0	--
1	ICP1	D6
2	ICP2	--



## DC Motors

Support for various DC motor controllers.

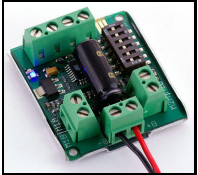
DimensionEngineering/Sabertooth.h	172
Pololu/DualSerial.h	176
Sanyo/LB1836M.h	178
Solarbotics/L298.h	179
Toshiba/TB6612FNG.h	181
L293D or SN754410	186



## DC Motors/DimensionEngineering/Sabertooth.h

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 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 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.

In order to use the boards you must first create each motor by using 'MAKE\_SABERTOOTH\_MOTOR' assigning each motor an address (if using Packetized mode - as per the DIP switches) and a motor number (1 or 2 - since each board supports 2 motors) and then combine them all into a driver using 'MAKE\_SABERTOOTH\_DRIVER' which then allows you to specify which mode you want to use.

This driver is then passed to 'sabertoothInit' when your application is initialising (in applInitHardware). Your main program can then set the speed of each motor using act\_setSpeed in the same way as for any other servo or motor.



Here is a small example for the Axon that makes the motors spin forward and backward at different speeds and assumes that the board address is set as 128:-

```
#include "sys/axon.h" // We are using the Axon
// Include files
#include "uart.h"
#include "Motors/DimensionEngineering/Sabertooth.h"

// Create motors 0 and 1 at address 128
SABERTOOTH_MOTOR motor1 = MAKE_SABERTOOTH_MOTOR(FALSE, 128,1);
SABERTOOTH_MOTOR motor2 = MAKE_SABERTOOTH_MOTOR(FALSE, 128,2);

// Create a driver that includes them both and uses UART0 at 19200 baud
SABERTOOTH_MOTOR_LIST motors[] = {&motor1,&motor2};
// Create the driver using simple serial mode....
SABERTOOTH_DRIVER driver = MAKE_SABERTOOTH_DRIVER(motors, UART0, 19200, SIMPLE);
// ... or use packetized mode if you have more than one board
//SABERTOOTH_DRIVER driver = MAKE_SABERTOOTH_DRIVER(motors, UART0, 19200,
PACKETIZED);

// Initialise the driver in my start up code
void appInitHardware(void){
    sabertoothInit(&driver);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
// Move the motors back and forth
int dir=1; // Are we increasing speed (+1) or decreasing speed (-1)
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Get the current speed of motor 2
    DRIVE_SPEED speed = act_getSpeed(&motor2);
    // Increase or decrease it
    speed += dir;
    // Change direction if we've hit the end stop
    if(speed == DRIVE_SPEED_MAX){
        dir = -1;
    }else if(speed == DRIVE_SPEED_MIN){
        dir = 1;
    }
    // Set the speed of both motors
    act_setSpeed(&motor1,speed);
    act_setSpeed(&motor2,speed);
    return 40000; // Only call me every 40ms
}
```

## Standard Function Summary

SABERTOOTH\_MOTOR

[MAKE SABERTOOTH\\_MOTOR\(boolean inverted, uint8 t address, uint8 t motornumber\)](#)

Defines a motor on a controller board.



## Standard Function Summary

SABERTOOTH\_DRIVER

[MAKE SABERTOOTH DRIVER\(SABERTOOTH MOTOR LIST motors, UART\\* uart, BAUD\\_RATE baudrate, SABERTOOTH\\_MODE mode\)](#)

Groups together all of the motors, from as many boards as you like in packetized mode or from one board if using simple mode, that are driven via the same UART.

[sabertoothInit\(SABERTOOTH\\_DRIVER\\* driver\)](#)

Initialise the motor driver.

## Standard Function Detail

### MAKE\_SABERTOOTH\_MOTOR

SABERTOOTH\_MOTOR MAKE\_SABERTOOTH\_MOTOR(boolean inverted, uint8\_t address, uint8\_t motornumber)

Defines a motor on a controller board.

The inverted parameter specifies whether or not the motor should turn in the opposite direction ie if the wheels are mounted on the left and right of the robot then one motor needs to turn clockwise and the other anti-clockwise in order to go forwards. So one motor should be defined as 'inverted'.

The address parameter defines the address of the board - as set on the dip switches.

The motornumber parameter specifies the motor number on the board ie motor 0 or motor 1.

For example: to define motor 0 on a board at address 128 then declare the motor as follows:-

```
SABERTOOTH_MOTOR motor1 = MAKE_SABERTOOTH_MOTOR(FALSE, 128, 0);
```

Having created the individual motors you need to group them into a list:-

```
SABERTOOTH_MOTOR_LIST motors[] = {&motor1, &motor2};
```

Note the '&' before the name of each motor.

### MAKE\_SABERTOOTH\_DRIVER

SABERTOOTH\_DRIVER MAKE\_SABERTOOTH\_DRIVER(SABERTOOTH\_MOTOR\_LIST motors, UART\* uart, BAUD\_RATE baudrate, SABERTOOTH\_MODE mode)

Groups together all of the motors, from as many boards as you like in packetized mode or from one board if using simple mode, that are driven via the same UART.



The first parameter is the list of motors and is followed by the UART to be used and the required baud rate.

The mode parameter can be either PACKETIZED or SIMPLE.

NB Check the data sheets to see what baud rates are allowed. When using PACKETIZED mode you must set up the DIP switches to match the board address and the baud rate is auto-detected. In SIMPLE mode then the DIP switches must be set to match the baud rate you are using.

---

## **sabertoothInit**

`sabertoothInit(SABERTOOTH_DRIVER* driver)`

Initialise the motor driver.

This should be called once - in your initialisation code: `apInitHardware()`;

This will setup the baud rate for the UART and set all motors to a speed of zero. As per the manual: this method will add a delay of 2 seconds before returning in order to give the boards a chance to initialise themselves.

You may now change the speed of the motors in the usual way via:-

```
act_setSpeed(motor1, speed);
```

Look at `actuators.h` for more information. Note especially that this provides the `act_getSpeed` call to query the current speed - which can save your main loop from trying to store this in yet another variable. Obviously the value returned is the last value you set using `setSpeed` - ie its the 'last requested speed' not the 'physical current speed'. To find the actual speed you would need some encoder hardware and use the encoder support provided elsewhere in this library.



## DC Motors/Pololu/DualSerial.h

Adds support for various motor controllers from Pololu.

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. The UART sends a 4 byte sequence to set the speed of a motor as follows:- 0x80, 0x00, b0mmmmmmmd, b0sssssss where 'mmmmmm' is the motor number, 'd' is the direction 1=fwd 0=rev and 'sssssss' is the speed.

In order to use the boards you must first create each motor by using 'MAKE\_POLOLU\_DS\_MOTOR' and then combine them all into a driver using 'MAKE\_POLOLU\_DS\_DRIVER'.

The driver is then passed to 'pololuDualSerialInit' when your application is initialising (in applInitHardware). Your main program can then set the speed of each motor using act\_setSpeed in the same way as for any other servo or motor.

Here is a small example for the Axon that makes the motors spin forward and backward at different speeds:-





```
#include "sys/axon.h" // We are using the Axon
// Include files
#include "uart.h"
#include "Motors/Pololu/DualSerial.h"

// Create two motors
POLOLU_DS_MOTOR motor1 = MAKE_POLOLU_DS_MOTOR(FALSE);
POLOLU_DS_MOTOR motor2 = MAKE_POLOLU_DS_MOTOR(FALSE);

// Create a driver that includes them both and uses UART0 at 19200 baud.
// The list of motors starts at motor number '2'
POLOLU_DS_MOTOR_LIST motors[] = {&motor1,&motor2};
POLOLU_DS_DRIVER driver = MAKE_POLOLU_DS_DRIVER(motors, UART0, 19200, 2);

// Initialise the hardware driver in my start up code
void appInitHardware(void){
    pololuDualSerialInit(&driver);
}

TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}

// Move the motors back and forth
int dir=1;
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Get the current speed of motor 2
    DRIVE_SPEED speed = act_getSpeed(&motor2);
    // Increase or decrease it
    speed += dir;
    // Change direction if we've hit the end stop
    if(speed == DRIVE_SPEED_MAX){
        dir = -1;
    }else if(speed == DRIVE_SPEED_MIN){
        dir = 1;
    }
    // Set the speed of both motors
    act_setSpeed(&motor1,speed);
    act_setSpeed(&motor2,speed);
    return 40000; // only call me every 40ms
}
```



## DC Motors/Sanyo/LB1836M.h

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.

Here is an example of how it can be used:

```
// Define the motors
SANYO_LB1836M_MOTOR Motor1 = MAKE_SANYO_LB1836M_MOTOR(FALSE, B1, D5);
SANYO_LB1836M_MOTOR Motor2 = MAKE_SANYO_LB1836M_MOTOR(FALSE, B2, D6);
SANYO_LB1836M_MOTOR_LIST onboard_motors[] = {&Motor1, &Motor2};
SANYO_LB1836M_MOTOR_DRIVER onboard_driver =
MAKE_SANYO_LB1836M_MOTOR_DRIVER(onboard_motors);
```

Inside 'applInitHardware':-

```
// Initialise the driver
sanyoLB1836M_Init(&onboard_driver);
```

Note that if you are using the BabyOrangutan then this has already done for you as it is built onto the board. Refer to the relevant 'sys' file for your BabyOrangutan board.



## DC Motors/Solarbotics/L298.h

Solarbotics L298 motor driver.



See

[http://www.solarbotics.com/assets/datasheets/solarbotics\\_l298\\_compact\\_motor\\_driver\\_kit.pdf](http://www.solarbotics.com/assets/datasheets/solarbotics_l298_compact_motor_driver_kit.pdf)  
for the datasheet.

If you create the additional circuitry shown on page 12 of the datasheet (3 resistors and one transistor) then you only need one IOPin per motor and it can be driven using the code in `motorPWM.h`

Otherwise you need to include this file.

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.

If you attempt to use pins that are not valid: say because they are not PWM pins, or the timer doesn't provide hardware PWM then error codes will be set on the status LED.

The remaining two pins are the 'enable' pins. If the motor is turning the wrong way then you can either swap the two enable pins or you can toggle the first parameter to `MAKE_SOLAR_L298_MOTOR`.

It's very easy to use: you just give it a list of motors and that's it. The motor can then be controlled using the commands in `actuators.h`.

Note that setting a speed of 0 will cause the motor to brake whereas disconnecting the motor will cause it to coast.

Here is an example of the code in your application:-



```
#include "Motors/Solarbotics/L298.h"
// Define two motors
SOLAR_L298_MOTOR left = MAKE_SOLAR_L298_MOTOR(FALSE, B1,B4,B5);
SOLAR_L298_MOTOR right = MAKE_SOLAR_L298_MOTOR(TRUE , B2,B6,B7);
```

```
// Create the list - remember to place an & at the
// start of each motor name
SOLAR_L298_MOTOR_LIST motors[] = {&left,&right};
```

```
// Create a driver for the list of motors
SOLAR_L298_MOTOR_DRIVER bank1 = MAKE_SOLAR_L298_MOTOR_DRIVER(motors);
```

```
// In my initialising code - pass the list of motors to control
void appInitHardware(void){
    // Initialise the motors
    solarL298Init(&bank1);
}
```

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Give each servo a start value - make them go full ahead
    act_setSpeed(&left,DRIVE_SPEED_MAX);
    act_setSpeed(&right,DRIVE_SPEED_MAX);
    return 0;
}
```

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    .. nothing to do. But in reality I would use act_speed to change the speed of
    each motor ...
    return 0;
}
```



## DC Motors/Toshiba/TB6612FNG.h

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.



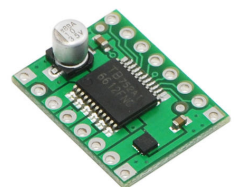
The breakout board comes with header pins for use with a bread board and must be soldered on. Alternatively: for use in a finished robot you can just solder your wires directly to the board. The board also comes with some capacitors and reverse voltage protection for the motor supply.

I have supplied two different software drivers for this board:

TB6612FNG\_2pin.h 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.

TB6612FNG\_3pin.h 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 add to utilise this feature then connect it to any 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.

### 3 Pin

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.

Here is an example of how you could define two motors on the Axon:-

```
#include <Motors/Toshiba/TB6612FNG.h>
TOSHIBA_TB6612FNG_3pin_MOTOR left =
MAKE_TOSHIBA_TB6612FNG_MOTOR_3pin(FALSE,E3,F0,F1);
TOSHIBA_TB6612FNG_3pin_MOTOR right =
MAKE_TOSHIBA_TB6612FNG_MOTOR_3pin(TRUE,E4,F2,F3);
TOSHIBA_TB6612FNG_3pin_MOTOR_LIST motors[] = {&left, &right};
TOSHIBA_TB6612FNG_3pin_MOTOR_DRIVER driver =
MAKE_TOSHIBA_TB6612FNG_3pin_MOTOR_DRIVER(motors);
void appInitHardware(void){
    toshibaTB6612FNG_3pin_Init(&driver);
}
```

## 2 Pin

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.

Here is an example of how you could define one motor on the Axon:-

```
#include <Motors/Toshiba/TB6612FNG.h>
TOSHIBA_TB6612FNG_2pin_MOTOR left =
MAKE_TOSHIBA_TB6612FNG_MOTOR_2pin(FALSE,E3,E4);
TOSHIBA_TB6612FNG_2pin_MOTOR_LIST motors[] = {&left};
TOSHIBA_TB6612FNG_2pin_MOTOR_DRIVER driver =
MAKE_TOSHIBA_TB6612FNG_2pin_MOTOR_DRIVER(motors);
void appInitHardware(void){
    toshibaTB6612FNG_2pin_Init(&driver);
}
```

## Common

Whether you are using the 2pin or 3pin the remaining code remains the same. You can use `act_SetSpeed` to set the speed of the motor.

### Standard Function Summary

	<a href="#">MAKE_TOSHIBA_TB6612FNG_MOTOR_3pin(inverted, pwm, enable1, enable2)</a> Create a motor using the 3 pin connection method.
	<a href="#">MAKE_TOSHIBA_TB6612FNG_3pin_MOTOR_DRIVER(motorlst)</a> Create the driver to control a list of 3 pin motors.
	<a href="#">toshibaTB6612FNG_3pin_Init(TOSHIBA_TB6612FNG_3pin_MOTOR_DRIVER* driver)</a> Initialise the 3 pin driver.
	<a href="#">MAKE_TOSHIBA_TB6612FNG_MOTOR_2pin(inverted, pwm1, pwm2)</a> Create a motor using the 2 pin connection method.
	<a href="#">MAKE_TOSHIBA_TB6612FNG_2pin_MOTOR_DRIVER(motorlst)</a> Create the driver to control a list of 2 pin motors.
	<a href="#">toshibaTB6612FNG_2pin_Init(TOSHIBA_TB6612FNG_2pin_MOTOR_DRIVER* driver)</a> Initialise the 2 pin driver.



## Standard Function Detail

### MAKE\_TOSHIBA\_TB6612FNG\_MOTOR\_3pin

MAKE\_TOSHIBA\_TB6612FNG\_MOTOR\_3pin(inverted, pwm, enable1, enable2)

Create a motor using the 3 pin connection method.

The first pin must be a PWM output pin, whereas the remaining two pins can be any output pins.

Example on the Axon:

```
TOSHIBA_TB6612FNG_3pin_MOTOR motor = MAKE_TOSHIBA_TB6612FNG_MOTOR_3pin(FALSE, E3, F0, F1);
```

If this is being used to control motor 1 then E3 is connected to PWMA, F0 is connected to AIN1, F1 is connected to AIN2. For motor2 then E3 is connected to PWMB, F0 is connected to BIN1, F1 is connected to BIN2.

---

### MAKE\_TOSHIBA\_TB6612FNG\_3pin\_MOTOR\_DRIVER

MAKE\_TOSHIBA\_TB6612FNG\_3pin\_MOTOR\_DRIVER(motorlst)

Create the driver to control a list of 3 pin motors.

```
TOSHIBA_TB6612FNG_3pin_MOTOR_LIST motors[] = { &motor1, &motor2 }; // as many as you like
TOSHIBA_TB6612FNG_3pin_MOTOR_DRIVER driver =
MAKE_TOSHIBA_TB6612FNG_3pin_MOTOR_DRIVER(motors);
```

---

### toshibaTB6612FNG\_3pin\_Init

toshibaTB6612FNG\_3pin\_Init( TOSHIBA\_TB6612FNG\_3pin\_MOTOR\_DRIVER\* driver )

Initialise the 3 pin driver.

This should be called in applnitHardware.

---

### MAKE\_TOSHIBA\_TB6612FNG\_MOTOR\_2pin

MAKE\_TOSHIBA\_TB6612FNG\_MOTOR\_2pin(inverted, pwm1, pwm2)

Create a motor using the 2 pin connection method.

Both pins must be PWM output pins or else the library will generate a runtime error.

Example on the Axon:

```
TOSHIBA_TB6612FNG_2pin_MOTOR motor = MAKE_TOSHIBA_TB6612FNG_MOTOR_2pin(FALSE, E3, E4);
```





## **MAKE\_TOSHIBA\_TB6612FNG\_2pin\_MOTOR\_DRIVER**

**MAKE\_TOSHIBA\_TB6612FNG\_2pin\_MOTOR\_DRIVER**(motorlst)

Create the driver to control a list of 2 pin motors.

```
TOSHIBA_TB6612FNG_2pin_MOTOR_LIST motors[] = { &motor1, &motor2 }; // as many
as you like
TOSHIBA_TB6612FNG_2pin_MOTOR_DRIVER driver =
MAKE_TOSHIBA_TB6612FNG_2pin_MOTOR_DRIVER(motors);
```

---

## **toshibaTB6612FNG\_2pin\_Init**

**toshibaTB6612FNG\_2pin\_Init**( TOSHIBA\_TB6612FNG\_2pin\_MOTOR\_DRIVER\* driver  
)

Initialise the 2 pin driver.

This should be called in applInitHardware.



## DC Motors/L293D or SN754410

Support for the L293D and plug-in replacements such as the 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.

These devices can be used by adding:

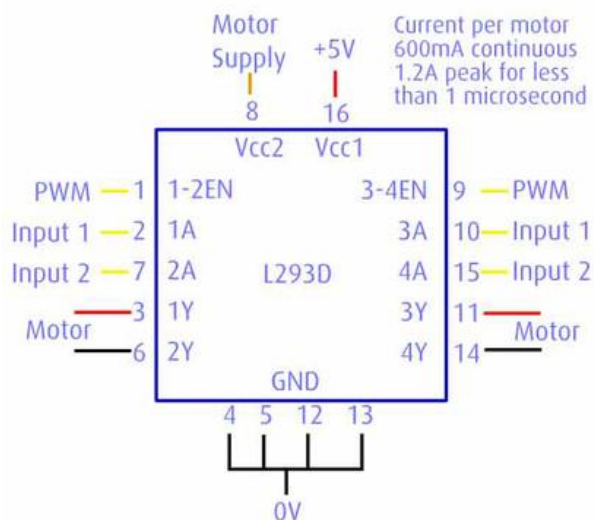
```
#include <motorPWM.h>
```

to the start of your program.

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 alternative between 'full speed' and 'coast'.

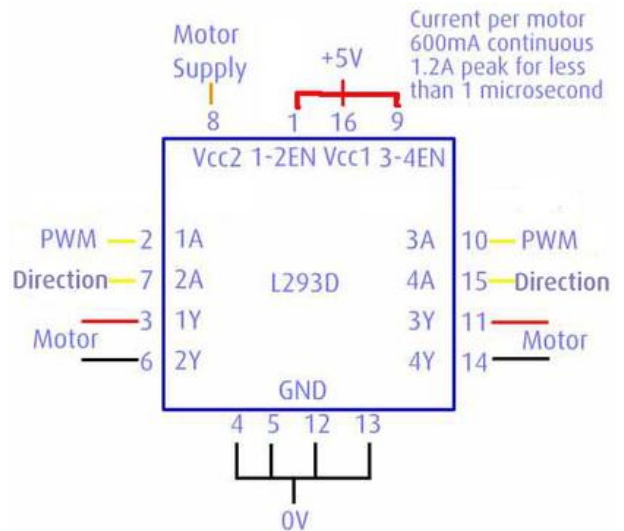
To create a 3 pin motor then use  
`"MAKE_MOTOR_3_PIN(inverted, pwmPin, input1,input2)"` (see page 78)



## 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.

To create a 2 pin motor then use  
`"MAKE_MOTOR(inverted, motorPin, directionPin)"`  
 (see page 78)



## Either Mode

Once you have defined each motor then you need to combine them into a MOTOR\_LIST. This list is then passed to a MOTOR\_DRIVER, and finally the driver is initialised using 'motorL293Init' or 'motorSN754410Init'.

## Example

Ok so here is an example for the Axon. Lets assume we have 2 motors called 'motor1' and 'motor2' and we are going to use the 3 Pin Mode for both of them. Here is the top of your program:-

```
#include <motorPWM.h>
MOTOR motor1 = MAKE_MOTOR_3_PIN(false,H5,F0,F1);
MOTOR motor2 = MAKE_MOTOR_3_PIN(false,H4,F2,F3);
static MOTOR_LIST motors_list[] = {&motor1,&motor2};
MOTOR_DRIVER motors = MAKE_MOTOR_DRIVER(motors_list);
```

Then in appInitHardware we need to initialise the driver:-

```
motorL293Init(&motors);
```

All done. Now we can control motor1 and motor2 using the commands in "actuators.h" (see page 23)

For example - this appControl logic will alternate between rotating the motors backward and forward every 10 seconds:-



```
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
    TICK_COUNT ms = loopStart / 1000; // Get current time in ms
    int16_t now = ms % (TICK_COUNT)10000; // 10 sec for a full swing
    if(now >= (int16_t)5000){ // Goes from 0ms...5000ms
        now = (int16_t)10000 - now; // then 5000ms...0ms
    }
    // 'now' has a value from 0 to 5000
    // Map it into DRIVE_SPEED range
    DRIVE_SPEED speed = interpolate(now, 0, 5000, DRIVE_SPEED_MIN,
DRIVE_SPEED_MAX);
    // Set speed for all motors/servos
    act_setSpeed(&motor1,speed);
    act_setSpeed(&motor2,speed);
    return 0;
}
```



## 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  $360^\circ / 200$  ie  $1.8^\circ$  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 knew.

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.

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).

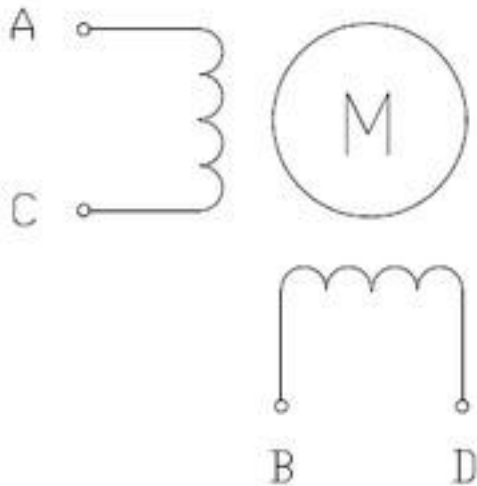


The next consideration is that there are two different categories of stepper motor: **bi-polar** and **uni-polar**.

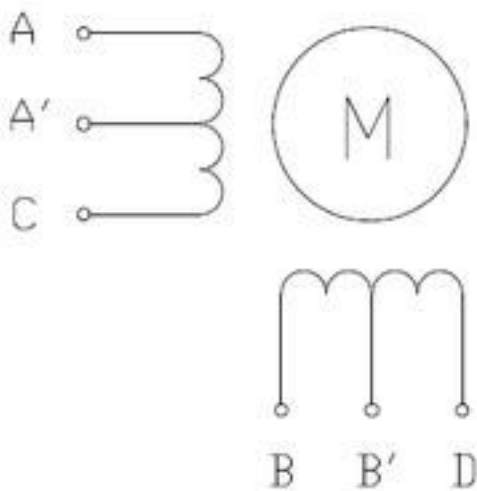
If you have already purchased a motor, or re-cycled one from some old hardware, then you need to determine the category.

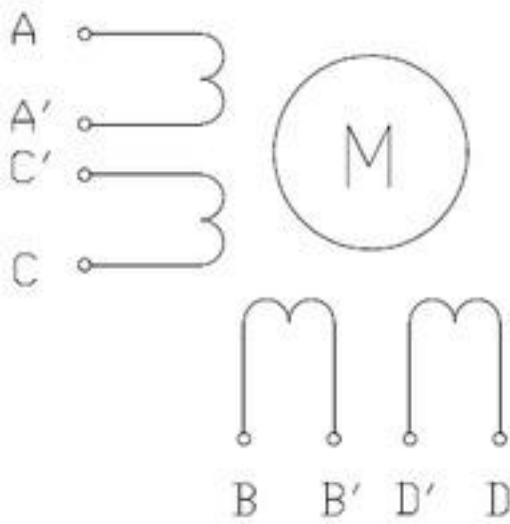
Generally: a bi-polar motor has 4 connections and a uni-polar has either 6 or 8 connections.

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.

## Drive Modes

All stepper motor drivers 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 drivers 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 drivers 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 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 as having 400 steps/revolution.

## Slip

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.

<code>_stepper_common.h</code>	193
<code>Pololu/A4983.h</code>	197
<code>Generic/L297.h</code>	199
<code>Generic/Bipolar.h</code>	201





## Stepper/\_stepper\_common.h

Provides generic support for all stepper motor controllers to set continuous rotation speeds or the (servo - like) movement to a given location.

This driver allows you to build a list of stepper motor controllers and then attach the list to a timer to generate the necessary pulses for each motor. The list can contain any mixture of different stepper motor controllers.

Assuming we have two Pololu A4983 stepper motor controllers then we can define each motor as follows:

```
#include <Stepper/Pololu/A4983.h>
POLOLU_A4983 motorA = MAKE_POLOLU_A4983(false,200,127,1,A0,A1,null,null);
POLOLU_A4983 motorB = MAKE_POLOLU_A4983(false,200,127,1,A2,A3,null,null);
```

The detail of the parameters for MAKE\_POLOLU\_A4983 is described in the section for that controller.

We can now combine them into a list called 'bank':

```
STEPPER_LIST bank[] = { &motorA.stepper, &motorB.stepper};
```

Then we can pass the list to a driver:

```
STEPPER_DRIVER driver = MAKE_STEPPER_DRIVER(bank,200);
```

Finally you need to initialise the driver from 'applnitHardware':

```
stepper_init(&driver,TIMER1);
```

You can now use the motors via the functions described in this section.

### Standard Function Summary

[MAKE STEPPER DRIVER\(STEPPER LIST\\* motors, uin16 t frequency\)](#)

Create a driver to communicate with one, or more, stepper motor controllers.

[stepper\\_init\(STEPPER\\_DRIVER\\* driver, const Timer\\* timer\)](#)

Initialises the driver and connects it to a timer.

[stepper setSpeed\(STEPPER MOTOR\\* motor, DRIVE\\_SPEED speed\)](#)

Set the continuous rotation speed to a value between DRIVE\_SPEED\_MIN and DRIVE\_SPEED\_MAX.



## Standard Function Summary

DRIVE\_SPEED

[stepper\\_getSpeed\(const STEPPER MOTOR\\* motor\)](#)

Returns the last value from 'stepper\_setSpeed'.

[stepper\\_setConnected\(STEPPER MOTOR\\* motor,boolean connected\)](#)

Connect or disconnect the motor.

boolean

[stepper\\_isConnected\(const STEPPER MOTOR\\* motor\)](#)

Test if a motor is connected.

boolean

[stepper\\_isInverted\(const STEPPER MOTOR\\* motor\)](#)

Test if the motor is inverted.

DRIVE\_SPEED

[stepper\\_getActualSpeed\(const STEPPER MOTOR\\* motor\)](#)

Returns the actual rotation speed of the motor.

uint16\_t

[stepper\\_getPosition\(const STEPPER MOTOR\\* motor\)](#)

Returns the current position of the motor.

[stepper\\_move\(STEPPER MOTOR\\* motor, int16 t steps\)](#)

Tell the motor to move by the specified number of steps and then stop.

int16\_t

[stepper\\_getStepsRemaining\(const STEPPER MOTOR\\* motor\)](#)

Returns the number of remaining steps from the last stepper\_move command.

## Standard Function Detail

### MAKE\_STEPPER\_DRIVER

MAKE\_STEPPER\_DRIVER(STEPPER\_LIST\* motors, uint16\_t frequency)

Create a driver to communicate with one, or more, stepper motor controllers.

The first parameter is the list of motor controllers.

The second parameter specifies the maximum step frequency in Hertz. 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.

### stepper\_init

stepper\_init(STEPPER\_DRIVER\* driver, const Timer\* timer)

Initialises the driver and connects it to a timer. This should be called from apInitHardware.



The first parameter is the address of the driver and the second parameter is an available 16 bit timer.

---

### **stepper\_setSpeed**

`stepper_setSpeed(STEPPER_MOTOR* motor, DRIVE_SPEED speed)`

Set the continuous rotation speed to a value between `DRIVE_SPEED_MIN` and `DRIVE_SPEED_MAX`.

If the motor has been disconnected then nothing will happen until it is reconnected. The motor will accelerate, or decelerate, from its current speed to the specified speed.

---

### **stepper\_getSpeed**

`DRIVE_SPEED stepper_getSpeed(const STEPPER_MOTOR* motor)`

Returns the last value from 'stepper\_setSpeed'.

NB This is not necessarily the actual speed that the motor is moving at - it is the target speed.

---

### **stepper\_setConnected**

`stepper_setConnected(STEPPER_MOTOR* motor, boolean connected)`

Connect or disconnect the motor. The second parameter should be `TRUE` to connect, or `FALSE` to disconnect.

When a motor is disconnected it stops receiving commands and, if the motor controller allows it, the current to the motor is removed.

---

### **stepper\_isConnected**

`boolean stepper_isConnected(const STEPPER_MOTOR* motor)`

Test if a motor is connected. Returns `TRUE` if it is, and `FALSE` if it is not.

---

### **stepper\_isInverted**

`boolean stepper_isInverted(const STEPPER_MOTOR* motor)`

Test if the motor is inverted. Returns `TRUE` if it is, `FALSE` if it is not.

You cannot change the inverted state except when creating the motor.

---

### **stepper\_getActualSpeed**

`DRIVE_SPEED stepper_getActualSpeed(const STEPPER_MOTOR* motor)`

Returns the actual rotation speed of the motor.

---



### **stepper\_getPosition**

`uint16_t stepper_getPosition(const STEPPER_MOTOR* motor)`

Returns the current position of the motor.

The returned value will be between 0 and the (number\_of\_steps\_per\_revolution - 1). eg if the stepper motor has 200 steps per revolution then it will be a number between 0 and 199.

---

### **stepper\_move**

`stepper_move(STEPPER_MOTOR* motor, int16_t steps)`

Tell the motor to move by the specified number of steps and then stop.

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.

---

### **stepper\_getStepsRemaining**

`int16_t stepper_getStepsRemaining(const STEPPER_MOTOR* motor)`

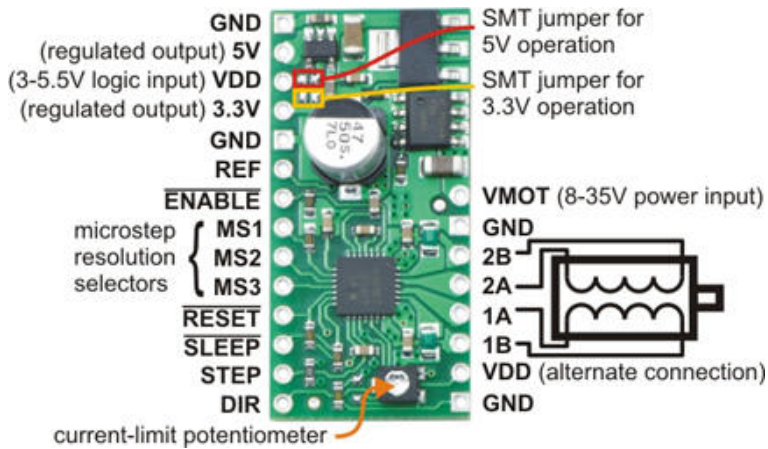
Returns the number of remaining steps from the last `stepper_move` command.

This can be used to test if a `stepper_move` command has finished by checking if the returned value is zero.



## Stepper/Pololu/A4983.h

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.

See "[\\_stepper\\_common.h](#)" (see page 193) for a description of the commands to control the motor.

### Standard Function Summary

POLOLU\_A4983

[MAKE\\_POLOLU\\_A4983\(inverted, steps, accel, every, direction, pulse, reset, enable\)](#)

Create a Pololu A4983 stepper motor controller.

### Standard Function Detail

#### MAKE\_POLOLU\_A4983

```
POLOLU_A4983 MAKE_POLOLU_A4983(inverted, steps, accel, every,
direction, pulse, reset, enable)
```

Create a Pololu A4983 stepper motor controller.

The parameters are as follows:

- **inverted** - TRUE or FALSE - if the motor is turning in the wrong direction then flip this parameter value



- steps - The number of steps per revolution. If you are micro stepping the motor then don't forget to multiply the actual value for the motor by the number of microsteps you have set up via the three pins
- accel - A number from 0 to DRIVE\_SPEED\_MAX. When the motor is changing speed it will limit the acceleration to this value
- every - The number of timer ticks between accelerating by the 'accel' value above
- direction - The IOPin used to connect to the DIR pin
- pulse - The IOPin used to connect to the STEP pin
- reset - The IOPin used to connect to the RESET pin, or may be 'null' if you don't need that functionality
- enable - The IOPin used to connect to the ENABLE pin, or may be 'null' if you don't need that functionality

Example:

Let's assume that we have a stepper motor with 200 steps per revolution and that we have configured the A983 to use half stepping to achieve 400 steps per revolution. Also assume we configure the driver to produce a maximum of 300 pulses per second. This means that the maximum speed of the motor will be 0.75 revolutions per second (ie 45 rpm).

If we used 'accel = 10' and 'every = 2' then this means the motor speed will increase by 10 units every 150th of a second (300 pulses per second divided by 2) until it reaches the requested speed. Note that values of 'accel = DRIVE\_SPEED\_MAX' and 'every = 1' will effectively give infinite acceleration.

We will use pins A0 and A1 for direction and pulse and leave the reset and enable unused.

```
POLOLU_A4983 motor = MAKE_POLOLU_A4983( FALSE, 400, 10, 2, A0, A1, null, null );
```



## Stepper/Generic/L297.h

Adds support for a generic L297 stepper motor controller.

See "*\_stepper\_common.h*" (see page 193) for a description of the commands to control the motor.

### Standard Function Summary

L297

[MAKE\\_L297\(inverted, steps, accel, every, direction, pulse, reset, enable\)](#)

Create an L297 stepper motor controller.

### Standard Function Detail

#### MAKE\_L297

```
L297 MAKE_L297(inverted, steps, accel, every,
direction, pulse, reset, enable)
```

Create an L297 stepper motor controller.

The parameters are as follows:

- inverted - TRUE or FALSE - if the motor is turning in the wrong direction then flip this parameter value
- steps - The number of steps per revolution. If you are half stepping the motor then don't forget to multiply the actual value for the motor by 2
- accel - A number from 0 to DRIVE\_SPEED\_MAX. When the motor is changing speed it will limit the acceleration to this value
- every - The number of timer ticks between accelerating by the 'accel' value above
- direction - The IOPin used to connect to the direction pin
- pulse - The IOPin used to connect to the pulse pin
- reset - The IOPin used to connect to the RESET pin, or may be 'null' if you don't need that functionality
- enable - The IOPin used to connect to the ENABLE pin, or may be 'null' if you don't need that functionality

Example:

Let's assume that we have a stepper motor with 200 steps per revolution and that we have configured the L297 to use half stepping to achieve 400 steps per revolution. Also assume we configure the driver to produce a maximum of 300 pulses per second. This means that the maximum speed of the motor will be 0.75 revolutions per second (ie 45 rpm).



If we used 'accel = 10' and 'every = 2' then this means the motor speed will increase by 10 units every 150th of a second (300 pulses per second divided by 2) until it reaches the requested speed. Note that values of 'accel = DRIVE\_SPEED\_MAX' and 'every = 1' will effectively give infinite acceleration.

We will use pins A0 and A1 for direction and pulse and leave the reset and enable unused.

```
L297 motor = MAKE_L297( FALSE, 400, 10, 2, A0, A1, null, null);
```





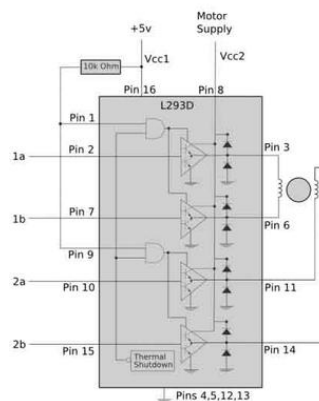
## Stepper/Generic/Bipolar.h

Adds support for a generic bipolar stepper motor controller such as the L293D.

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.

See "*\_stepper\_common.h*" (see page 193) for a description of the commands to control the motor.

If you are using an L293D 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.

### Standard Function Summary

BIPOLAR

[MAKE\\_BIPOLAR\(inverted, steps, accel, every, coil1A, coil1B, coil2A, coil2B, enable, mode\)](#)

Create a bipolar stepper motor controller.

### Standard Function Detail

#### MAKE\_BIPOLAR

BIPOLAR MAKE\_BIPOLAR(inverted, steps, accel, every, coil1A, coil1B, coil2A, coil2B, enable, mode)

Create a bipolar stepper motor controller.

The parameters are as follows:

- inverted - TRUE or FALSE - if the motor is turning in the wrong direction then flip this parameter value



- steps - The number of steps per revolution. If you are half stepping the motor then don't forget to multiply the actual value for the motor by 2
- accel - A number from 0 to DRIVE\_SPEED\_MAX. When the motor is changing speed it will limit the acceleration to this value
- every - The number of timer ticks between accelerating by the 'accel' value above
- coil1A - The IOPin used to connect to one end of coil 1
- coil1B - The IOPin used to connect to the other end of coil 1
- coil2A - The IOPin used to connect to one end of coil 2
- coil2B - The IOPin used to connect to the other end of coil 2
- enable - The IOPin used to connect to the ENABLE pin, or may be 'null' if you don't need that functionality
- mode - this sets the stepping mode and must be either FULL\_STEP or HALF\_STEP

Example:

Let's assume that we have a stepper motor with 200 steps per revolution and that we choose to use half stepping mode to achieve 400 steps per revolution. Also assume we configure the driver to produce a maximum of 300 pulses per second. This means that the maximum speed of the motor will be 0.75 revolutions per second (ie 45 rpm).

If we used 'accel = 10' and 'every = 2' then this means the motor speed will increase by 10 units every 150th of a second (300 pulses per second divided by 2) until it reaches the requested speed. Note that values of 'accel = DRIVE\_SPEED\_MAX' and 'every = 1' will effectively give infinite acceleration.

We will use pins A0 to A3 for the coils and leave the enable unused.

```
BIPOLAR motor = MAKE_BIPOLAR( FALSE, 200, 10, 2, A0, A1, A2, A3, null,
HALF_STEP );
```



## Sensors

Sensors allow your program to sample inputs from the real world: acceleration, humidity, current, voltage, etc as well as 'eyes' that can view the unobstructed distance ahead, colours etc.

They enrich your code and allow your robot to react to its environment.

I have split the various sensors into: types, manufacturers and models.

The 'types' cover areas such as Acceleration, Compass, Distance, Voltage and many more.

The manufacturer and model then allow you to drill down to select the appropriate device.

This library tries to abstract everything by type. Eh? Well we have already mentioned that one type is 'distance'. So as long as all distance sensors return a value in the same units (cm, say) then it means that we can swap one sensor for any other distance sensor without changing our main program. Obviously different distance sensors may vary in accuracy, price, etc. So if your 'cheap' distance sensor isn't accurate enough then swap in a more accurate one - the rest of your code stays the same.

So its kinda plug'n'play.

All sensors have 3 key routines:-

1. An initialisation command which should be called in `appInitHardware`. For some sensors that don't require any initialisation you can get away with not doing this - whereas you will get an error for sensors that do require it but it is missing. So best practise says you should always initialise all of the sensors to avoid errors today or in future releases
2. A read command which will re-sample the input and store it in a local variable.
3. One or more variables associated with the sensor that allow you to access the read values

This may sound complex - why can't I just read a sensor and return its value in one go (as with earlier lib versions)?

For simple sensors only returning a single value then the problem with the 'old way' was that they all had to return the same data type (a restriction of the C/C++ language). We wanted to remove this restriction so that some, like 'pressure', could return a 'float' whilst others would just return a form of integer and thereby reduce the requirement on floating point maths wherever possible as it is: slow and bloats the compiled program.

More complex/advanced sensors can return more than one value. EG a 3 axis accelerometer can return 3 values so reading the sensor to get a single value no longer makes sense.

Note that the variables associated with the sensor mean that you don't need to create your own variables to store the answers - and in fact this is actively discouraged as it increases the overall RAM requirements of your program.



The following sections start with an initial description of the sensor type. Each device will work the same way.

Each device type also has a corresponding 'Dump' routine, eg `compassDump(device)`, `accelerometerDump(device)`, which will output the value of the sensor to the current `rprintf` destination and is useful for debugging.

Acceleration	205
Compass	211
Current	224
Distance	229
Encoder	242
GPS	252
Gyro	257
Humidity	263
IMU	265
Pressure	268
Temperature	270
Voltage	273



## Sensors/Acceleration

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.

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 carefull some of them require a 3.3v supply so connecting them to the usual 5v will fry them!

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.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Acceleration/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> accel = MAKE_<DEVICE>(F0,F1,F2);
```

In your apInitHardware you should initialise the device:-

```
accelerometerInit(accel);
```

Then in uour main loop you can read the sensor using:-

```
accelerometerRead(accel);
```

Each axis can then be read independently into a variable of type 'ACCEL\_TYPE':-



```
ACCEL_TYPE x = accel.accelerometer.x_axis_mG;  
ACCEL_TYPE y = accel.accelerometer.y_axis_mG;  
ACCEL_TYPE z = accel.accelerometer.z_axis_mG;
```

Or dumped to the current rprintf destination using:-

```
accelerationDump(accel);
```

Whilst the above code will show changing values then 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. This is done as follows:-

1. Use the above code to 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 `applInitHardware`: after you have initialised the sensor then calibrate each axis by passing in the minimum and maximum values that you noted down. ie to calibrate the x axis use

```
accelerometerCalibrateX(device, -935,1361);
```

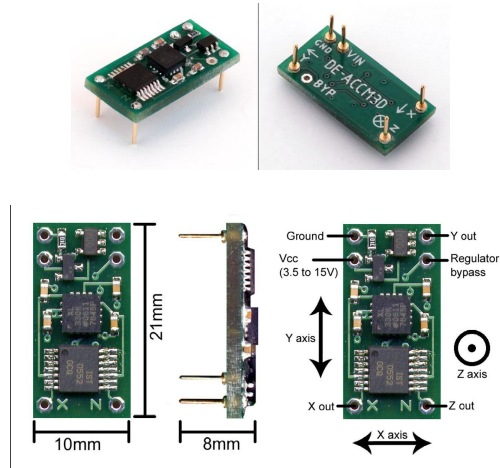
Now re-run your tests and you should find that the calibrated axes will return values between -1000 and +1000.

DimensionEngineering/ACCM3D2.h	207
AnalogDevices/ADXL335.h	208
AnalogDevices/ADXL345.h	209
Freescale/MMA7260QT.h	210



## Sensors/Acceleration/DimensionEngineering/ACCM3D2.h

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.

Each axis requires an ADC pin and so the device can be declared by specifying the X, Y and Z input pins. For example on the Axon this could be done as follows:-

```
#include "DimensionEngineering/ACCM3D2.h"
DE_ACCM3D2 accel = MAKE_DE_ACCM3D2(ADC0,ADC1,ADC2);
```

Having caused the sensor to be read using:-

```
accelerometerRead(accel);
```

Each axis can then be read independently:-

```
ACCEL_TYPE x = accel.accelerometer.x_axis_mG;
ACCEL_TYPE y = accel.accelerometer.y_axis_mG;
ACCEL_TYPE z = accel.accelerometer.z_axis_mG;
```

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



## Sensors/Acceleration/AnalogDevices/ADXL335.h

The ADXL335 is a 3 axis accelerometer from Analog Devices capable of measuring  $\pm 3g$ .

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

Supplier: [http://www.sparkfun.com/commerce/product\\_info.php?products\\_id=9268](http://www.sparkfun.com/commerce/product_info.php?products_id=9268)

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

Each axis requires an ADC pin and so the device can be declared by specifying the X, Y and Z input pins. For example on the Axon this could be done as follows:-

```
#include "AnalogDevices/ADXL335.h"
ADXL335 accel = MAKE_ADXL335(ADC0,ADC1,ADC2);
```

Having caused the sensor to be read using:-

```
accelerometerRead(accel);
```

Each axis can then be read independently:-

```
ACCEL_TYPE x = accel.accelerometer.x_axis_mG;
ACCEL_TYPE y = accel.accelerometer.y_axis_mG;
ACCEL_TYPE z = accel.accelerometer.z_axis_mG;
```

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





## Sensors/Acceleration/AnalogDevices/ADXL345.h

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

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.

The device can be created, specifying the I2C address, as follows:-

```
#include "AnalogDevices/ADXL345.h"
ADXL345 accel = MAKE_ADXL345(0xA6);
```

Having caused the sensor to be read using:-

```
accelerometerRead(accel);
```

Each axis can then be read independently:-

```
ACCEL_TYPE x = accel.accelerometer.x_axis_mG;
ACCEL_TYPE y = accel.accelerometer.y_axis_mG;
ACCEL_TYPE z = accel.accelerometer.z_axis_mG;
```

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

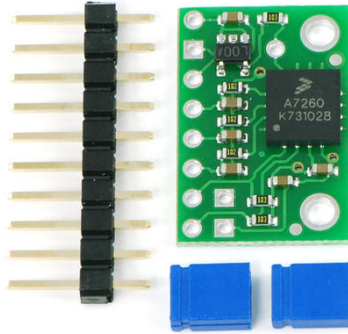


## Sensors/Acceleration/Freescale/MMA7260QT.h

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

Data sheet: [http://www.pololu.com/file/download/MMA7260QT.pdf?file\\_id=0J87](http://www.pololu.com/file/download/MMA7260QT.pdf?file_id=0J87)

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



The device requires one ADC pin for each axis and this library provides four different constructors - one for each sensitivity range:

```
MAKE_FS_MMA7260QT_1g5(channelX, channelY, channelZ)
MAKE_FS_MMA7260QT_2g(channelX, channelY, channelZ)
MAKE_FS_MMA7260QT_4g(channelX, channelY, channelZ)
MAKE_FS_MMA7260QT_6g(channelX, channelY, channelZ)
```

For example on the Axon you could define a sensor using the 2g range as follows:-

```
#include "Freescale/MMA7260QT.h"
FS_MMA7260QT accel = MAKE_FS_MMA7260QT_2g(ADC0,ADC1,ADC2);
```

Note that you will still need to set the jumpers on the device to select the 2g range.

Having caused the sensor to be read using:-

```
accelerometerRead(accel);
```

Each axis can then be read independently:-

```
ACCEL_TYPE x = accel.accelerometer.x_axis_mG;
ACCEL_TYPE y = accel.accelerometer.y_axis_mG;
ACCEL_TYPE z = accel.accelerometer.z_axis_mG;
```

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



## Sensors/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 device 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'.

All reading from this library are in degree 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 but you will need to check the individual data sheets to see what power supply they need.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Acceleration/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myCompass = MAKE_<DEVICE>(i2cAddress or ADC pins);
```

In your appInitHardware you should initialise the device:-

```
compassInit(myCompass);
```

Then in your main loop you can read the sensor using:-

```
compassRead(myCompass);
```

Each value can then be read independently into a variable of type 'COMPASS\_TYPE':-

```
COMPASS_TYPE bearing = myCompass.compass.bearingDegrees;
COMPASS_TYPE roll    = myCompass.compass.rollDegrees;
COMPASS_TYPE pitch   = myCompass.compass.pitchDegrees;
```

Or dumped to the current rprintf destination using:-

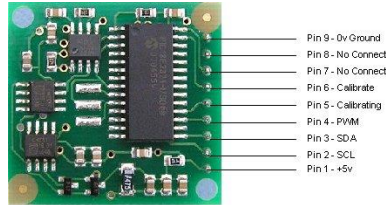
```
compassDump(myCompass);
```

Devantech/CMPS03.h	212
Devantech/CMPS09.h	214
Honeywell/HMC6343.h	215
Honeywell/HMC6352.h	218
Honeywell/HMC5843.h	221



## Sensors/Compass/Devantech/CMPS03.h

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.

The device requires a 5V regulated supply so on the Axon you will need to use one of the ADC header supply pins.

Example for the Axon that uses Pin 4 of the device connected to F4 on the Axon :-

```
#include "sys/Axon.h"
#include "rprintf.h"
#include "Sensors/Compass/Devantech/CMPS03.h"
CMPS03 cmps03 = MAKE_CMPS03(F4);
void appInitHardware(void){
    //setup UART
    uartInit(UART1, 9600);
    rprintfInit(&uart1SendByte);
    // Initialise the compass
    compassInit(cmps03);
}
```

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
```

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    compassRead(cmps03);
    rprintf("CMPS03 = %d\n", cmps03.compass.bearingDegrees);
    return 0;
}
```

To use the I2C bus instead then you only need to change the one line of code where the compass is defined to:-



```
CMPS03_I2C cmps03 = MAKE_CMPS03_I2C_At(0xC0);
```

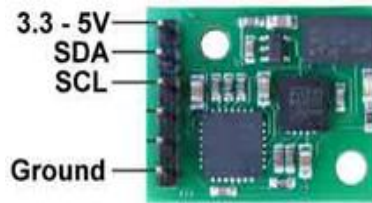
The 0xC0 parameter is the I2C address of the device. The default value is 0xC0 but you can change this value via software to another address. Read the data sheet to find out how. Having created the device you will need to add it to an I2C bus as shown in "*i2cBus.h*" (see page 42)

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.



## Sensors/Compass/Devantech/CMPS09.h

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.

Example :-

```
#include "sys/Axon.h" // Change this to fit your board
#include "rprintf.h"
#include "Sensors/Compass/Devantech/CMPS09.h"
CMPS09_I2C cmps09 = MAKE_CMPS09(); // use default 0xC0 I2C address
void appInitHardware(void){
    //setup UART
    uartInit(UART1, 9600);
    rprintfInit(&uart1SendByte);
    // Initialise the compass
    compassInit(cmps09);
}
```

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
```

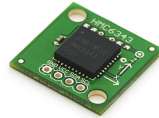
```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    compassRead(cmps09);
    // Print each value one at a time
    rprintf("CMPS09 = %d\n", cmps09.compass.bearingDegrees);
    rprintf("CMPS09 roll = %d\n", cmps09.compass.rollDegrees);
    rprintf("CMPS09 pitch = %d\n", cmps09.compass.pitchDegrees);
    // Or print all in one go
    compassDump(cmps09);
    rprintfCRLF();
    return 0;
}
```

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.



## Sensors/Compass/Honeywell/HMC6343.h

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](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!

To create a device at the default address use:-

```
HMC6343 compass = MAKE_HMC6343();
```

or if you have changed the address to say 0x40 then use:

```
HMC6343 compass = MAKE_HMC6343_At(0x40);
```

Example for the Axon:-



```

#include "sys/Axon.h"
#include "rprintf.h"
#include "Sensors/Compass/Honeywell/HMC6343.h"
HMC6343 compass = MAKE_HMC6343();
void appInitHardware(void){
    //setup UART
    uartInit(UART1, 9600);
    rprintfInit(&uart1SendByte);
    // Initialise compass
    compassInit(compass);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    compassRead(compass);
    rprintf("Compass = %d\n",compass.compass.bearingDegrees);
    return 0;
}

```

## Advanced Function Summary

void	<a href="#">HMC6343_1Hz(HMC6343* compass)</a> Changes the default refresh rate to once per second.
void	<a href="#">HMC6343_5Hz(HMC6343* compass)</a> Changes the default refresh rate to five times per second (ie every 200ms)
void	<a href="#">HMC6343_10Hz(HMC6343* compass)</a> Changes the default refresh rate to ten times per second ie every 100ms

## Advanced Function Detail

### HMC6343\_1Hz

```
void HMC6343_1Hz(HMC6343* compass)
```

Changes the default refresh rate to once per second.

### HMC6343\_5Hz

```
void HMC6343_5Hz(HMC6343* compass)
```

Changes the default refresh rate to five times per second (ie every 200ms)





## **HMC6343\_10Hz**

```
void HMC6343_10Hz(HMC6343* compass)
```

Changes the default refresh rate to ten times per second ie every 100ms



## Sensors/Compass/Honeywell/HMC6352.h

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](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!

To create a device at the default address use:-

```
HMC6352 compass = MAKE_HMC6352();
```

or if you have changed the address to say 0x40 then use:

```
HMC6352 compass = MAKE_HMC6352_At(0x40);
```

Example for the Axon:-



```

#include "sys/Axon.h"
#include "rprintf.h"
#include "Sensors/Compass/Honeywell/HMC6352.h"
HMC6352 compass = MAKE_HMC6352();
void appInitHardware(void){
    //setup UART
    uartInit(UART1, 9600);
    rprintfInit(&uart1SendByte);
    // Initialise compass
    compassInit(compass);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    compassRead(compass);
    rprintf("Compass = %d\n",compass.compass.bearingDegrees);
    return 0;
}

```

## Advanced Function Summary

void	<a href="#">HMC6352_1Hz(HMC6352* compass)</a> Sets the refresh rate to once per second.
void	<a href="#">HMC6352_5Hz(HMC6352* compass)</a> Sets the refresh rate to five timers per second ie every 200ms.
void	<a href="#">HMC6352_10Hz(HMC6352* compass)</a> Sets the refresh rate to ten times per second ie every 100ms.
void	<a href="#">HMC6352_20Hz(HMC6352* compass)</a> Sets the refresh rate to twenty times per second ie every 50ms.

## Advanced Function Detail

### HMC6352\_1Hz

```
void HMC6352_1Hz(HMC6352* compass)
```

Sets the refresh rate to once per second.

### HMC6352\_5Hz

```
void HMC6352_5Hz(HMC6352* compass)
```

Sets the refresh rate to five timers per second ie every 200ms.



### **HMC6352\_10Hz**

```
void HMC6352_10Hz(HMC6352* compass)
```

Sets the refresh rate to ten times per second ie every 100ms.

---

### **HMC6352\_20Hz**

```
void HMC6352_20Hz(HMC6352* compass)
```

Sets the refresh rate to twenty times per second ie every 50ms.



## Sensors/Compass/Honeywell/HMC5843.h

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 address 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.

To create a device use:-

```
HMC5843 compass = MAKE_HMC5843();
```

Example for the Axon:-

```
#include "sys/Axon.h"
#include "rprintf.h"
#include "Sensors/Compass/Honeywell/HMC5843.h"
HMC5843 compass = MAKE_HMC5843();
void appInitHardware(void){
    //setup UART
    uartInit(UART1, 9600);
    rprintfInit(&uart1SendByte);
    // Initialise compass
    compassInit(compass);
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Take a reading
    compassRead(compass);
```

```
// Print out the values in degrees
rprintf("Compass Bearing = %d\n",compass.compass.bearingDegrees);
rprintf("Compass Roll = %d\n",compass.compass.rollDegrees);
rprintf("Compass Pitch = %d\n",compass.compass.pitchDegrees);
```



```
// Or just use the dump command
compassDump(compass);
```

```
// Alternatively the raw x,y,z magnetometer values can be shown
rprintf("Compass Raw: x=%d, y=%d, z=%d\n",
        compass.rawX, compass.rawY, compass.rawZ);
return 0;
}
```

## Advanced Function Summary

void	<a href="#">HMC5843_1Hz(HMC5843* compass)</a> Changes the default refresh rate to once per second.
void	<a href="#">HMC5843_2Hz(HMC5843* compass)</a> Changes the default refresh rate to twice per second (every 500ms).
void	<a href="#">HMC5843_5Hz(HMC5843* compass)</a> Changes the default refresh rate to five times per second (ie every 200ms)
void	<a href="#">HMC5843_10Hz(HMC5843* compass)</a> Changes the default refresh rate to ten times per second ie every 100ms
void	<a href="#">HMC5843_20Hz(HMC5843* compass)</a> Changes the default refresh rate to twenty times per second ie every 50ms
void	<a href="#">HMC5843_50Hz(HMC5843* compass)</a> Changes the default refresh rate to fifty times per second ie every 20ms

## Advanced Function Detail

### HMC5843\_1Hz

```
void HMC5843_1Hz(HMC5843* compass)
```

Changes the default refresh rate to once per second.

### HMC5843\_2Hz

```
void HMC5843_2Hz(HMC5843* compass)
```

Changes the default refresh rate to twice per second (every 500ms).



### **HMC5843\_5Hz**

```
void HMC5843_5Hz(HMC5843* compass)
```

Changes the default refresh rate to five times per second (ie every 200ms)

---

### **HMC5843\_10Hz**

```
void HMC5843_10Hz(HMC5843* compass)
```

Changes the default refresh rate to ten times per second ie every 100ms

---

### **HMC5843\_20Hz**

```
void HMC5843_20Hz(HMC5843* compass)
```

Changes the default refresh rate to twenty times per second ie every 50ms

---

### **HMC5843\_50Hz**

```
void HMC5843_50Hz(HMC5843* compass)
```

Changes the default refresh rate to fifty times per second ie every 20ms

---



## Sensors/Current

Supports current measurement.

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

These devices use one ADC pin for each reading but you will need to check the individual data sheets to see what power supply they need.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Current/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myCurrent = MAKE_<DEVICE>(ADC pin);
```

In your applnitHardware you should initialise the device:-

```
currentInit(myCurrent);
```

Then in your main loop you can read the sensor using:-

```
currentRead(myCurrent);
```

The value can then be read independently into a variable of type 'CURRENT\_TYPE':-

```
CURRENT_TYPE amps = myCurrent.current.amps;
```

Or dumped to the current rprintf destination using:-

```
currentDump(myCurrent);
```

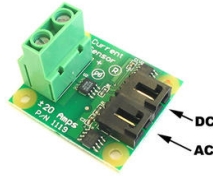
Phidget/AC20A.h	225
Phidget/DC20A.h	226
Phidget/AC50A.h	227
Phidget/DC50A.h	228





## Sensors/Current/Phidget/AC20A.h

Phidget 20A AC current sensor.



This must be connected to an ADC pin. It may be defined using:

```
Phidget_AC20A sensor = MAKE_Phidget_AC20A(ADC0);
```

Where F0 is changed to be the required ADC input pin.

The sensor should be initialised in `applnInitHardware` using:-

```
currentInit(sensor);
```

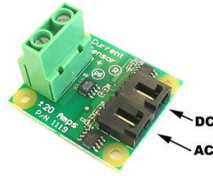
The sensor can then be read using:-

```
currentRead(sensor);
```

and then the current in amps will be in `sensor.current.amps`

## Sensors/Current/Phidget/DC20A.h

Phidget 20A DC current sensor.



This must be connected to an ADC pin. It may be defined using:

```
Phidget_DC20A sensor = MAKE_Phidget_DC20A(ADC0);
```

Where F0 is changed to be the required ADC input pin.

The sensor should be initialised in `applnithHardware` using:-

```
currentInit(sensor);
```

The sensor can then be read using:-

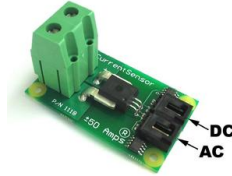
```
currentRead(sensor);
```

and then the current in amps (-20 to +20) will be in `sensor.current.amps`



## Sensors/Current/Phidget/AC50A.h

Phidget 50A AC current sensor.



This must be connected to an ADC pin. It may be defined using:

```
Phidget_AC50A sensor = MAKE_Phidget_AC50A(ADC0);
```

Where F0 is changed to be the required ADC input pin.

The sensor should be initialised in `applnInitHardware` using:-

```
currentInit(sensor);
```

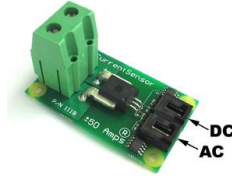
The sensor can then be read using:-

```
currentRead(sensor);
```

and then the current in amps will be in `sensor.current.amps`

## Sensors/Current/Phidget/DC50A.h

Phidget 50A DC current sensor.



This must be connected to an ADC pin. It may be defined using:

```
Phidget_DC50A sensor = MAKE_Phidget_DC50A(ADC0);
```

Where F0 is changed to be the required ADC input pin.

The sensor should be initialised in `applnithHardware` using:-

```
currentInit(sensor);
```

The sensor can then be read using:-

```
currentRead(sensor);
```

and then the current in amps, -50 to +50, will be in `sensor.current.amps`



## Sensors/Distance

Supports measurement of distances.

All reading from this library are in 'cm'.

What do they do? They measure the distance from the device to an obstruction in front of it.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

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 to pass through a beam of light' - 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.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-



```
// Include the relevant sensor file
#include "Sensors/Distance/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myDistance = MAKE_<DEVICE>(ADC pin);
```

In your applnitHardware you should initialise the device:-

```
distanceInit(myDistance);
```

Then in your main loop you can read the sensor using:-

```
distanceRead(myDistance);
```

The value can then be read independently into a variable of type 'DISTANCE\_TYPE':-

```
DISTANCE_TYPE cm = myDistance.distance.cm;
```

Or dumped to the current rprintf destination using:-

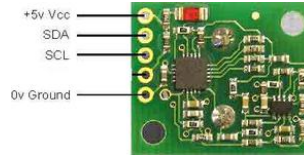
```
distanceDump(myDistance);
```

Devantech/SRF02_Sonar.h	231
Devantech/SRF04_Sonar.h	232
Devantech/SRF05_Sonar.h	233
Devantech/SRF08_Sonar.h	234
Maxbotix/EZ1.h	235
Maxbotix/MB7077.h	236
Ping/PingSonar.h	238
Sharp/GP2.h	239



## Sensors/Distance/Devantech/SRF02\_Sonar.h

Devantech SRF02 sonar.



The sonar uses the I2C bus and also requires a +5v regulated supply. On the Axon this means using the power headers from an ADC pin. 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.

It may be defined using:

```
Devantech_SRF02 sensor = MAKE_Devantech_SRF02();
```

The sensor should be initialised in `aplnitHardware` using:-

```
distanceInit(sensor);
```

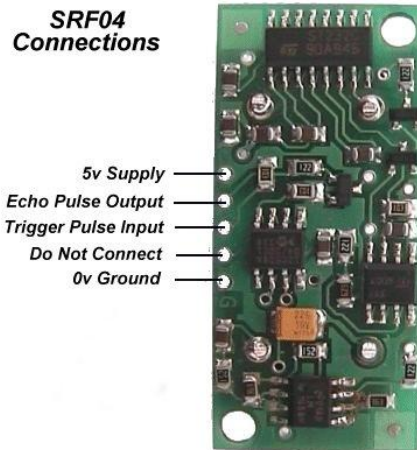
The sensor can then be read using:-

```
distanceRead(sensor);
```

and then the distance in cm will be in `sensor.distance.cm`

## Sensors/Distance/Devantech/SRF04\_Sonar.h

Devantech SRF04 sonar.



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

```
Devantech_SRF04 sensor = MAKE_Devantech_SRF04(F0,F1);
```

Where F0 is the microcontroller pin connected to the 'Trigger Pulse Input' of the device; and F1 is the microcontroller pin connected to the 'Echo Pulse Output' of the device.

The sensor should be initialised in `applnInitHardware` using:-

```
distanceInit(sensor);
```

The sensor can then be read using:-

```
distanceRead(sensor);
```

and then the distance in cm will be in `sensor.distance.cm`

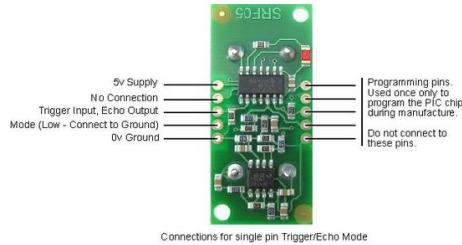
According to the datasheet you should not read the sensor within 50mS of taking the previous reading. Otherwise you will get ghost echoes from the previous reading. This library will automatically detect if that happens and, in the event, will return the previous reading.





## Sensors/Distance/Devantech/SRF05\_Sonar.h

Devantech SRF05 sonar.



The sonar can be connected to any I/O pin but requires a +5v regulated supply. On the Axon this means using an ADC pin. 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. It may be defined using:

```
Devantech_SRF05 sensor = MAKE_Devantech_SRF05(F0);
```

Where F0 is changed to be the required ADC input pin.

The sensor should be initialised in `applnInitHardware` using:-

```
distanceInit(sensor);
```

The sensor can then be read using:-

```
distanceRead(sensor);
```

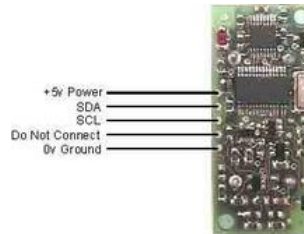
and then the distance in cm will be in `sensor.distance.cm`

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.



## Sensors/Distance/Devantech/SRF08\_Sonar.h

Devantech SRF08 sonar.



The sonar uses the I2C bus and also requires a +5v regulated supply. On the Axon this means using the power headers from an ADC pin. 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.

It may be defined using:

```
Devantech_SRF08 sensor = MAKE_Devantech_SRF08();
```

The sensor should be initialised in `aplnitHardware` using:-

```
distanceInit(sensor);
```

The sensor can then be read using:-

```
distanceRead(sensor);
```

and then the distance in cm will be in `sensor.distance.cm`.

This device also has a Light Dependent Resistor (LDR) that measures the ambient light level. Following a `distanceRead` command then this value can be read as follows:

```
uint8_t lightLevel = sensor.ldr;
```

This value has no associated 'unit of measurement' but can be used to distinguish 'lighter' versus 'darker'.



## Sensors/Distance/Maxbotix/EZ1.h

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. It may be defined using:

```
Maxbotix_EZ1 sensor = MAKE_Maxbotix_EZ1(ADC0);
```

The sensor should be initialised in `aplnitHardware` using:-

```
distanceInit(sensor);
```

The sensor can then be read using:-

```
distanceRead(sensor);
```

and then the distance in cm will be in `sensor.distance.cm`

## Sensors/Distance/Maxbotix/MB7077.h

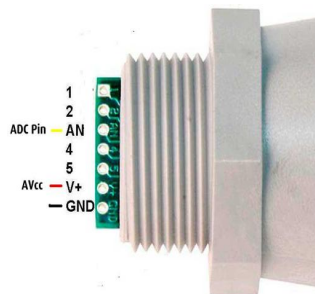
Maxbotix MB7077 sensor (suitable for use under water).



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 in the MAKE command whether the device is under water or not as well as the ADC pin used to read the values:-

```
Maxbotix_MB7077 sonar = MAKE_Maxbotix_MB7077(ADC0, TRUE);
```

The sensor should be initialised in `aplnitHardware` using:-

```
distanceInit(sonar);
```

The sensor can then be read using:-



```
distanceRead(sonar);
```

and then the distance in cm will be in sonar.distance.cm and will be adjusted to be correct for 'in air' or 'in water'.

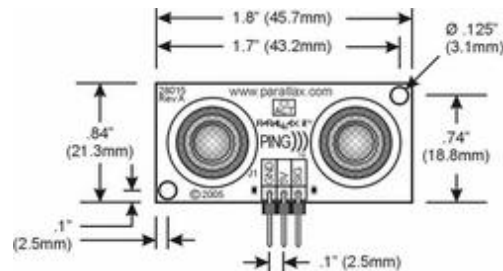
If your robot is amphibious then you can change the 'inWater' member variable at runtime to reflect the current environment for the device:-

```
sonar.inWater = TRUE; // Sonar is under water  
sonar.inWater = FALSE; // Sonar is in air
```



## Sensors/Distance/Ping/PingSonar.h

Parallax PING sonar.



The sonar can be connected to any I/O pin but requires a +5v regulated supply. On the Axon this means using an ADC pin. The sonar can measure distances in the range 2 to 300 cm. It may be defined using:

```
PingSonar sensor = MAKE_PingSonar(F0);
```

Where F0 is changed to be the required ADC input pin.

The sensor should be initialised in `aplnithHardware` using:-

```
distanceInit(sensor);
```

The sensor can then be read using:-

```
distanceRead(sensor);
```

and then the distance in cm will be in `sensor.distance.cm`

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.



## Sensors/Distance/Sharp/GP2.h

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.

Because these devices use 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.

The devices output a voltage and so should be connected 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 !

Regardless of which of these sensors you are using then they can be used in the same way. As an example the GP2D12 can be defined by:-

```
Sharp_GP2D12 sensor = MAKE_Sharp_GP2D12(ADC0);
```

where ADC0 is the ADC pin you are using.

The sensor should be initialised in `aplnitHardware` using:-

```
distanceInit(sensor);
```

The sensor can then be read using:-

```
distanceRead(sensor);
```

and then the distance in cm will be in `sensor.distance.cm`

Since these sensors return a curved graph of 'distance vs voltage' then some floating point maths is involved in transforming the values into a linear graph of cm. This means you will need to make sure that the floating point libraries 'libm.a', and 'libc.a' are included in your link step by using the '-lm -lc' options as per the "Getting started with AVRStudio" section.

There is another range of devices called 'wide field'. 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. 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°





Since these devices have more pins then their MAKE command is bigger. For example:

```
Sharp_GP2Y3A002K0F sensor = MAKE_Sharp_GP2Y3A002K0F(ADC0, B0,B1,B2,B3,B4, B5);
```

The first parameter specifies the ADC pin used to read the distance for the currently selected LED.

The next five pins are general output pins used to select which LED is active. Note that only one LED is active at a time.

The final parameter is an output pin used to turn the device on/off. Since we continually poll the device then this pin is normally held high the whole time. If you are short on output pins then specify 'null' for this parameter and connect Vin of the device to Vcc so that it is always active.

The sensor should be initialised in `aplnitHardware` using:-

```
distanceInit(sensor);
```

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. Note that `distanceRead` 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.

Here is an example of how to use it:

```
if(distanceRead(sensor)==TRUE){
    // We have finished reading one LED and it has started reading the next one
    // The value in sensor.pinNo represents the next LED we are now starting to
    read so the
    // 'just-read' LED is the previous one
    if(sensor.pinNo == 0){
        // We have just completed reading all 5 LEDs
        // The individual distance values are in sensor.led_cm[x] where x is 0 to
4
        // The overall 'average' distance is in sensor.distance.cm
    }
}else{
    // The sensor is busy reading
}
```



## Sensors/Encoder

Supports measurement of encoders in ticks.

All reading from this library are in 'ticks'.

What do they do? 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 angle has moved, and the number of ticks the encoder produced 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. 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.

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. More modern designs use encoders for volume control as there is no mechanical contact - so less noise and nothing to 'wear out'.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Encoder/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myEncoder = MAKE_<DEVICE>(...);
```

In your applInitHardware you should initialise the device:-

```
encoderInit(myEncoder);
```

Then in your main loop you can read the sensor using:-

```
encoderRead(myEncoder);
```

The value can then be read independently into a variable of type 'ENCODER\_TYPE':-

```
ENCODER_TYPE ticks = myEncoder.encoder.value;
```

Or dumped to the current rprintf destination using:-



```
encoderDump(myEncoder);
```

## Encoder interpolation

As of WebbotLib Version 1.25 the `encoderRead` function can, if 'interpolation' is enabled, return two additional pieces of information which, as per the example above, can be accessed using:-

```
TICK_COUNT t1 = myEncoder.encoder.timeSinceLastTick;  
TICK_COUNT t2 = myEncoder.encoder.durationOfLastTick;
```

The 'timeSinceLastTick' is the number of  $\mu\text{S}$  since we last received a tick; and `durationOfLastTick` is the number of  $\mu\text{S}$  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  $1000\mu\text{S}$  and the encoder is giving 200 ticks per revolution. Then a full revolution would currently require  $1000 \times 200 = 200,000\mu\text{S} = 0.2$  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  $1000\mu\text{S}$  and the 'timeSinceLastTick' is  $500\mu\text{S}$  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 via the `MAKE` command and also at runtime by setting the `myEncoder.encoder.interpolate` variable to either `TRUE` or `FALSE`.

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.

Generic/quadrature.h  
Generic/fastquad.h

244  
248



## Sensors/Encoder/Generic/quadrature.h

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 have find your are missing ticks because you have a high resolution encoder then you may want to consider using the fastquad.h routines 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 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.

Like all encoders we can read the encoder using the encoderRead function to store the current value of the counter. We can then access this value through the encoder.value property.

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 encoderSubtract function to reduce the value. The reason being that otherwise we may 'miss' some pulses. Performing:

```
// read the quadrature encoder
encoderRead(quadrature);
// get the value from the last read
ENCODER_TYPE ticks = quadrature.encoder.value;
// reduce the value by the same amount
encoderSubtract(quadrature, 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:  $16 \times 2,000 = 32,000$  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:  $16 \times 1,300 = 20,800$  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!

## Standard Function Summary

	<a href="#"><u>MAKE_GENERIC_QUADRATURE(const IOPin* channelA,const IOPin* channelB,boolean inverted,uint16_t numStripes,boolean interpolate)</u></a> Create a new quadrature encoder sensor giving 2 ticks per encoder stripe.
	<a href="#"><u>MAKE_GENERIC_QUADRATUREx2(const IOPin* channelA,const IOPin* channelB,boolean inverted,uint16_t numStripes,boolean interpolate)</u></a> Create a new quadrature encoder sensor giving 4 ticks per encoder stripe.
	<a href="#"><u>encoderSubtract(ENCODER* encoder, ENCODER_TYPE count)</u></a> Subtract a given value from the encoder counter.
uint16_t	<a href="#"><u>encoderTicksPerRevolution(const QUADRATURE* encoder)</u></a> Returns the number of counter ticks generated for each 360 degree revolution.

## Standard Function Detail

### MAKE\_GENERIC\_QUADRATURE

```
MAKE_GENERIC_QUADRATURE(const IOPin* channelA,const IOPin*
channelB,boolean inverted,uint16_t numStripes,boolean interpolate)
```

Create a new quadrature encoder sensor giving 2 ticks per encoder stripe.

The channelA parameter specifies the IOPin to use for channel A and must support hardware pin change interrupts - see "*pinChange.h*" (see page 82) .

The channelB parameters specifies the IOPin to use for channel B and may be any IOPin.



The inverted parameter is similar in concept to the idea of inverted motors or sensors. If you have encoders on the motors of a differential drive robot then, when going forward, one encoder would count up and the other would count down (as the wheels are rotating in different directions). This parameter allows you to 'flip' one of the encoders so that they both count upwards.

The numStripes parameter allows you to specify how many stripes there are on the encoder. This is purely specified for use by the encoderTicksPerRevolution function so that your main code can convert the counter value into an angular rotation.

The final parameter specifies whether interpolation should be turned on or off by default.

Example:

```
#include "Sensors/Encoder/Generic/quadrature.h"
QUADRATURE quad1 = MAKE_QUADRATURE(B4,B5,FALSE,32, FALSE);
// then in your main code you can read the encoder as follows:-
encoderRead(quad1);
ENCODER_TYPE ticks = quad1.encoder.value;
```

---

## MAKE\_GENERIC\_QUADRATUREx2

MAKE\_GENERIC\_QUADRATUREx2(const IOPin\* channelA,const IOPin\* channelB,boolean inverted,uint16\_t numStripes,boolean interpolate)

Create a new quadrature encoder sensor giving 4 ticks per encoder stripe.

The channelA parameter specifies the IOPin to use for channel A and must support hardware pin change interrupts. See "*pinChange.h*" (see page 82)

The channelB parameter specifies the IOPin to use for channel B and must support hardware pin change interrupts. See "*pinChange.h*" (see page 82)

The inverted parameter is similar in concept to the idea of inverted motors or sensors. If you have encoders on the motors of a differential drive robot then, when going forward, one encoder would count up and the other would count down (as the wheels are rotating in different directions). This parameter allows you to 'flip' one of the encoders so that they both count upwards.

The numStripes parameter allows you to specify how many stripes there are on the encoder. This is purely specified for use by the encoderTicksPerRevolution function so that your main code can convert the counter value into an angular rotation.

The final parameter specifies whether interpolation should be turned on or off by default.

Example:



```
#include "Sensors/Encoder/Generic/quadrature.h"
QUADRATURE quad1 = MAKE_QUADRATUREx2(B4,B5,FALSE,32,FALSE);
// then in your main code you can read the encoder as follows:-
encoderRead(quad1);
ENCODER_TYPE ticks = quad1.encoder.value;
```

---

## **encoderSubtract**

`encoderSubtract(ENCODER* encoder, ENCODER_TYPE count)`

Subtract a given value from the encoder counter.

---

## **encoderTicksPerRevolution**

`uint16_t encoderTicksPerRevolution(const QUADRATURE* encoder)`

Returns the number of counter ticks generated for each 360 degree revolution.



## Sensors/Encoder/Generic/fastquad.h

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 using the `MAKE_GENERIC_FAST_QUADRATUREx2` constructor which 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 `encoderRead` function to store the current value of the counter. We can then access this value through the `encoder.value` property.

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 `encoderSubtract` function to reduce the value. The reason being that otherwise we may 'miss' some pulses. Performing:

```
// read the quadrature encoder
encoderRead(quadrature);
// get the value from the last read
ENCODER_TYPE ticks = quadrature.encoder.value;
// reduce the value by the same amount
encoderSubtract(quadrature, 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:  $16 \times 9,000 = 144,000$  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:  $16 \times 2,900 = 46,400$  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!

### Standard Function Summary

	<a href="#"><u>MAKE_GENERIC_FAST_QUADRATURE(INTaaa,const IOPin* channelB,boolean inverted,uint16_t numStripes,boolean interpolate)</u></a> Create a new fast quadrature encoder sensor giving two ticks per stripe.
	<a href="#"><u>MAKE_GENERIC_FAST_QUADRATUREx2(INTaaa,INTbbb,boolean inverted,uint16_t numStripes,boolean interpolate)</u></a> Create a new fast quadrature encoder sensor giving 4 ticks per stripe.
	<a href="#"><u>encoderSubtract(ENCODER* encoder, ENCODER_TYPE count)</u></a> Subtract a given value from the encoder counter.
uint16_t	<a href="#"><u>encoderTicksPerRevolution(const QUADRATURE* encoder)</u></a> Returns the number of counter ticks generated for each 360 degree revolution.

### Standard Function Detail

#### MAKE\_GENERIC\_FAST\_QUADRATURE

```
MAKE_GENERIC_FAST_QUADRATURE(INTaaa,const IOPin* channelB,boolean
inverted,uint16_t numStripes,boolean interpolate)
```

Create a new fast quadrature encoder sensor giving two ticks per stripe.



The INTaaa parameter specifies the INT pin to use for channel A eg INT0, INT1 etc. The use of an invalid value will result in a compile/link error.

The channelB parameter specifies the IOPin to use for channel B and may be any IOPin.

The inverted parameter is similar in concept to the idea of inverted motors or sensors. If you have encoders on the motors of a differential drive robot then, when going forward, one encoder would count up and the other would count down (as the wheels are rotating in different directions). This parameter allows you to 'flip' one of the encoders so that they both count upward.

The numStripes parameter allows you to specify how many stripes there are on the encoder. This is purely specified for use by the encoderTicksPerRevolution function so that your main code can convert the counter value into an angular rotation.

The final parameter specifies whether interpolation should be turned on or off by default.

Example:

```
#include "Sensors/Encoder/Generic/fastquad.h"
FAST_QUADRATURE quad1 = MAKE_GENERIC_FAST_QUADRATURE( INT0, B5, FALSE, 32,
FALSE);
// then in your main code you can read the encoder as follows:-
encoderRead(quad1);
ENCODER_TYPE ticks = quad1.encoder.value;
```

---

## MAKE\_GENERIC\_FAST\_QUADRATUREx2

MAKE\_GENERIC\_FAST\_QUADRATUREx2( INTaaa, INTbbb, boolean inverted, uint16\_t numStripes, boolean interpolate)

Create a new fast quadrature encoder sensor giving 4 ticks per stripe.

The INTaaa parameter specifies the INT pin to use for Channel A eg INT0, INT1 etc. The use of an invalid value will result in a compile/link error.

The INTbbb parameter specifies the INT pin to use for Channel B eg INT0, INT1 etc. The use of an invalid value will result in a compile/link error.

The inverted parameter is similar in concept to the idea of inverted motors or sensors. If you have encoders on the motors of a differential drive robot then, when going forward, one encoder would count up and the other would count down (as the wheels are rotating in different directions). This parameter allows you to 'flip' one of the encoders so that they both count upwards.

The numStripes parameter allows you to specify how many stripes there are on the encoder. This is purely specified for use by the encoderTicksPerRevolution function so that your main code can convert the counter value into an angular rotation.



The final parameter specifies whether interpolation should be turned on or off by default.

Example:

```
#include "Sensors/Encoder/Generic/fastquad.h"
FAST_QUADRATUREx2 quad1 =
MAKE_GENERIC_FAST_QUADRATUREx2(INT0,INT1,FALSE,32,FALSE);
// then in your main code you can read the encoder as follows:-
encoderRead(quad1);
ENCODER_TYPE ticks = quad1.encoder.value;
```

---

## **encoderSubtract**

`encoderSubtract(ENCODER* encoder, ENCODER_TYPE count)`

Subtract a given value from the encoder counter.

---

## **encoderTicksPerRevolution**

`uint16_t encoderTicksPerRevolution(const QUADRATURE* encoder)`

Returns the number of counter ticks generated for each 360 degree revolution.



## Sensors/GPS

gps.h	253
NMEA/gpsNMEA.h	254



## **Sensors/GPS/gps.h**

Defines a data structure for holding the information common to most GPS receivers.

See NMEA/gpsNMEA.h for an implementation based on the common NMEA format messages.



## Sensors/GPS/NMEA/gpsNMEA.h

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. You must define your UART to have an input buffer so that any characters received whilst your program is busy are not lost.

Here is an example for a generic ATmega168 board that uses a software UART:-

```
#include "sys/atmega168.h"
#include "uartsw.h"
#include "NMEA/gpsNMEA.h"
```

```
// Create the software UART receive buffer
unsigned char rxArr[10];
cBuffer rxBuf = MAKE_BUFFER(rxArr);
```

```
// Create the software uart to listen via Timer Capture Input (ie pin B0)
SW_UART swUART = MAKE_SW_UART_BUFFERED( &rxBuf, null, TIMER1, B0, null, FALSE);
```

```
// Create the gps and connect via the software UART at 4800 baud
GPS_NMEA gps = MAKE_GPS_NMEA(&swUART, 4800);
```

```
// This routine is called once only and allows you to do any initialisation
// Dont use any 'clock' functions here - use 'delay' functions instead
void appInitHardware(void){
}
```

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}
```



```
// This routine is called repeatedly - its your main loop
TICK_CONTROL appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Call gpsNMEAprocess frequently - to interpret any current msg
    gpsNMEAprocess(&gps);
    // Test if we have received a msg and we have a valid satellite signal
    if(gps.info.valid){
        // We have got some valid gps data.
    }
    return 0; // no delay - go at full speed
}
```

## Standard Function Summary

GPS\_NMEA

[MAKE\\_GPS\\_NMEA\(uart, baudrate\)](#)

Creates a GPS\_NMEA sensor.

boolean

[gpsNMEAprocess\(GPS\\_NMEA\\* gps\)](#)

Interprets any incoming message from the GPS receiver.

## Standard Function Detail

### MAKE\_GPS\_NMEA

GPS\_NMEA MAKE\_GPS\_NMEA(uart, baudrate)

Creates a GPS\_NMEA sensor.

The first parameter is the address of a UART and the second parameter is the required baud rate. For example to use hardware UART0 at 4800 baud:

```
GPS_NMEA gps = MAKE_GPS_NMEA(UART0,4800);
```

### gpsNMEAprocess

boolean gpsNMEAprocess(GPS\_NMEA\* gps)

Interprets any incoming message from the GPS receiver.

You must call this function from within your main loop in order to process any incoming message. Failure to do so will result in the loss of some messages. If you find you are losing messages then increase the size of the UART receive buffer.

This function will return FALSE if no message has been processed during the call. Otherwise it will return TRUE.

When TRUE is returned then you can query which individual fields have been set during the call by examining the settings in the info.changes.values structure. Note that just because a value has been received this doesn't mean that the GPS has got a satellite fix.

For example - if we have created a GPS receiver called 'myGPS' and we are only interested in changes to the longitude or latitude fields then we could write the following:



```
if( gpsNMEAprocess(&myGPS) ){
    // We have received something
    if(myGPS.info.valid){
        // And we have a satellite fix
        if( myGPS.info.changes.values.longitude &&
myGPS.info.changes.values.latitude){
            // Longitude and latitude have been received - so fetch the
values
            double longitude = myGPS.info.logitude;
            double latitude = myGPS.info.latitude;
        }
    }
}
```





## Sensors/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.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Gyro/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> device = MAKE_<DEVICE>(F0,F1);
```

In your applInitHardware you should initialise the device:-

```
gyroInit(device);
```

Then in your main loop you can read the sensor using:-

```
gyroRead(device);
```

Each axis can then be read independently into a variable of type 'ACCEL\_TYPE':-

```
GYRO_TYPE x = device.gyro.x_axis_degrees_per_second;
GYRO_TYPE y = device.gyro.y_axis_degrees_per_second;
```

Or dumped to the current rprintf destination using:-

```
gyroDump(device);
```

InvenSense/IDG300.h	259
InvenSense/IDG500.h	260
ST/LPR530AL.h	261





## Sensors/Gyro/InvenSense/IDG300.h

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](http://www.sparkfun.com/datasheets/Components/IDG-300_Datasheet.pdf)

Requires 2 ADC channels.

Note that the device requires a 3v supply.

To create the device use:-

```
IDG300 idg300 = MAKE_IDG300(ADC0, ADC1);
```

Don't forget that you should initialise the device in `applInitHardware`:-

```
gyroInit(idg300);
```

You can read the device by using:

```
gyroRead(idg300);
```

and then the results are in:-

```
idg300.gyro.x_axis_degrees_per_second  
idg300.gyro.y_axis_degrees_per_second
```



## Sensors/Gyro/InvenSense/IDG500.h

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](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).

To create the device use:-

```
IDG500 idg500 = MAKE_IDG500(ADC0, ADC1, slow);
```

Where slow is TRUE for the SLOW output, or FALSE for the FAST output.

Don't forget that you should initialise the device in `aplnitHardware`:-

```
gyroInit(idg500);
```

You can read the device by using:

```
gyroRead(idg500);
```

and then the results are in:-

```
idg500.gyro.x_axis_degrees_per_second  
idg500.gyro.y_axis_degrees_per_second
```



## Sensors/Gyro/ST/LPR530AL.h

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.

To create the device use:-

```
LPR530AL lpr = MAKE_LPR530AL(ADC0, ADC1, TRUE);
```

Where the last parameter is TRUE for the slower output, or FALSE for the faster output.

Don't forget that you should initialise the device in `applInitHardware`:-

```
gyroInit(lpr);
```

You can read the device by using:

```
gyroRead(lpr);
```

and then the results are in:-

```
lpr.gyro.x_axis_degrees_per_second  
lpr.gyro.y_axis_degrees_per_second
```



## Sensors/Gyro/ST/LY530ALH.h

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).

To create the device use:-

```
LY530ALH yaw = MAKE_LY530ALH(ADC0,slow)
```

Where slow is TRUE for the SLOW output, or FALSE for the FAST output.

Don't forget that you should initialise the device in `applInitHardware`:-

```
gyroInit(yaw);
```

You can read the device by using:

```
gyroRead(yaw);
```

and then the results are in:-

```
yaw.gyro.z_axis_degrees_per_second
```



## Sensors/Humidity

Supports humidity measurement.

All reading from this library are in 'percentage'.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Humidity/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myHumidity = MAKE_<DEVICE>(ADC pin);
```

In your apInitHardware you should initialise the device:-

```
humidityInit(myHumidity);
```

Then in your main loop you can read the sensor using:-

```
humidityRead(myHumidity);
```

The value can then be read independently into a variable of type 'HUMIDITY\_TYPE':-

```
HUMIDITY_TYPE percent = myHumidity.humidity.percent;
```

Or dumped to the current rprintf destination using:-

```
humidityDump(myHumidity);
```

Phidget/Humidity.h

264



## Sensors/Humidity/Phidget/Humidity.h

Phidget humidity sensor.



This must be connected to an ADC pin. It may be defined using:

```
Phidget_Humidity sensor = MAKE_Phidget_Humidity(ADC0);
```

Where ADC0 is changed to be the required ADC input channel.

The sensor should be initialised in `applnInitHardware` using:-

```
humidityInit(sensor);
```

The sensor can then be read using:-

```
humidityRead(sensor);
```

and then the humidity in percent will be in `sensor.humidity.percent`





## Sensors/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.

In your `aplnitHardware` you should initialise the sensor:-

```
imuInit(device);
```

Then in your main loop you can read the sensor using:-

```
imuRead(device);
```

Once read then the values are available in the following member variables:-

```
ACCEL_TYPE xa = device.imu.x_axis_mG;  
ACCEL_TYPE ya = device.imu.y_axis_mG;  
ACCEL_TYPE za = device.imu.z_axis_mG;  
GYRO_TYPE xr = device.imu.x_axis_degrees_per_second;  
GYRO_TYPE yr = device.imu.y_axis_degrees_per_second;  
GYRO_TYPE zr = device.imu.z_axis_degrees_per_second;  
COMPASS_TYPE yaw = device.imu.bearingDegrees;  
COMPASS_TYPE roll = device.imu.rollDegrees;  
COMPASS_TYPE pitch = device.imu.pitchDegrees;
```

Or they can all be dumped to the `rprintf` destination using:

```
imuDump(device);
```

Sparkfun/razor.h

266



## Sensors/IMU/Sparkfun/razor.h

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. WebbotLib does not currently support the firmware supplied pre-installed on the board.

As the device is accessed over a uart then the device can be created by specifying a uart and a baud rate (default is 38,400 baud) ie:

```
RAZOR imu = MAKE_RAZOR( UART2, 38400 );
```

Like most sensors it needs to be initialised in `aplnitHardware` using:

```
imuInit(imu);
```

The device can then be read in your main loop using:

```
imuRead(imu);
```

### Standard Function Summary

[razorLED\(const RAZOR\\* razor, boolean on\)](#)

Controls the status LED on the Razor board.



## Standard Function Detail

### razorLED

`razorLED(const RAZOR* razor, boolean on)`

Controls the status LED on the Razor board.

The second parameter allows you to specify if the LED should be turned on or off.



## Sensors/Pressure

Supports pressure measurement in kPa.

The returned values are relative to the sensor reading at power up.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Pressure/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myPressure = MAKE_<DEVICE>(ADC pin);
```

In your apInitHardware you should initialise the device:-

```
pressureInit(myPressure);
```

Then in your main loop you can read the sensor using:-

```
pressureRead(myPressure);
```

The value can then be read independently into a variable of type 'PRESSURE\_TYPE':-

```
PRESSURE_TYPE kPa = myPressure.pressure.kPa;
```

Or dumped to the current rprintf destination using:-

```
pressureDump(myPressure);
```

Motorola/MPX5100a.h

269



## Sensors/Pressure/Motorola/MPX5100a.h

Motorola pressure sensor returning 45mV per kPa.

This must be connected to an ADC pin. It may be defined using:

```
MPX5100A sensor = MAKE_MPX5100A(ADC0);
```

Where ADC0 is changed to be the required ADC input channel.

The sensor should be initialised in `applnInitHardware` using:-

```
pressureInit(sensor);
```

The sensor can then be read using:-

```
pressureRead(sensor);
```

and then the pressure will be in `sensor.pressure.kPa`



## Sensors/Temperature

Supports temperature measurement.

All readings from this library are in 'celsius'.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the #include and the MAKE command used to create the device.

So here is the generic way to work with a <DEVICE> of a given <MAKE> and <MODEL>:-

```
// Include the relevant sensor file
#include "Sensors/Temperature/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myTemperature = MAKE_<DEVICE>(ADC pin);
```

In your apInitHardware you should initialise the device:-

```
temperatureInit(myTemperature);
```

Then in your main loop you can read the sensor using:-

```
temperatureRead(myTemperature);
```

The value can then be read independently into a variable of type 'TEMPERATURE\_TYPE':-

```
TEMPERATURE_TYPE celsius = myTemperature.temperature.celsius;
```

Or dumped to the current rprintf destination using:-

```
temperatureDump(myTemperature);
```

Phidget/Temperature.h  
Devantech/TPA81.h

271  
272



## Sensors/Temperature/Phidget/Temperature.h

Phidget temperature sensor.



This must be connected to an ADC pin and can return temperatures from -40 to +125 degrees celsius. It may be defined using:

```
Phidget_Temperature sensor = MAKE_Phidget_Temperature(ADC0);
```

Where ADC0 is changed to be the required ADC input pin.

The sensor should be initialised in `applnitHardware` using:-

```
temperatureInit(sensor);
```

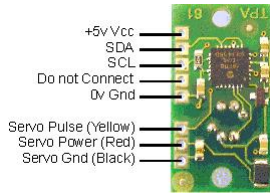
The sensor can then be read using:-

```
temperatureRead(sensor);
```

and then the temperature will be in `sensor.temperature.celsius`

## Sensors/Temperature/Devantech/TPA81.h

Devantech thermal array.



This must be connected to the I2C bus and a 5V power source. Note that we do not use the in-built servo driver.

It may be defined using:

```
Devantech_TPA81 tpa = MAKE_Devantech_TPA81();
```

If you have changed the I2C address of the device from the default value of 0xD0 to, say, 0xD4 then you will need to use:

```
Devantech_TPA81 tpa = MAKE_Devantech_TPA81_At(0xD4);
```

The sensor should be initialised in `aplnitHardware` using:-

```
temperatureInit(tpa);
```

The sensor can then be read using:-

```
temperatureRead(tpa);
```

and then the ambient temperature will be in `tpa.temperature.celsius`

The device also returns 8 individual values which are available in `tpa.sensor[0]` to `tpa.sensor[7]`





## Sensors/Voltage

Supports voltage measurement.

All reading from this library are in 'volts' regardless of whether it is AC or DC. AC voltages are positive whereas DC voltages positive or negative.

These devices use one ADC pin for each reading but you will need to check the individual data sheets to see what power supply they need.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another by only changing the `#include` and the `MAKE` command used to create the device.

So here is the generic way to work with a `<DEVICE>` of a given `<MAKE>` and `<MODEL>`:-

```
// Include the relevant sensor file
#include "Sensors/Voltage/<MAKE>/<MODEL>.h"
// Create the device
<DEVICE> myVoltage = MAKE_<DEVICE>(ADC pin);
```

In your `aplnitHardware` you should initialise the device:-

```
voltageInit(myVoltage);
```

Then in your main loop you can read the sensor using:-

```
voltageRead(myVoltage);
```

The value can then be read independently into a variable of type `'VOLTAGE_TYPE'`:-

```
VOLTAGE_TYPE volts = myVoltage.voltage.volts;
```

Or dumped to the current `rprintf` destination using:-

```
voltageDump(myVoltage);
```

Phidget/30V.h

274



## Sensors/Voltage/Phidget/30V.h

Phidget DC voltage sensor measuring from -30v to +30v.



This must be connected to an ADC pin. It may be defined using:

```
Phidget_30V sensor = MAKE_Phidget_30V(ADC0);
```

Where ADC0 is changed to be the required ADC input channel.

The sensor should be initialised in `applnInitHardware` using:-

```
voltageInit(sensor);
```

The sensor can then be read using:-

```
voltageRead(sensor);
```

and then the voltage will be in `sensor.voltage.volts`



## Displays

Adds support for various standalone display hardware.

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

- Character displays
- Graphic displays

Character displays 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, or 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. 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

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.



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). 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.

The available commands are listed in "*\_display\_common.h*" (see page 278)

Here is a code snippet example of how to use a 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;
}
```

<u>_display_common.h</u>	278
MatrixOrbital/MatrixOrbital.h	283
SparkFun/serLCD.h	284
Generic/HD44780.h	285



## Displays/\_display\_common.h

Defines all of the commands that are common to all displays.

This file is automatically included once you include the header file for the actual display you are using.

### Standard Function Summary

	<a href="#"><u>displayInit(DISPLAY* display)</u></a> Initialise the display prior to use.
DISPLAY_COLUMN	<a href="#"><u>displayColumns(const DISPLAY* display)</u></a> Returns the maximum number of columns, ie characters, across the display.
DISPLAY_LINE	<a href="#"><u>displayLines(const DISPLAY* display)</u></a> Returns the maximum number of character lines down the display.
Writer	<a href="#"><u>displayGetWriter(const DISPLAY* display)</u></a> Returns a Writer that can be used to send rprintf output to the display.
	<a href="#"><u>displayClear(DISPLAY* display)</u></a> Clear the display ie fill it with spaces.
	<a href="#"><u>displayHome(DISPLAY* display)</u></a> Move the cursor to the top left corner of the display.
	<a href="#"><u>displayGoto(DISPLAY* display, DISPLAY_COLUMN x, DISPLAY_LINE y)</u></a> Move the cursor to the specified column and line.
	<a href="#"><u>displayLineWrap(DISPLAY* display,boolean on)</u></a> Turn automatic line wrapping on or off.
	<a href="#"><u>displayAutoScroll(DISPLAY* display,boolean on)</u></a> Turn auto-scrolling on or off.
	<a href="#"><u>displayBacklight(DISPLAY* display,boolean on)</u></a> Turn the display back light on or off (assuming that it has one and that it can be controlled in software).
	<a href="#"><u>displayBrightness(DISPLAY* display,uint8_t percent)</u></a> Set the brightness of the display to a number between 0 (darkest) and 100 (brightest).



## Standard Function Summary

	<a href="#"><code>displayContrast(DISPLAY* display,uint8_t percent)</code></a> Set the contrast of the display to a number between 0 (least) and 100 (most).
<code>uint8_t</code>	<a href="#"><code>displaySendByte(DISPLAY* display,uint8_t byte)</code></a> Send a character, or command byte, to the display.
	<a href="#"><code>displayHorizGraph(DISPLAY* display, DISPLAY_COLUMN x, DISPLAY_LINE y, uint16_t val, uint16_t max, uint8_t count)</code></a> Draws a horizontal bar graph on the display.
	<a href="#"><code>displayVertGraph(DISPLAY* display, DISPLAY_COLUMN x, DISPLAY_LINE y, uint16_t val, uint16_t max, uint8_t count)</code></a> Draws a vertical bar graph on the display.

## Standard Function Detail

### displayInit

`displayInit(DISPLAY* display)`

Initialise the display prior to use.

This should be called from your `aplnitHardware`. If you are using Project Designer then this will be automatically generated for you.

### displayColumns

`DISPLAY_COLUMN displayColumns(const DISPLAY* display)`

Returns the maximum number of columns, ie characters, across the display.

### displayLines

`DISPLAY_LINE displayLines(const DISPLAY* display)`

Returns the maximum number of character lines down the display.

### displayGetWriter

`Writer displayGetWriter(const DISPLAY* display)`

Returns a Writer that can be used to send `rprintf` output to the display.

For example: assuming we have defined a display called 'display' and we want to send the message 'Hello World' to it then we can write:



```
// Redirect rprintf to the display
Writer old = rprintfInit( displayGetWriter(&display) );
rprintf("Hello World\n");
// Restore rprintf to go to its original position
rprintfInit(old);
```

---

## displayClear

`displayClear(DISPLAY* display)`

Clear the display ie fill it with spaces.

---

## displayHome

`displayHome(DISPLAY* display)`

Move the cursor to the top left corner of the display.

---

## displayGoto

`displayGoto(DISPLAY* display, DISPLAY_COLUMN x, DISPLAY_LINE y)`

Move the cursor to the specified column and line.

Both numbers start at 0 so `displayHome` is the same as using this command to go to 0,0.

If you attempt to move the cursor outside of the possible display area then the cursor will wrap around so that it is always within the display area.

---

## displayLineWrap

`displayLineWrap(DISPLAY* display, boolean on)`

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.

---

## displayAutoScroll

`displayAutoScroll(DISPLAY* display, boolean on)`

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 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.

---

## **displayBacklight**

`displayBacklight(DISPLAY* display,boolean on)`

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 backlight is turned off - to save battery power.

---

## **displayBrightness**

`displayBrightness(DISPLAY* display,uint8_t percent)`

Set the brightness of the display to a number between 0 (darkest) and 100 (brightest).

If your display cannot control the brightness via software then this command is ignored.

---

## **displayContrast**

`displayContrast(DISPLAY* display,uint8_t percent)`

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.

---

## **displaySendByte**

`uint8_t displaySendByte(DISPLAY* display,uint8_t byte)`

Send a character, or command byte, to the display.

Normally display output is achieved using `rprintf` - but this command may be used to create a Writer to send data to the display:-

```
MAKE_WRITER(display_put_char){
    return displaySendByte(&display,byte);
}
```

Project Designer will create this code for you automatically.

---



## **displayHorizGraph**

```
displayHorizGraph(DISPLAY* display, DISPLAY_COLUMN x, DISPLAY_LINE y,  
uint16_t val, uint16_t max, uint8_t count)
```

Draws a horizontal bar graph on the display.

The x,y parameters specify the start location of the left side of the graph.

The val parameter specifies the value for the bar and should be between 0 and 'max'

The max parameter specifies the maximum value that can be shown

The count parameter specifies how many characters are to be used across the display for a single bar.

---

## **displayVertGraph**

```
displayVertGraph(DISPLAY* display, DISPLAY_COLUMN x, DISPLAY_LINE y,  
uint16_t val, uint16_t max, uint8_t count)
```

Draws a vertical bar graph on the display.

The x,y parameters specify the start location of the bottom left of the graph.

The val parameter specifies the value for the bar and should be between 0 and 'max'

The max parameter specifies the maximum value that can be shown

The count parameter specifies how many characters are to be used up the display for a single bar.



## Displays/MatrixOrbital/MatrixOrbital.h

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.

To create a new device then define a Writer:

```
Writer(myDisplay_put_char); // Forward definition of myDisplay_put_char - detail
to come
```

Then create the device

```
MATRIX_ORBITAL myDisplay = MAKE_MATRIX_ORBITAL_MOSAL162A(UART0,
19200,&myDisplay_put_char);
```

or

```
MATRIX_ORBITAL myDisplay = MAKE_MATRIX_ORBITAL_MOSAL202A(UART0,
19200,&myDisplay_put_char);
```

Now implement the body of the Writer to send a byte to this display:-

```
MAKE_WRITER(myDisplay_put_char){
    return displaySendByte(&myDisplay, byte);
}
```

If that seems complex - then don't worry - the Project Designer application will write all of the above code for you!

Once defined you should call displayInit from apInitHardware and then you can use other display commands as listed in "*\_display\_common.h*" (see page 278)



## Displays/SparkFun/serLCD.h

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.

To create a new device then define a Writer:

```
Writer(myDisplay_put_char); // Forward definition of myDisplay_put_char - detail
to come
```

Then create the device. All of the WebbotLib constructors are the same: they just cope with different numbers of columns and lines eg

```
SPARKFUN_SERLCD myDisplay =
MAKE_SPARKFUN_SERLCD_CC_LL(uart, baud, &myDisplay_put_char)
```

where CC is the number of columns (16 or 20) and LL is the number of lines (2 or 4).

The default baud rate is 9600 but it may be changed.

So for a 16 x 2 line display on UART0 then:

```
SPARKFUN_SERLCD myDisplay = MAKE_SPARKFUN_SERLCD_16_2(UART0, 9600,
&myDisplay_put_char);
```

Now implement the body of the Writer to send a byte to this display:-

```
MAKE_WRITER(myDisplay_put_char){
    return displaySendByte(&myDisplay, byte);
}
```

If that seems complex - then don't worry - the Project Designer application will write all of the above code for you!

Once defined you should call `displayInit` from `applInitHardware` and then you can use other display commands as listed in "`_display_common.h`" (see page 278)



## Displays/Generic/HD44780.h

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.

When using a MAKE command to create one of these devices then we need to specify its dimensions but the first choice is whether to use MAKE\_HD44780\_8PIN or MAKE\_HD44780\_4PIN depending on the number of IO pins available.

Both of these MAKE commands expect you to specify:-

1. The number of character columns across the display (normally 8, 16 or 20)
2. The number of lines down the display (1, 2, or 4)
3. The IO pins to use for RS, RW, and E (normally connected to pins 4,5, and 6 of the LCD)
4. The data I/O pins - either 4 or 8 of them depending on whether you've used MAKE\_HD44780\_4PIN or MAKE\_HD44780\_8PIN
5. Finally the address of a Writer to be used to send data to this device (autogenerated by Project Designer)

Once defined you should call displayInit from applnithHardware and then you can use other display commands as listed in "*\_display\_common.h*" (see page 278)



## 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 banks: 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.h*" (see page 31) 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.

I have come up with an initial programming interface and the AVRcam is the first guinea pig for it because a) it is new to WebbotLib and b) I have someone to test it for me! The Blackfin camera is already supported directly and I will only move it to this new interface once the interface has been proved.



The other reason for having a standard programming interface is try to encourage camera suppliers to support the WebbotLib interface. As the adoption of WebbotLib increases then hopefully this will further encourage suppliers to become compliant. If you are a supplier then contact me via <http://webbot.org.uk>

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 dont 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'.

<u>_camera_common.h</u>	288
AVRcam/avrcam.h	293
Surveyor/blackfin.h	294



## Cameras/\_camera\_common.h

### Standard Function Summary

void	<a href="#"><u>cameraInit(CAMERA * camera)</u></a> Initialise a camera.
uint16_t	<a href="#"><u>cameraXresolution(CAMERA* camera)</u></a> Return the camera x resolution (ie number of pixels across).
uint16_t	<a href="#"><u>cameraYresolution(CAMERA* camera)</u></a> Return the camera y resolution (ie number of pixels down).
uint8_t	<a href="#"><u>cameraNumColorBins(const CAMERA* camera)</u></a> Returns the number of colour bins supported by this model of camera.
boolean	<a href="#"><u>cameraSetBin(CAMERA* camera,uint8 t bin,const COLOR*min, const COLOR*max)</u></a> Sets the value of a given colour bin.
void	<a href="#"><u>cameraSetMinBlobSize(CAMERA* camera, uint32 t minSize)</u></a> Reduces the amount of data returned by the camera by specifying the smallest blob size (width * height) that we are interested in.
uint8_t	<a href="#"><u>cameraGetBlobs(CAMERA* camera,uint8 t bin)</u></a> Return the number of blobs which match the specified colour bin (or all).
const CAMERA_BLOB*	<a href="#"><u>cameraFetchBlob(const CAMERA* camera, uint8 t blobNo)</u></a> Return the given blob from the list.
boolean	<a href="#"><u>cameraGetPixel(CAMERA* camera,uint16 t x, uint16 t y, COLOR * color)</u></a> Returns the colour of a given pixel.
char *	<a href="#"><u>cameraGetVersion(CAMERA* camera)</u></a> Reads the firmware revision number from the camera.

### Standard Function Detail

#### cameraInit

```
void cameraInit(CAMERA * camera)
    Initialise a camera.
```





See the description of your actual camera as to how to MAKE a new one. Let's assume you have created an AVRcam on UART2 using:-

```
AVRCAM camera1 = MAKE_AVRCAM(UART2);
```

Then you can initialise the camera using:-

```
cameraInit( camera1 );
```

---

## cameraXresolution

```
uint16_t cameraXresolution(CAMERA* camera)
```

Return the camera x resolution (ie number of pixels across).

Some cameras allow you to change the resolution - so don't rely on this always returning the same value!

---

## cameraYresolution

```
uint16_t cameraYresolution(CAMERA* camera)
```

Return the camera y resolution (ie number of pixels down).

Some cameras allow you to change the resolution - so don't rely on this always returning the same value!

---

## cameraNumColorBins

```
uint8_t cameraNumColorBins(const CAMERA* camera)
```

Returns the number of colour bins supported by this model of camera.

Note that this figure will NEVER change for a given model but may change from one model to another. A typical value is 8.

---

## cameraSetBin

```
boolean cameraSetBin(CAMERA* camera,uint8_t bin,const COLOR*min, const COLOR*max)
```

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/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.

---



So to set colour bin #2 to cover a range of REDs then we could do:-

```
COLOR min,max; // create two empty variables
colorSetRGB(&min, 128,0,0); // set min to RGB(128,0,0)
colorSetRGB(&max, 255,128,128); // set max to RGB(255,128,128)
cameraSetBin(camera1, 2, &min, &max); // Set the color bin
```

---

## **cameraSetMinBlobSize**

`void cameraSetMinBlobSize(CAMERA* camera, uint32_t minSize)`

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);

Example: if we know that the smallest blob that we are interested in is 30 pixels then we call call:

```
cameraSetMinBlobSize(camera,30);
```

Some cameras support this function themselves and will therefore not even send blobs that are smaller than this. Otherwise WebbotLib will automatically discard any blobs smaller than this. The result, for a WebbotLib developer, is the same. ie "don't worry about it!"

---

## **cameraGetBlobs**

`uint8_t cameraGetBlobs(CAMERA* camera,uint8_t bin)`

Return the number of blobs which match the specified colour bin (or all).

This will ask the camera to return the matching blob areas which match the specified colour bin. The colour bin can either be 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 cameraFetchBlob to iterate through the list of blobs.

Note that the list returned is always sorted into descending rectangle size - so the biggest rectangle is always at the top of the list.

Here's an example:-

```
uint8_t blobs = cameraGetBlobs( camera, 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
        const CAMERA_BLOB* blob = cameraFetchBlob(camera, b);
        // See _camera_common.h to see the fields in a CAMERA_BLOB
    }
}
```



## cameraFetchBlob

`const CAMERA_BLOB* cameraFetchBlob(const CAMERA* camera, uint8_t blobNo)`

Return the given blob from the list.

The return value is a pointer to a blob whose contents cannot be changed.

See `cameraGetBlobs(CAMERA* camera,uint8_t bin)` for an example.

---

## cameraGetPixel

`boolean cameraGetPixel(CAMERA* camera,uint16_t x, uint16_t y, COLOR * color)`

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.

Example to read the pixel at 10,10 :-

```
uint16_t x = 10;
uint16_t y = 10;
COLOR color;
if(cameraGetPixel(camera, x, y, &color)){
    // The color is in 'color' so lets dump it out
    rprintf("Pixel @ %u,%u =",x,y); colorDump(&color);
}else{
    // We had problems talking to the camera
}
```

---

## cameraGetVersion

`char * cameraGetVersion(CAMERA* camera)`

Reads the firmware revision number from the camera.

Note that the returned string is only valid until the next command is issued to the camera at which point it will get over-written. So either print it out straight away or save it somewhere else.

If there is a problem talking to the camera then the value returned will be NULL.

Here is an example to get the version and send it out to the current rprintf destination:-



```
char* strVer = cameraGetVersion(camera);
if(strVer){
    rprintf("Camera version:");
    rprintfStr(strVer);
    rprintfCRLF();
}else{
    rprintf("No reply\n");
}
```



## Cameras/AVRcam/avrcam.h

Support for the AVRcam camera.

This camera seems to have little support and availability nowadays. I am using it as my 'guinea pig' for the WebbotLib standard interface commands shown in "*\_camera\_common.h*" (see page 288) .

As mentioned previously certain commands are handled slowly by a given camera. For the AVRcam then the following commands are slow (ie 4 seconds or so) to complete:-

- cameraGetPixel(CAMERA\* camera,uint16\_t x, uint16\_t y, COLOR \* color)

### Standard Function Summary

	<a href="#">MAKE_AVRcam(UART*)</a>
	Create a new AVRcam camera.

### Standard Function Detail

#### MAKE\_AVRcam

MAKE\_AVRcam ( UART\* )

Create a new AVRcam camera.

Example to create a camera using Uart 0:

```
#include <AVRcam/avrcam.h>
AVRCAM camera = MAKE_AVRcam(UART0);
```



## Cameras/Surveyor/blackfin.h

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.

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 160 by 120 but you can change this either by using the `blackfinSetResolution` command or by adding a definition prior to including `blackfin.h`. For example: to set the default resolution as 640 by 480 then the top of your program should look something like this:

```
#define BLACKFIN_RESOLUTION_DEFAULT BLACKFIN_640_BY_480  
#include "Cameras/Surveyor/blackfin.h"
```

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.h`. If you use a non-YUV colour space then this library will automatically convert it into a YUV colour as and when required.



Here is a complete example for the Axon using UART3 to talk to the camera at 115,200 baud and that sends information to your PC via UART1 at 115,200 baud:

```
// Define a receive buffer from the camera
#define UART3_RX_BUFFER_SIZE 80

// Define a transmit buffer to the PC
#define UART1_TX_BUFFER_SIZE 80

// Set the default resolution to 640 x 480
#define BLACKFIN_RESOLUTION_DEFAULT BLACKFIN_640_BY_480

// Include library files
#include <buffer.h>
#include <sys/Axon.h>
#include <Cameras/Surveyor/blackfin.h>
#include <rprintf.h>

// Define colour bins
#define RED 6
#define BLUE 3
// Create the camera
BLACKFIN_CAMERA camera = MAKE_BLACKFIN_CAMERA(UART3);

// Set the baud rate and then initialise the camera
void appInitHardware(void){
    // Initialise the UART to the Camera at the expected baud rate
    uartInit(UART3, 115200);
    // Initialise the UART to the PC at the expected baud rate
    uartInit(UART1, 115200);
    // Send rprintf to the PC
    rprintfInit(&uart1SendByte);
    // Put the camera into a known state
    blackfinInit(&camera);
}
```



```

TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Define the colour bins
    COLOR min,max;

    // Set RED bin - using YUV values
    colorSetYUV(&min, 50, 73,200);
    colorSetYUV(&max, 78,100,250);
    blackfinSetBinRange(&camera, RED, &min, &max);

    // Set BLUE bin - using RGB values
    colorSetRGB(&min, 0,0,100);
    colorSetRGB(&max, 50,50,255);
    blackfinSetBinRange(&camera, BLUE, &min, &max);
    return 0;
}
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Find RED pixels
    int nBlobs = blackfinDetectBlobs(&camera,RED);
    if(nBlobs>0){
        // Get the first (ie biggest) blob and dump to PC
        const BLACKFIN_BLOB* blob = blackfinFetchBlob(&camera, 0);
        rprintf("Blobs=%d Biggest: Center=",nBlobs);
        rprintfNum(10,4,FALSE,'0',blob->xCenter);
        rprintfChar(',');
        rprintfNum(10,4,FALSE,'0',blob->yCenter);
        rprintf(" Pixels=");rprintfNum(10,10,FALSE,' ',blob->pixels);
        rprintfCRLF();
    }else{
        rprintf("No blobs\n");
    }
    return 0;
}

```

## Standard Function Summary

	<a href="#"><code>blackfinInit(BLACKFIN_CAMERA* camera)</code></a> Initialise the camera.
<code>BLACKFIN_RESOLUTION</code>	<a href="#"><code>blackfinGetResolution(const BLACKFIN_CAMERA* camera)</code></a> Get the current resolution of the camera.
	<a href="#"><code>blackfinSetResolution(BLACKFIN_CAMERA* camera, BLACKFIN_RESOLUTION res)</code></a> Changes the working resolution of the camera.
<code>const COLOR *</code>	<a href="#"><code>blackfinMeanColor(BLACKFIN_CAMERA* camera)</code></a> This will return the mean colour of all the pixels in view.





## Standard Function Summary

	<a href="#">blackfinGetPixel(BLACKFIN_CAMERA* camera, uint16 t x, uint16 t y, COLOR * color)</a> Read the colour of an individual pixel.
	<a href="#">blackfinSetBinRange(BLACKFIN_CAMERA* camera, uint8 t bin, const COLOR* min, const COLOR* max)</a> Sets a colour bin on the camera to the specified range of colours.
uint32_t	<a href="#">blackfinCountPixels(BLACKFIN_CAMERA* camera, uint8 t bin, uint16 t x1, uint16 t x2, uint16 t y1, uint16 t y2)</a> Return the number of pixels in a rectangle that match a given colour bin.
uint8_t	<a href="#">blackfinDetectBlobs(BLACKFIN_CAMERA* camera, uint8 t bin)</a> Returns the number of blobs that match a given colour bin.
const BLACKFIN_BLOB*	<a href="#">blackfinFetchBlob(BLACKFIN_CAMERA* camera, uint8 t blobNo)</a> Returns a specific blob that was found using the last call to blackfinDetectBlobs.

## Standard Function Detail

### blackfinInit

```
blackfinInit(BLACKFIN_CAMERA* camera)
```

Initialise the camera.

This will send commands to the camera to initialise it and **MUST** be called once in your startup code before issuing any other commands to the camera.

### blackfinGetResolution

```
BLACKFIN_RESOLUTION blackfinGetResolution(const BLACKFIN_CAMERA* camera)
```

Get the current resolution of the camera.

This will return one of the following constants:-

```
BLACKFIN_160_BY_120
BLACKFIN_320_BY_240
BLACKFIN_640_BY_480
BLACKFIN_1280_BY_1024
```



## blackfinSetResolution

`blackfinSetResolution(BLACKFIN_CAMERA* camera, BLACKFIN_RESOLUTION res)`

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.

---

## blackfinMeanColor

`const COLOR * blackfinMeanColor(BLACKFIN_CAMERA* camera)`

This will return the mean colour of all the pixels in view.

You can convert this colour into a given colour space, such as RGB or YUV, using the commands in 'colors.h' or if you want to dump it out to your current rprintf destination then you can use 'colorDump'.

---

## blackfinGetPixel

`blackfinGetPixel(BLACKFIN_CAMERA* camera, uint16_t x, uint16_t y, COLOR * color)`

Read the colour of an individual pixel.

The x and y values should be within range for the current screen resolution. ie if the screen resolution is 640x480 then the x value can be in the range 0...639 and the y value 0...479.

The colour is returned in the colour variable you specified. For example:

```
// Create a color variable
COLOR my_color;
// Read pixel at 10,20
blackfinGetPixel(&camera,10,20, &my_color);
// Dump it out using rprintf.
rprintf("The pixel = ");
colorDump(&my_color);
```

---

## blackfinSetBinRange

`blackfinSetBinRange(BLACKFIN_CAMERA* camera, uint8_t bin, const COLOR* min, const COLOR* max)`

Sets a colour bin on the camera to the specified range of colours.

A colour bin can be thought of as a palette of colours. The camera allows you to create 10 of these palettes and they are numbered from 0 to 9. Having defined one, or more, then they can be referenced from other commands by specifying the colour bin number.

---



## **blackfinCountPixels**

```
uint32_t blackfinCountPixels(BLACKFIN_CAMERA* camera, uint8_t bin,  
uint16_t x1, uint16_t x2, uint16_t y1, uint16_t y2)
```

Return the number of pixels in a rectangle that match a given colour bin.

The co-ordinates of the rectangle can be specified in any order you like ie it doesn't matter whether the first 'x' is the left or the right. We sort it out for you.

---

## **blackfinDetectBlobs**

```
uint8_t blackfinDetectBlobs(BLACKFIN_CAMERA* camera, uint8_t bin)
```

Returns the number of blobs that match a given colour bin.

If there are no matches then it will return 0. Otherwise it may return up to 16 different blobs - sorted into order so that the largest one comes first, and the smallest one last.

Each blob can be returned with a call to blackfinFetchBlob.

---

## **blackfinFetchBlob**

```
const BLACKFIN_BLOB* blackfinFetchBlob(BLACKFIN_CAMERA* camera, uint8_t  
blobNo)
```

Returns a specific blob that was found using the last call to blackfinDetectBlobs.

The blob\_no parameter starts at 0. The returned value points to a structure that will be filled in with the following details for the blob:

- the left and right x co-ordinates of the rectangle,
- the top and bottom y co-ordinates of the rectangle,
- the xCenter and yCenter of the rectangle,
- the number of pixels in the rectangle that matched the colour bin



## 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.

sdCard.h	301
spiEEPROM.h	304
i2cEEPROM.h	309
FileSystem/FAT.h	314



```

/ 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

```

1. *Journal of the American Medical Association*, 2000; 283: 2639-2645.

Ok - enough advertising!

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. Unsurprisingly:- if you try doing it then the computer may report all sorts of errors.

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.

Whichever mode you use then you will need to call 'sdCardInit' to initialise the card.

For 'stand alone' format then you are limited to the sdCardRead and sdCardWrite functions described below.

If you would prefer to use 'Computer format' then refer to Storage/FileSystem/FAT.h

## Standard Function Summary

SD_CARD	<a href="#"><u>MAKE SD CARD(const IOPin* select)</u></a> Create an SD memory card device.
boolean	<a href="#"><u>sdCardInit(SD_CARD* card)</u></a> Initialise the SD card.
boolean	<a href="#"><u>sdCardRead(SD_CARD* card,uint32 t absSector,void* dta,uint8 t numSectors)</u></a> Read a 512 byte sector from the card.
boolean	<a href="#"><u>sdCardWrite(SD_CARD* card, uint32 t absSector,const void* dta,uint8 t numSectors)</u></a> Write a 512 byte sector of data to the card.
STORAGE_CLASS*	<a href="#"><u>sdCardGetStorageClass(void)</u></a> Returns the STORAGE_CLASS for SD cards.



## Standard Function Detail

### MAKE\_SD\_CARD

```
SD_CARD MAKE_SD_CARD(const IOPin* select)
```

Create an SD memory card device.

The parameter is the IO pin used to select the device (ie connected to the CS pin of the card).

---

### sdCardInit

```
boolean sdCardInit(SD_CARD* card)
```

Initialise the SD card.

---

### sdCardRead

```
boolean sdCardRead(SD_CARD* card, uint32_t absSector, void* dta, uint8_t numSectors)
```

Read a 512 byte sector from the card.

---

### sdCardWrite

```
boolean sdCardWrite(SD_CARD* card, uint32_t absSector, const void* dta, uint8_t numSectors)
```

Write a 512 byte sector of data to the card.

---

### sdCardGetStorageClass

```
STORAGE_CLASS* sdCardGetStorageClass(void)
```

Returns the STORAGE\_CLASS for SD cards.

This is used by diskInit(DISK \*disk, uint8\_t numBuffers, const STORAGE\_CLASS\* class, void\* device) if you want to use the card as a disk drive.



## Storage/spiEEPROM.h

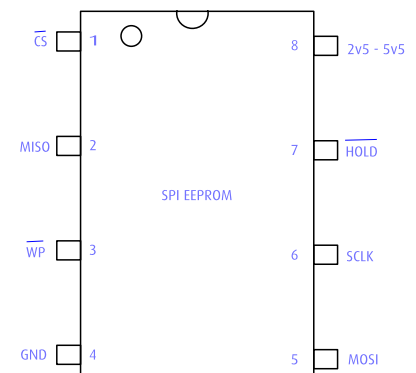
Provides generic support for EEPROM devices accessed over an SPI bus.

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'.

Assuming we want to create a device called 'myEEPROM' with an address size of 2 bytes, a page size of 128 bytes, and a total capacity of 64k and we are using the F1 I/O pin to select the device then:

```
#include "Storage/spiEEPROM.h"
// Create the device
SPI_EEPROM myEEPROM = MAKE_SPI_EEPROM(F1, 2, 128, 64 * KB);
```

```
// Build a list of the devices that are on the same bus - in this case only one
SPI_DEVICE_LIST devices[] = { &myEEPROM._device_ };
```

```
// Create the hardware bus
SPI spiBus = MAKE_SPI(devices);
```

```
// Initialise the driver in my start up code
void appInitHardware(void){
    // Initialise the SPI bus in master mode
    spiBusInit(&spi, TRUE);
}
```

You can now read and write bytes to the EEPROM.

## Standard Function Summary

uint8_t	<a href="#">spiEEPROM_readByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr)</a> Read a single byte from the EEPROM.
void	<a href="#">spiEEPROM_writeByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr, uint8_t data)</a> Write a single byte to the EEPROM.
void	<a href="#">spiEEPROM_readBytes(SPI_EEPROM* eeprom, EEPROM_ADDR addr, void* dest, size_t numBytes)</a> Read a sequence of bytes from the EEPROM.



## Standard Function Summary

void

[`spiEEPROM\_writeBytes\(SPI\_EEPROM\* eeprom, EEPROM\_ADDR addr, const void\* src, size\_t numBytes\)`](#)

Write a sequence of bytes to the EEPROM.

EEPROM\_ADDR

[`spiEEPROM\_totalBytes\(const SPI\_EEPROM\* eeprom\)`](#)

Returns the total number of bytes in the EEPROM.

## Standard Function Detail

### spiEEPROM\_readByte

`uint8_t spiEEPROM_readByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr)`

Read a single byte from the EEPROM.

The parameters specify the EEPROM device to read, and the 32 bit address of the byte in the EEPROM that you want to read.

So to read bytes 120 to 130 from the EEPROM then:

```
for(EEPROM_ADDR addr = 120; addr <= 130; addr++){
    uint8_t byte = spiEEPROM_readByte( &myEEPROM, addr);
}
```

### spiEEPROM\_writeByte

`void spiEEPROM_writeByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr, uint8_t data)`

Write a single byte to the EEPROM.

The parameters specify the EEPROM device, the 32 bit address of the byte in the EEPROM that you want to write, and the value.

So to write a random value to byte 100 in the EEPROM then:

```
uint8_t val = random();
spiEEPROM_writeByte(&myEEPROM, 100, val);
```

Note that writes to the EEPROM are relatively slow. So if you are writing a sequence of bytes to an EEPROM then it is faster to use the `spiEEPROM_writeBytes` command to write them in one go.

### spiEEPROM\_readBytes

`void spiEEPROM_readBytes(SPI_EEPROM* eeprom, EEPROM_ADDR addr, void* dest, size_t numBytes)`

Read a sequence of bytes from the EEPROM.



The parameters specify the EEPROM device to read, the 32 bit address of the first byte in the EEPROM that you want to read, the address where you want to read the data to, and the number of bytes to read.

So to read the 11 bytes starting at 120 from the EEPROM into memory then:

```
// Create a buffer to store the bytes
uint8_t buffer[11];
// Read from the eeprom into buffer
spiEEPROM_readBytes(&myEEPROM, 120, buffer, 11);
```

---

### **spiEEPROM\_writeBytes**

```
void spiEEPROM_writeBytes(SPI_EEPROM* eeprom, EEPROM_ADDR addr, const
void* src, size_t numBytes)
```

Write a sequence of bytes to the EEPROM.

The parameters specify the EEPROM device to write to, the 32 bit address of the first byte in the EEPROM that you want to write to, the address where you want to write the data from, and the number of bytes to write.

So to write the 11 bytes to the address starting at 120 in the EEPROM from memory then:

```
// Create a buffer
uint8_t buffer[11];
// -- Code here puts values in the buffer --
// Write the buffer to EEPROM
spiEEPROM_writeBytes(&myEEPROM, 120, buffer, 11);
```

Often an EEPROM is used to store an array of values where each value is the same size. So lets create a routine to log changes for the ADC inputs. So each 'value' can be stored in the following structure:

```
typedef struct s_adc_log{
    TICK_COUNT time; // The current time
    uint8_t adcs[8]; // 8 adc values
} ADC_LOG;
```

Create a variable to hold one of these 'values':

```
ADC_LOG log;
```

Create a variable to store the current address in the EEPROM we are writing to:

```
EEPROM_ADDR addr; // Will be initialised to 0
```

Create a function to populate the data and return 'TRUE' if the data has changed:



```
boolean getValues(){
    boolean changed = FALSE;
    for(uint8_t n = 0; n < 8; n++){
        uint8_t val = a2dConvert8bit(ADC_NUMBER_TO_CHANNEL(n));
        if( val != log.adcs[n] ){
            // The value has changed
            log.adcs[n] = val;
            changed = TRUE;
        }
    }
    return changed;
}
```

Create a function to write the value to EEPROM:-

```
void write(TICK_COUNT time){
    // Put in the time stamp
    log.time = time;
    // Write to eeprom
    spiEEPROM_writeBytes(&myEEPROM, addr, &log, sizeof(ADC_LOG));
    // bump address for next time
    addr += sizeof(ADC_LOG);
}
```

In your startup code - write out the initial value:

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    getValues();
    write(loopStart);
}
```

In your main loop - only write out the values if they have changed:

```
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    if(getValues()){
        write(loopStart);
    }
}
```

---

## **spiEEPROM\_totalBytes**

EEPROM\_ADDR spiEEPROM\_totalBytes(const SPI\_EEPROM\* eeprom)

Returns the total number of bytes in the EEPROM.

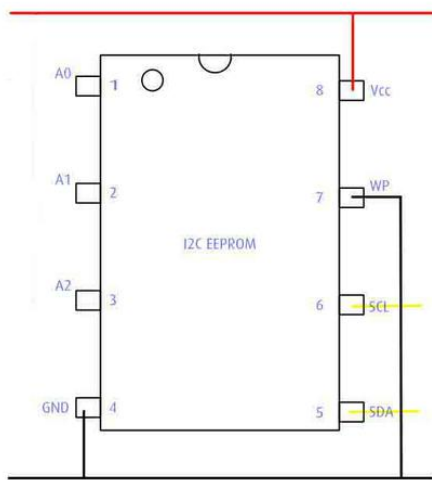


## Storage/i2cEEPROM.h

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'.

This header file has a generic MAKE macro but due to the complexities it also has a MAKE command for several standard ATmel EEPROMs.

Assuming we want to create a device called 'myEEPROM' which is an AT24C1024B and we want to use address 0xC0 then:

```
#include "Storage/i2cEEPROM.h"
// Create the device
I2C_EEPROM myEEPROM = MAKE_I2C_EEPROM_AT24C1024B(FALSE,FALSE); // Address = 0xC0
```

You can now read and write bytes to the EEPROM.

## Standard Function Summary

uint8_t	<a href="#">i2cEEPROM_readByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr)</a> Read a single byte from the EEPROM.
void	<a href="#">i2cEEPROM_writeByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr, uint8_t data)</a> Write a single byte to the EEPROM.
void	<a href="#">i2cEEPROM_readBytes(SPI_EEPROM* eeprom, EEPROM_ADDR addr, void* dest, size_t numBytes)</a> Read a sequence of bytes from the EEPROM.



## Standard Function Summary

void

[`i2cEEPROM\_writeBytes\(SPI\_EEPROM\* eeprom, EEPROM\_ADDR addr, const void\* src, size\_t numBytes\)`](#)

Write a sequence of bytes to the EEPROM.

EEPROM\_ADDR

[`i2cEEPROM\_totalBytes\(const SPI\_EEPROM\* eeprom\)`](#)

Returns the total number of bytes in the EEPROM.

## Standard Function Detail

### `i2cEEPROM_readByte`

`uint8_t i2cEEPROM_readByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr)`

Read a single byte from the EEPROM.

The parameters specify the EEPROM device to read, and the 32 bit address of the byte in the EEPROM that you want to read.

So to read bytes 120 to 130 from the EEPROM then:

```
for(EEPROM_ADDR addr = 120; addr <= 130; addr++){
    uint8_t byte = spiEEPROM_readByte( &myEEPROM, addr);
}
```

### `i2cEEPROM_writeByte`

`void i2cEEPROM_writeByte(SPI_EEPROM* eeprom, EEPROM_ADDR addr, uint8_t data)`

Write a single byte to the EEPROM.

The parameters specify the EEPROM device, the 32 bit address of the byte in the EEPROM that you want to write, and the value.

So to write a random value to byte 100 in the EEPROM then:

```
uint8_t val = random();
spiEEPROM_writeByte(&myEEPROM, 100, val);
```

Note that writes to the EEPROM are relatively slow. So if you are writing a sequence of bytes to an EEPROM then it is faster to use the `spiEEPROM_writeBytes` command to write them in one go.

### `i2cEEPROM_readBytes`

`void i2cEEPROM_readBytes(SPI_EEPROM* eeprom, EEPROM_ADDR addr, void* dest, size_t numBytes)`

Read a sequence of bytes from the EEPROM.



The parameters specify the EEPROM device to read, the 32 bit address of the first byte in the EEPROM that you want to read, the address where you want to read the data to, and the number of bytes to read.

So to read the 11 bytes starting at 120 from the EEPROM into memory then:

```
// Create a buffer to store the bytes
uint8_t buffer[11];
// Read from the eeprom into buffer
spiEEPROM_readBytes(&myEEPROM, 120, buffer, 11);
```

---

### **i2cEEPROM\_writeBytes**

```
void i2cEEPROM_writeBytes(SPI_EEPROM* eeprom, EEPROM_ADDR addr, const
void* src, size_t numBytes)
```

Write a sequence of bytes to the EEPROM.

The parameters specify the EEPROM device to write to, the 32 bit address of the first byte in the EEPROM that you want to write to, the address where you want to write the data from, and the number of bytes to write.

So to write the 11 bytes to the address starting at 120 in the EEPROM from memory then:

```
// Create a buffer
uint8_t buffer[11];
// -- Code here puts values in the buffer --
// Write the buffer to EEPROM
spiEEPROM_writeBytes(&myEEPROM, 120, buffer, 11);
```

Often an EEPROM is used to store an array of values where each value is the same size. So lets create a routine to log changes for the ADC inputs. So each 'value' can be stored in the following structure:

```
typedef struct s_adc_log{
    TICK_COUNT time; // The current time
    uint8_t adcs[8]; // 8 adc values
} ADC_LOG;
```

Create a variable to hold one of these 'values':

```
ADC_LOG log;
```

Create a variable to store the current address in the EEPROM we are writing to:

```
EEPROM_ADDR addr; // Will be initialised to 0
```

Create a function to populate the data and return 'TRUE' if the data has changed:





```
boolean getValues(){
    boolean changed = FALSE;
    for(uint8_t n = 0; n < 8; n++){
        uint8_t val = a2dConvert8bit(ADC_NUMBER_TO_CHANNEL(n));
        if( val != log.adcs[n] ){
            // The value has changed
            log.adcs[n] = val;
            changed = TRUE;
        }
    }
    return changed;
}
```

Create a function to write the value to EEPROM:-

```
void write(TICK_COUNT time){
    // Put in the time stamp
    log.time = time;
    // Write to eeprom
    spiEEPROM_writeBytes(&myEEPROM, addr, &log, sizeof(ADC_LOG));
    // bump address for next time
    addr += sizeof(ADC_LOG);
}
```

In your startup code - write out the initial value:

```
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    getValues();
    write(loopStart);
}
```

In your main loop - only write out the values if they have changed:

```
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    if(getValues()){
        write(loopStart);
    }
}
```

---

## i2cEEPROM\_totalBytes

EEPROM\_ADDR i2cEEPROM\_totalBytes(const SPI\_EEPROM\* eeprom)

Returns the total number of bytes in the EEPROM.



## Storage/FileSystem/FAT.h

Overlays a FAT file system on top of an underlying storage element.

The underlying storage element must be capable of reading / writing 512 byte (sector) blocks.

Prior to use then the storage element must have been formatted.

For an SD card then this may have been done using something like the free download from <http://www.sdcard.org/consumers/formatter/>

The benefit of the FAT system is that the same media can be shared with Windows and other operating systems. So it produces a truly 'hot pluggable hard drive' for removable media. ie an SD card may be 'removable' but an 'EEPROM' chip may not. But why not use the same code for both - SD card for debugging and EEPROM for release - as an example. WebbotLib lets you do that without changes.

The FAT system is cool. But before it can be used then you must connect the underlying device to the FAT file system and this may look a bit complex.

The key is to use the 'disklnit' command and thats a bit fiddly too!

So here is an example for an SD card:-

```
// Choose which pins will simulate the SPI bus - in this case the hardware ones
#define MOSI B2
#define MISO B3
#define SCLK B1
#define CS F0
```

```
// Create the sd card
SD_CARD card = MAKE_SD_CARD(CS);
```

```
// Build a list of the devices that are on the same bus
SPI_DEVICE_LIST devices[] = { &card._device_ };
```

```
// Create the software bus
SPI_SW software_spi = MAKE_SW_SPI(devices, MOSI, MISO, SCLK);
```



```
// Create the disk file system
DISK disk;
```

Then in your appInitHardware:-

```
// Initialise the card
sdCardInit(&card);
```

```
// Try to make it into a hard drive
diskInit(&disk, 1, sdCardGetStorageClass(), &card);
```

## Standard Function Summary

boolean	<a href="#"><u>diskInit(DISK *disk, uint8_t numBuffers, const STORAGE_CLASS* class, void* device)</u></a> Initialise a disk interface.
boolean	<a href="#"><u>diskFlush(const DISK *disk)</u></a> Flush all pending changes to the disk.
uint32_t	<a href="#"><u>diskFreeSpace(const DISK* disk)</u></a> Return the number of available Kb on the disk.
int8_t	<a href="#"><u>diskMkdir(DISK* disk, const char* dirname)</u></a> Create a directory.
boolean	<a href="#"><u>fileFindFirst(const DISK* disk, FILE_ITERATOR *list, const char* dirname)</u></a> Find the first file entry in a directory.
boolean	<a href="#"><u>fileFindNext(FILE_ITERATOR *list)</u></a> Return the next file in the directory.
int8_t	<a href="#"><u>fileOpen(DISK* disk, FILE* file, const char* filename, char mode)</u></a> Open a file.
void	<a href="#"><u>fileClose(FILE* file)</u></a> Close a file.
void	<a href="#"><u>fileFlush(FILE* file)</u></a> Flush any file changes to the disk.
boolean	<a href="#"><u>fileSetPos(FILE *file, uint32_t pos)</u></a> Set the current position in the file for fileReadNext or fileWriteNext.
uint32_t	<a href="#"><u>fileGetPos(const FILE* file)</u></a> Return the current position in the file.



## Standard Function Summary

uint32_t	<a href="#">fileGetSize(const FILE* file)</a> Return the size of the file in bytes.
size_t	<a href="#">fileReadNext(FILE *file, size_t size, void *buf)</a> Perform a sequential read from the current file position.
size_t	<a href="#">fileWriteNext(FILE *file, size_t size, const void *buf)</a> Perform a sequential write to the file.
size_t	<a href="#">fileRead(FILE *file, uint32_t offset, size_t size, void *buf)</a> Perform a random read from the file.
size_t	<a href="#">fileWrite(FILE* file, uint32_t offset, size_t size, const void* buf)</a> Perform a random write into a file.
boolean	<a href="#">fileExists(DISK * disk, const char* filename)</a> Test if a file already exists.
boolean	<a href="#">fileDelete(DISK * disk, const char* filename)</a> Delete a file.
Writer	<a href="#">fileGetWriter(const FILE* file)</a> Return a Writer to allow rprintf output to be redirected to the file.

## Standard Function Detail

### diskInit

`boolean diskInit(DISK *disk, uint8_t numBuffers, const STORAGE_CLASS* class, void* device)`

Initialise a disk interface.

'numBuffers' is the number of 512 byte buffers to be used and must be at least 1. Larger numbers will make this module work faster.

Returns FALSE if failed, else TRUE

### diskFlush

`boolean diskFlush(const DISK *disk)`

Flush all pending changes to the disk.

### diskFreeSpace

`uint32_t diskFreeSpace(const DISK* disk)`

Return the number of available Kb on the disk.



## diskMkdir

`int8_t diskMkdir(DISK* disk, const char* dirname)`

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
```

---

## fileFindFirst

`boolean fileFindFirst(const DISK* disk, FILE_ITERATOR *list, const char* dirname)`

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 file information can be retrieved from the FILE\_ITERATOR. The file iterator is then passed into the file\_findNext call to retrieve the next file.

Example: to list the contents of the root folder

```
FILE_ITERATOR f;
boolean ok;
ok = fileFindFirst(&disk, &f, "/");
while(ok){
    rprintf("%s (%lu bytes)", f.filename, f.fileSize);
    if(f.attribute.flags.isDirectory){
        rprintf(" <dir>");
    }
    rprintf("\n");
    ok = fileFindNext(&f);
}
```

---

## fileFindNext

`boolean fileFindNext(FILE_ITERATOR *list)`

Return the next file in the directory.

Returns FALSE if there are no more files.

See "*fileFindFirst(const DISK\* disk, FILE\_ITERATOR \*list, const char\* dirname)*" (see page 317) for an example.

---



## fileOpen

```
int8_t fileOpen(DISK* disk, FILE* file, const char* filename, char mode)
```

Open a file.

The mode parameter can either be:

```
'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.
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
```

---

## fileClose

```
void fileClose(FILE* file)
```

Close a file.

This will flush any outstanding changes to the disk and then close the file.

---

## fileFlush

```
void fileFlush(FILE* file)
```

Flush any file changes to the disk.

This the equivalent of closing the file and then re-opening it - but is much faster.

In order to reduce the amount of reading from, and writing to, the disk then some data is held in memory. This function will force any changes to the file to be flushed out to the disk.

If your application keeps a file open for writing for a period of time (such as a data logger) then it is wise to call this function every now and then. Otherwise, when you turn off the power then some data may be lost.

---

## fileSetPos

```
boolean fileSetPos(FILE *file, uint32_t pos)
```

Set the current position in the file for fileReadNext or fileWriteNext.



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.

---

### **fileGetPos**

uint32\_t fileGetPos(const FILE\* file)

Return the current position in the file.

---

### **fileGetSize**

uint32\_t fileGetSize(const FILE\* file)

Return the size of the file in bytes.

---

### **fileReadNext**

size\_t fileReadNext(FILE \*file, size\_t size, void \*buf)

Perform a sequential read from the current file position.

The returned value is the number of bytes actually read. This will normally be the same as the number of bytes requested unless you are trying to read beyond the end of the file.

Example: to open an existing file and dump its contents out using rprintf

```
FILE file;
char buffer[80];
if(fileOpen(&disk,&file,"/TEST.TXT",'r')==0){
    size_t bytes;
    while( bytes=fileReadNext(&file,sizeof(buffer),buffer)) > 0 ){
        rprintfStrLen(buffer,0,bytes);
    }
    fileClose(&file);
}
```

### **fileWriteNext**

size\_t fileWriteNext(FILE \*file, size\_t size, const void \*buf)

Perform a sequential write to the file.

This will write to the current position and then update the current position ready for the next call.

The returned value is the number of bytes actually written which will be the same as the specified value unless the disk is full.

---



## **fileRead**

```
size_t fileRead(FILE *file,uint32_t offset, size_t size,void *buf)
```

Perform a random read from the file.

This function ignores the current file position and allows you read from any position within the file.

The returned value is the number of bytes actually read. This will always be the same as the number of bytes requested - unless you are trying to read beyond the end of the file.

---

## **fileWrite**

```
size_t fileWrite(FILE* file,uint32_t offset,size_t size,const void* buf)
```

Perform a random write into a file.

This will ignore the current file position and allow you to write anywhere within the file. The returned value is the number of bytes actually written which will be the same as the specified value unless the disk is full.

---

## **fileExists**

```
boolean fileExists(DISK * disk,const char* filename)
```

Test if a file already exists.

Return TRUE if it does exist, or FALSE if it doesn't.

---

## **fileDelete**

```
boolean fileDelete(DISK * disk,const char* filename)
```

Delete a file.

This will return FALSE if the file doesn't exist or is marked as read only.

---

## **fileGetWriter**

```
Writer fileGetWriter(const FILE* file)
```

Return a Writer to allow rprintf output to be redirected to the file.

This is useful if you are creating a text file and writing out formatted text and numbers.

Example: Write out some data to a file.

---





```
// Assume on entry that rprintf is going to your PC
if(fileExists(&disk,"/TEST.TXT")){
    rprintf("Delete file first\n"); // => PC
    fileDelete(&disk,"/TEST.TXT");
}
```

```
FILE f; // Holds the reference to the open file
rprintf("Write to /TEST.TXT\n"); // => PC
if(fileOpen(&disk,&f,"/TEST.TXT",'w')==0){
    // Redirect output to the file
    Writer old = rprintfInit(fileGetWriter(&f));
    // Write out some stuff
    for(int line = 1; line < 100; line++){
        rprintf("Line %d: ABCDEFGHIJKLMNOPQRSTUVWXYZ\n",line); // => file
    }
    // Close the file
    fileClose(&f);
    // Restore rprintf to go to its original location
    rprintfInit(old);
    rprintf("Done\n"); // => PC
}else{
    rprintf("Couldn't create file\n"); // => PC
}
```



## Servos

Adds support for serial and I2C servo controllers as well as other intelligent servos such as the AX12.

If you are using a standard servo requiring a pulse every 20ms that is plugged directly onto your own board then see "*servos.h*" (see page 98)

Dynamixel/AX12.h	323
Devantech/SD21.h	328
Generic/Serial Servo Controller	330



## Servos/Dynamixel/AX12.h

Adds support for the Dynamixel AX-12 servo.

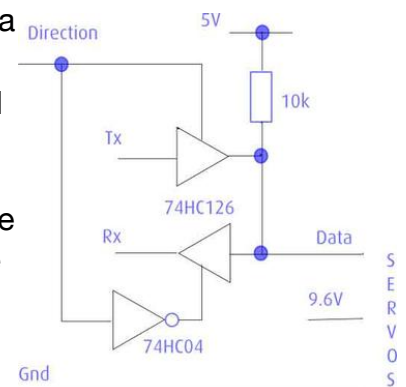
See <http://www.trossenrobotics.com/dynamixel-ax-12-robot-actuator.aspx> for more info and a data sheet.



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.

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 '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.

### Standard Function Summary

	<a href="#"><u>MAKE_DYNAMIXEL_AX12(boolean inverted, uint8_t id, boolean continuous)</u></a> A macro to create a new AX12 servo.
void	<a href="#"><u>ax12Init(DYNAMIXEL_AX12_DRIVER* driver, BAUD_RATE baud)</u></a> Initialise the driver to a given baud rate.
uint16_t	<a href="#"><u>ax12GetInfo(DYNAMIXEL_AX12* servo)</u></a> Read the current settings from a servo.
void	<a href="#"><u>ax12Dump(DYNAMIXEL_AX12* servo)</u></a> Dump the current status of a given servo to the rprintf destination.
void	<a href="#"><u>ax12DumpAll(const DYNAMIXEL_AX12_DRIVER* driver)</u></a> Dump the status of ALL of the servos to the rprintf destination.



## Standard Function Summary

void

[ax12Begin\(DYNAMIXEL\\_AX12\\_DRIVER\\* driver\)](#)

Marks the start of a group of commands.

void

[ax12End\(DYNAMIXEL\\_AX12\\_DRIVER\\* driver\)](#)

Marks the end of a command group.

## Advanced Function Summary

void

[ax12SetXXXX\(servo, val\)](#)

Macros to allow advanced users to set a value in the servos Control Table.

void

[ax12Send\(const DYNAMIXEL\\_AX12\\_DRIVER\\* driver, uint8\\_t id, size\\_t len, uint8\\_t\\* data\)](#)

Send a sequence of bytes to the AX12.

## Standard Function Detail

### MAKE\_DYNAMIXEL\_AX12

MAKE\_DYNAMIXEL\_AX12(boolean inverted, uint8\_t, id, boolean continuous)

A macro to create a new AX12 servo.

The 'inverted' parameter has the same meaning as with other servos and motors. See "*actuators.h*" (see page 23)

The 'id' parameter is the unique ID number for the servo.

The 'continuous' parameter should be set to TRUE if you want to use the servo in continuous rotation mode (ie as a motor). FALSE is used to specify that it functions as a servo.

The servos must then be grouped into a DYNAMIXEL\_AX12\_LIST and finally we create a driver using MAKE\_DYNAMIXEL\_AX12\_DRIVER specifying the list, the hardware UART we want to use, and which I/O pin to use for the direction line. Note that due to the high communication speed required you must use a hardware UART.

Example:-



```
// Create 2 servos
DYNAMIXEL_AX12 servo1 = MAKE_DYNAMIXEL( FALSE, 2, FALSE );
DYNAMIXEL_AX12 servo2 = MAKE_DYNAMIXEL( FALSE, 3, FALSE );
// Put them in a list
DYNAMIXEL_AX12_LIST servos = { &servo1, &servo2 };
// Create a driver using UART1
DYNAMIXEL_AX12_DRIVER driver = MAKE_DYNAMIXEL_AX12_DRIVER( list, UART0, F0);
```

Once set up the servo position, or speed, can be controlled like any other servo by using:

```
"act_setSpeed(__ACTUATOR_COMMON* act, DRIVE_SPEED speed)" (see page 24)
```

---

## ax12Init

```
void ax12Init(DYNAMIXEL_AX12_DRIVER* driver, BAUD_RATE baud)
```

Initialise the driver to a given baud rate.

This must be called before the servos can be used. It is usually called from your `aplnitHardware` function.

The default baud rate is 1000000. Yes: 1 million bits per second !

---

## ax12GetInfo

```
uint16_t ax12GetInfo(DYNAMIXEL_AX12* servo)
```

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. Look in the AX12.h file to see the list of error codes.

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.

---

## ax12Dump

```
void ax12Dump(DYNAMIXEL_AX12* servo)
```

Dump the current status of a given servo to the `rprintf` destination.

This will execute an `ax12GetInfo` and either print out the error message or print out the information returned.

Also see: `ax12DumpAll`

---

## ax12DumpAll

```
void ax12DumpAll(const DYNAMIXEL_AX12_DRIVER* driver)
```

Dump the status of ALL of the servos to the `rprintf` destination.

---



This allows you to view a snapshot of all of the servos connected to one AX12 driver.

---

## **ax12Begin**

```
void ax12Begin(DYNAMIXEL_AX12_DRIVER* driver)
```

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 an ax12End command is received at the end of your main loop.

---

## **ax12End**

```
void ax12End(DYNAMIXEL_AX12_DRIVER* driver)
```

Marks the end of a command group.

See ax12Begin for a description of the Begin/End commands.

## **Advanced Function Detail**

### **ax12SetXXXX**

```
void ax12SetXXXX(servo, val)
```

Macros to allow advanced users to set a value in the servos Control Table.

Look inside Dynamixel/AX12.h to see the different options for XXXX.

Each of the macros expects the address of a servo and the 8 or 16 bit value to be written.

For example: to turn on the LED for servo12 then use:

```
ax12SetLED(&servo12, 1);
```

## **ax12Send**

```
void ax12Send(const DYNAMIXEL_AX12_DRIVER* driver, uint8_t id, size_t len, uint8_t* data)
```

Send a sequence of bytes to the AX12.

The byte sequence should start with 0xff, 0xff and end in the calculated checksum.

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 you can also use the functions defined in "*actuators.h*" (see page 23)

So to set the servo to its center position we could use:

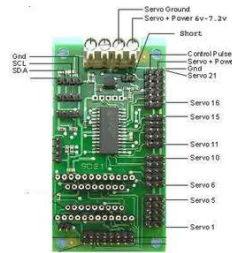
```
act_setSpeed(servo1, 0);
```



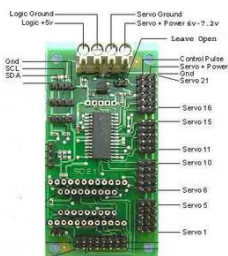
## Servos/Devantech/SD21.h

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

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



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



To use this controller you will need to define the servos as follows:

```
#include "Servos/Devantech/SD21.h"
// Define two servos on the SD21 card.
// This is similar to MAKE_SERVO(inverted, iopin, center, range)
// Described in "servos.h" (see page 98)
// except that there is no iopin to attach the servo to
SERVO left = MAKE_REMOTE_SERVO(FALSE, 1500, 500);
SERVO right = MAKE_REMOTE_SERVO(TRUE , 1500, 500);

// Create a single list of the servos on the SD21
// The first entry is 'Servo 0' on the SD21 board
SERVO* PROGMEM servos1[] = {&left,&right};

// Create a driver for the SD21 card
// and give it the list of servos
SERVO_DRIVER sd21 = MAKE_I2C_SERVO_DRIVER(0xC2,servos1);
```





```
// Create an I2C bus that just contains the SD21
static I2C_DEVICE_LIST i2c_list[] = { &sd21.i2cInfo };
I2C_HARDWARE_BUS i2c = MAKE_I2C_HARDWARE_BUS(i2c_list);
```

In `apnInitHardware` you will need to initialise the i2c bus and SD21 driver:-

```
// Initialise the i2c bus
i2cBusInit(&i2c);
// Initialise the servo controller
sd21Init(&sd21);
```

You may then move the servos using the functions in "*actuators.h*" (see page 23)

For example - the following code will continuously move the servos backwards and forwards:-

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
    TICK_COUNT ms = loopStart / 1000; // Get current time in ms
    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
    act_setSpeed(&left,speed);
    act_setSpeed(&right,speed);
    return 0;
}
```



## Servos/Generic/Serial Servo Controller

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).

To use the controller(s) you will need to define the servos in the standard way other than using MAKE\_REMOTE\_SERVO rather than MAKE\_SERVO because no IO pin needs to be specified :-

```
#include "servos.h"
SERVO left = MAKE_REMOTE_SERVO(FALSE, 1500, 500);
SERVO right = MAKE_REMOTE_SERVO(TRUE, 1500, 500);
// Create a single list of the servos
// The first entry is servo 0
SERVO* PROGMEM servos[] = {&left,&right};
// Create a driver and give it the list of servos
SERVO_DRIVER driver = MAKE_SERVO_DRIVER(servos);
```

In `aplnitHardware` you will need to initialise the driver:-



```
// Initialise the servo controller specifying the UART
// baud rate, and protocol
servoSerialInit(&driver,UART0,19200,POLOLU_COMPACT);
```

The protocol parameter must be either MINI\_SSC or POLOLU\_COMPACT

You may then move the servos using the functions in "*actuators.h*" (see page 23)



## Controller

Add support for hand held controllers.

Sony/ps2.h

333

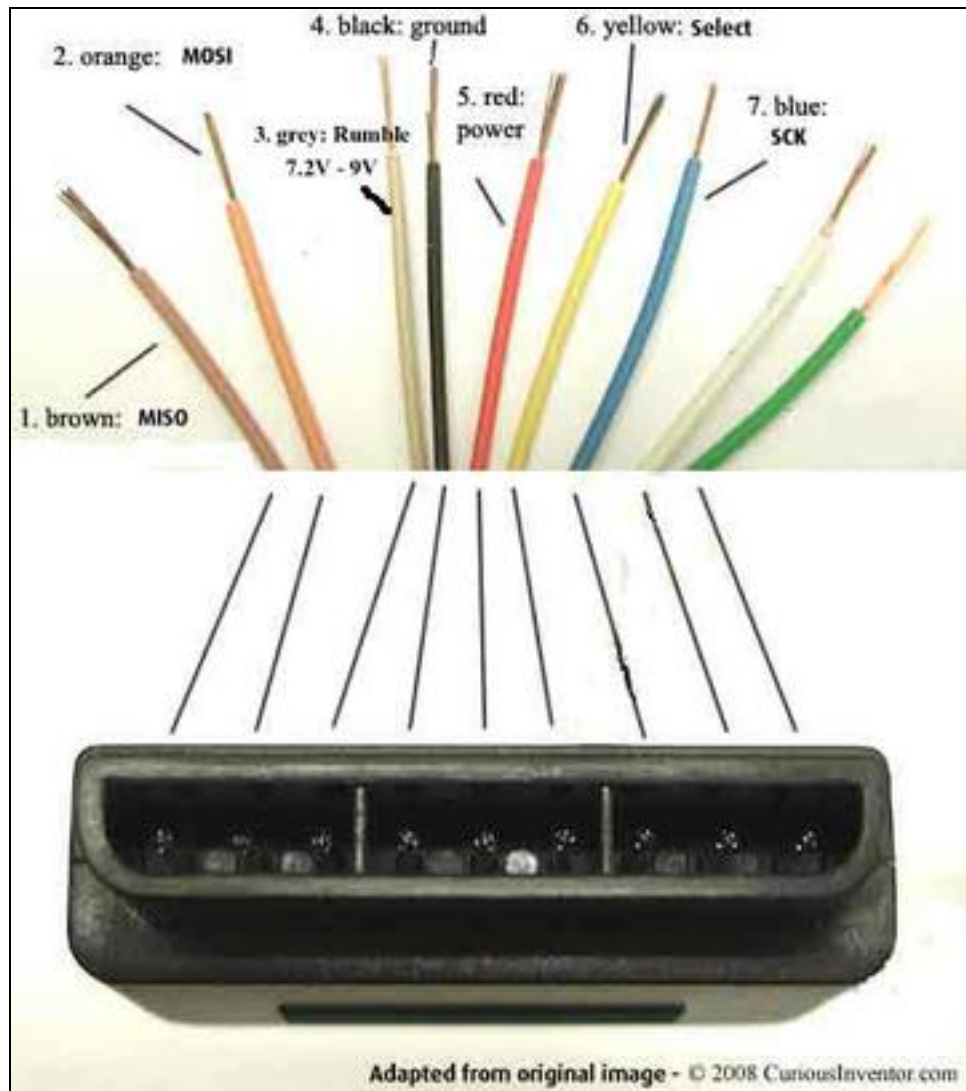


## Controller/Sony/ps2.h

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.



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.

The controllers are operated over an SPI bus and so you can do this by using either the hardware SPI bus, "*spi.h*" (see page 105) , or by creating a software SPI bus using whichever pins you prefer, "*spisw.h*" (see page 114) .

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.**

To create a controller which is selected using F0 then use:

```
SONY_PS2_controller1 = MAKE_SONY_PS2(F0);
```

You are now ready to use the device - but you may want to look at the 'calibrate' and 'setAnalogMode' functions for further set up that you can call from `applInitHardware`.

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.

The joystick names all start with `PS2_STICK_` and are defined in this H file.

Here is an example for the Axon and Axon II:-

```
// Create a controller SELECTed using F0
SONY_PS2_controller1 = MAKE_SONY_PS2(F0);

// Create a list of devices on the SPI bus
static SPI_DEVICE_LIST spiBus_list[] = {&controller1._device_};

// Create a software SPI Bus using:-
// B5 = MOSI, B6=MISO, B7=Clock
SPI_SW spiBus = MAKE_SW_SPI(spiBus_list,B5,B6,B7);
```

In `applInitHardware` we will initialise the SPI bus and then calibrate the joysticks which also sets analog mode:-



```
// Initialise the SPI bus
spiBusInit(&spiBus,true);
// Calibrate and set dead zone = 27
sonyPS2_calibrate(&controller1, 27);
```

In our main loop we read the controller:-

```
if(sonyPS2_read(&controller1)){
    // We can now test the buttons
    // and the joysticks
}
```

## Standard Function Summary

boolean	<a href="#">sonyPS2_calibrate(SONY PS2* controller, uint8_t deadzone)</a> Calibrate the joysticks on the controller.
boolean	<a href="#">sonyPS2_setAnalogMode(SONY PS2* controller)</a> Activates the joysticks.
boolean	<a href="#">sonyPS2_isAnalogMode(const SONY PS2* controller)</a> Returns TRUE if the controller is already in analogue mode.
boolean	<a href="#">sonyPS2_read(SONY PS2* controller)</a> Read the values from the controller and store them.
boolean	<a href="#">sonyPS2_buttonPressed(const SONY PS2* controller, uint16_t button)</a> Return TRUE if the specified button has been pressed.
boolean	<a href="#">sonyPS2_buttonDown(const SONY PS2* controller, uint16_t button)</a> Return TRUE if the button has just been pressed or FALSE if it was already pressed or is not pressed at all.
boolean	<a href="#">sonyPS2_buttonHeld(const SONY PS2* controller, uint16_t button)</a> Return TRUE if the button continues to be held down.
boolean	<a href="#">sonyPS2_buttonUp(const SONY PS2* controller, uint16_t button)</a> Returns TRUE if the button has just been released.
int8_t	<a href="#">sonyPS2_joystick(const SONY PS2* controller, PS2_STICK stick)</a> Read a joystick value relative to its centre point including any dead zone.
uint8_t	<a href="#">sonyPS2_buttonPressure(const SONY PS2* controller, uint16_t button)</a> Return a value representing how hard a button has been



## Standard Function Summary

	pressed.
void	<a href="#">soneyPS setRumble(SONY PS2* controller, uint8 t left, boolean right)</a> Turn on the rumble motors.
uint16_t	<a href="#">soneyPS2 buttonsChanged(const SONY PS2* controller)</a> Returns information on which buttons have changed state since the previous call to read the controller.

## Advanced Function Summary

uint16_t	<a href="#">soneyPS2 buttonsRaw(const SONY PS2* controller)</a> Return the status of all 16 buttons.
uint8_t	<a href="#">soneyPS2 joystickRaw(const SONY PS2* controller, PS2 STICK stick)</a> Reads the raw value from a joystick.

## Standard Function Detail

### soneyPS2\_calibrate

```
boolean soneyPS2_calibrate(SONY_PS2* controller, uint8_t deadzone)
```

Calibrate the joysticks on the controller.

This command should only be called once - when your program starts up. 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 not being 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.

### soneyPS2\_setAnalogMode

```
boolean soneyPS2_setAnalogMode(SONY_PS2* controller)
```

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. Note that the 'calibrate' function will automatically try to call this command.

---

### **sonyPS2\_isAnalogMode**

`boolean sonyPS2_isAnalogMode(const SONY_PS2* controller`

`)`  
Returns TRUE if the controller is already in analogue mode.

Note that this is only valid after one successful 'read' command has been issued.

---

### **sonyPS2\_read**

`boolean sonyPS2_read(SONY_PS2* controller)`

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.

---

### **sonyPS2\_buttonPressed**

`boolean sonyPS2_buttonPressed(const SONY_PS2* controller, uint16_t button)`

Return TRUE if the specified button has been pressed.

The button names all start with PS2\_BTN\_ and are defined in this H file.

Note that you can only test for one button at a time. If you need to test for multiple buttons then compare with `sonyPS2_buttonsRaw(const SONY_PS2* controller)`

---

### **sonyPS2\_buttonDown**

`boolean sonyPS2_buttonDown(const SONY_PS2* controller, uint16_t button)`

Return TRUE if the button has just been pressed or FALSE if it was already pressed or is not pressed at all.

---

### **sonyPS2\_buttonHeld**

`boolean sonyPS2_buttonHeld(const SONY_PS2* controller, uint16_t button)`

Return TRUE if the button continues to be held down.

---



### **sonyPS2\_buttonUp**

`boolean sonyPS2_buttonUp(const SONY_PS2* controller, uint16_t button)`

Returns TRUE if the button has just been released.

---

### **sonyPS2\_joystick**

`int8_t sonyPS2_joystick(const SONY_PS2* controller, PS2_STICK stick)`

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'.

The joystick names all start with PS2\_STICK\_ and are defined in this H file.

---

### **sonyPS2\_buttonPressure**

`uint8_t sonyPS2_buttonPressure(const SONY_PS2* controller, uint16_t button)`

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.

---

### **sonyPS\_setRumble**

`void sonyPS_setRumble(SONY_PS2* controller, uint8_t left, boolean right)`

Turn on the rumble motors.

Note that there are two different values - one for each motor. One motor can either be on or off whereas the other can be set to any value between 0 and 255.

The motor settings will only change when the next call to read the controller is used.

---

### **sonyPS2\_buttonsChanged**

`uint16_t sonyPS2_buttonsChanged(const SONY_PS2* controller)`

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 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.

---



## Advanced Function Detail

### sonyPS2\_buttonsRaw

uint16\_t sonyPS2\_buttonsRaw(const SONY\_PS2\* controller)

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. The button names all start with PS2\_BTN\_ and are defined in this file.

This function is useful if you want to do a 'Press any button to continue' function ie

```
sonyPS2_read(&controller);
uint16_t current = sonyPS2_buttonsRaw(&controller);
do{
    sonyPS2_read(&controller);
} while (sonyPS2_buttonsRaw(&controller) == 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)
sonyPS2_read(&controller);
if( (sonyPS2_buttonsRaw(&controller) & COMBO) == COMBO){
    // Both pressed
}
```

---

### sonyPS2\_joystickRaw

uint8\_t sonyPS2\_joystickRaw(const SONY\_PS2\* controller, PS2\_STICK stick)

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.

The joystick names all start with PS2\_STICK\_ and are defined in this H file.



## Gait

Adds support for using gaits in both design mode and free running mode.

GaitDesigner.h

341

GaitRunner.h

343



## 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 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 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 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 `aplnitHardware` 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 343) so that the robot can play the gait without the connection to the PC.

## Standard Function Summary

void	<a href="#">gaitDesignerInit(GAIT_DESIGNER* gait)</a> Initialise the gait program.
void	<a href="#">gaitDesignerProcess(GAIT_DESIGNER* gait)</a> 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 341) :-

#include the gait file produced by Gait Designer using the File | Export menu option.

Looking at this file you will find that it defines a constant (G8\_ANIM\_xxxx) for each animation in the gait and these numbers 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 `apInitHardware` 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	<a href="#"><u>MAKE G8_RUNNER</u></a> Create a standalone gait runner.
	<a href="#"><u>gaitRunnerInit(G8_RUNNER* runner)</u></a> Initialise the G8_RUNNER object.
boolean	<a href="#"><u>gaitRunnerProcess(G8_RUNNER* runner)</u></a> Tell the servos to move to their next position.
	<a href="#"><u>gaitRunnerPlay(G8_RUNNER* runner, uint8 t animation, int16 t loopSpeed, DRIVE_SPEED speed, int16 t repeatCount)</u></a> Start playing a given animation.
boolean	<a href="#"><u>gaitRunnerIsPlaying(const G8_RUNNER* runner)</u></a> Return TRUE if an animation is still playing or FALSE if it has stopped.
	<a href="#"><u>gaitRunnerStop(G8_RUNNER* runner)</u></a> Force the current animation to stop once it has reached the last frame of the animation.
int16_t	<a href="#"><u>gaitRunnerRepeatCount(const G8_RUNNER* runner)</u></a> Returns the number of loops remaining before the animation ends.
	<a href="#"><u>gaitRunnerSetSpeed(G8_RUNNER* runner, DRIVE_SPEED speed)</u></a> Sets the current playback speed of the animation.
DRIVE_SPEED	<a href="#"><u>gaitRunnerGetSpeed(const G8_RUNNER* runner)</u></a> Returns the current playback speed for the animation.

## Advanced Function Summary

	<a href="#"><u>gaitRunnerSetDelta(G8_RUNNER* runner, uint8 t limbNumber, DRIVE_SPEED speed)</u></a> Allows you to modify the gait in real time.
DRIVE_SPEED	<a href="#"><u>gaitRunnerGetDelta(const G8_RUNNER* runner, uint8 t limbNumber)</u></a> 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:

$((65.536 * \text{loopSpeed}) / \text{speed})$  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 347)



## Text2Speech/Text2Speech.h

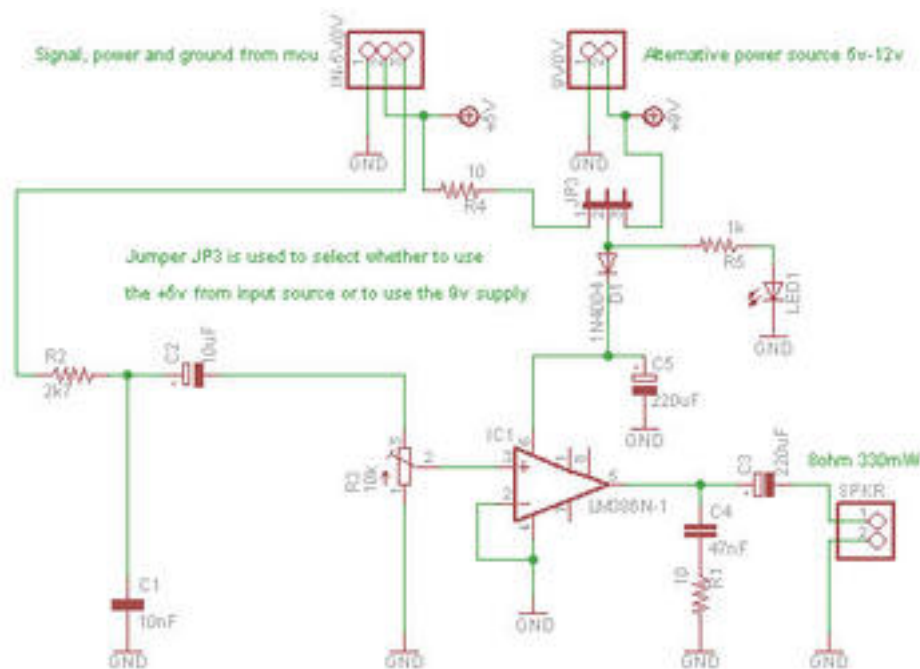
Implements a voice synthesiser using a single hardware PWM pin.

### 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:



I have uploaded my Eagle schematic and board files at [http://www.societyofrobots.com/member\\_tutorials/node/216](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 if the speaker is in some kind of box. Try cupping the speaker 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

If you are using Project Designer then you can add the 'Text to Speech' synthesiser from the 'Basic' category of devices. You can also make it the destination for error messages and `rprintf` when generating the code for your project.

Otherwise you will need to call:

```
speechInit(H6);
```

from your `applInitHardware` where H6 is the hardware PWM output pin of your choice.

Either way: the simplest method to output some text is:

```
sayText("Death to all humans");
```

However: this has some drawbacks:

1. The compiler will store the string in both RAM and program memory.
2. The message is fixed

WebbotLib also provides a `Writer` - which means that you can use '`rprintf`' to output variable information eg:

```
Writer old = rprintfInit(getSpeechWriter()); // Send rprintf to speech
rprintf("Death to all humans\n"); // String is only in program memory
rprintf("A number is %u\n",9); // Output a variable msg
rprintfInit(old); // Revert rprintf output to its original place
```

Note that when accessed via `rprintf` WebbotLib 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](http://www.societyofrobots.com/robot_tutorial.shtml) but using an ATmega328P rather than the ATmega8.

### Standard Function Summary

	<a href="#"><code>speechInit(const IOPin* pin)</code></a> Initialise the speech system.
	<a href="#"><code>sayText(const char * src)</code></a> Immediately speaks the English text.
<code>uint8_t</code>	<a href="#"><code>getPitch(void)</code></a> Returns the current default pitch.
	<a href="#"><code>setPitch(uint8_t pitch)</code></a> Changes the default pitch of the voice.

### Advanced Function Summary

<code>Writer</code>	<a href="#"><code>getSpeechWriter(void)</code></a> Obtain a Writer that can be used with <code>rprintf</code> commands.
---------------------	--

### Standard Function Detail

#### **speechInit**

`speechInit(const IOPin* pin)`

Initialise the speech system.

This should be called in `applInitHardware` and the single parameter is an `IOPin` that provides hardware PWM.

If you are using Project Designer then this is done for you auto-magically.



## sayText

`sayText(const char * src)`

Immediately speaks the English text.

For example:

```
sayText("Death to all humans");
```

---

## getPitch

`uint8_t getPitch(void)`

Returns the current default pitch.

---

## setPitch

`setPitch(uint8_t pitch)`

Changes the default pitch of the voice.

Higher numbers will result in a lower voice and smaller number will give a higher voice.

## Advanced Function Detail

### getSpeechWriter

`Writer getSpeechWriter(void)`

Obtain a Writer that can be used with `rprintf` commands.

Although the `sayText(const char * src)` command is useful for writing out fixed pieces of text there are times when we want to output variable text. For example we may want to speak the value of a distance sensor - whose value is only known at runtime.

These values can be converted into text using the `rprintf` commands but we need to make sure that `rprintf` output is going to the speech sub-system rather than the default location which may a UART for data logging.

Assuming you are using WebbotLib to read the distance sensors then the distance will be returned in cm and can be dumped out by `rprintf` using the `distanceDump` command. So all we need to do is tell `rprintf` to 'speak' the result rather than to send it to where it is currently going:

```
// Tell rprintf to send output to speech, and remember the old destination
Writer old = rprintfInit(getSpeechWriter());
// Dump the sensor values to rprintf (ie speech)
distanceDump(sensor);
// Make rprintf output go to original device
rprintfInit(old);
```



## Maths

The Maths folder contains various helper routines for more complex mathematical processing such as Vectors and Matrices.

Vector2D.h	353
Vector3D.h	359
Matrix3D.h	365





## 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  $\text{SQRT}(1 \times 1 + 0 \times 0) = 1$  and the length of the second is  $\text{SQRT}(3 \times 3 + 0 \times 0) = 3$ . We can normalise these headings by dividing the X,Y values by the length ie  $(1,0) \Rightarrow (1/1, 0/1) = (1,0)$  and the second is  $(3,0) \Rightarrow (3/3, 0/3) = (1,0)$ . 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  $0^\circ$  then the object is 'dead-ahead'; but if it is less than  $-90^\circ$  or greater than  $90^\circ$  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

## Standard Function Summary

VECTOR2D	<a href="#">MAKE_VECTOR2D(x,y)</a> Construct a new vector with the given x,y values.
double	<a href="#">vector2d GetX(const VECTOR2D* vector)</a> Returns the X value of the vector.
double	<a href="#">vector2d GetY(const VECTOR2D* vector)</a> Returns the Y value of the vector.
	<a href="#">vector2d SetX(VECTOR2D* vector, double x)</a> Set the X value of an exiting vector.
	<a href="#">vector2d SetY(VECTOR2D* vector, double y)</a> Set the Y value of an exiting vector.
	<a href="#">vector2d Set(VECTOR2D* vector, double x, double y)</a> Overwrite the vector with new values.
double	<a href="#">vector2d Length(const VECTOR2D* vector)</a> Get the length of the vector.



## Standard Function Summary

	<a href="#">vector2d Normalise(VECTOR2D* dst,const VECTOR2D* src)</a> Normalise the vector so that it has a length of 1.
	<a href="#">vector2d Add(VECTOR2D* dst,const VECTOR2D* src)</a> Add two vectors ie $dst = dst + src$
	<a href="#">vector2d Subtract(VECTOR2D* dst,const VECTOR2D* src)</a> Subtract two vectors.
	<a href="#">vector2d Copy(VECTOR2D* dst,const VECTOR2D* src)</a> Set a vector to be a copy of another vector.
	<a href="#">vector2d Scale(VECTOR2D* v,double scale)</a> Multiply both the X and Y values of the vector by the given scale factor.
double	<a href="#">vector2d AngleRadians(const VECTOR2D* v1, const VECTOR2D* v2)</a> Return the angle, in radians, between two vectors.

## Advanced Function Summary

double	<a href="#">vector2d LengthSquared(const VECTOR2D* vector)</a> Return the length squared of the vector.
double	<a href="#">vector2d DotProduct(const VECTOR2D* v1, const VECTOR2D* v2)</a> 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  $\text{SQRT}(x*x + y*y)$

---



**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:  $\text{Length}(v1) * \text{Length}(v2) * \cos(\text{angle between them})$

If 'v1' and 'v2' have been normalised then their lengths are 1.0 and so the above simplifies to:  $\cos(\text{angle between them})$



## 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.

### Standard Function Summary

VECTOR3D	<a href="#"><u>MAKE VECTOR3D(x,y,z)</u></a> Construct a new vector with the given x,y,z values.
double	<a href="#"><u>vector3d GetX(const VECTOR3D* vector)</u></a> Returns the X value of the vector.
double	<a href="#"><u>vector3d GetY(const VECTOR3D* vector)</u></a> Returns the Y value of the vector.
double	<a href="#"><u>vector3d GetZ(const VECTOR3D* vector)</u></a> Returns the Z value of the vector.
	<a href="#"><u>vector3d SetX(VECTOR3D* vector, double x)</u></a> Set the X value of an exiting vector.
	<a href="#"><u>vector3d SetY(VECTOR3D* vector, double y)</u></a> Set the Y value of an exiting vector.
	<a href="#"><u>vector3d SetZ(VECTOR3D* vector, double y)</u></a> Set the Z value of an exiting vector.
	<a href="#"><u>vector3d Set(VECTOR3D* vector, double x, double y, double z)</u></a> Overwrite the vector with new values.
double	<a href="#"><u>vector3d Length(const VECTOR3D* vector)</u></a> Get the length of the vector.
	<a href="#"><u>vector3d Normalise(VECTOR3D* dst,const VECTOR3D* src)</u></a> Normalise the vector so that it has a length of 1.
	<a href="#"><u>vector3d Add(VECTOR3D* dst, const VECTOR3D* src)</u></a> Add two vectors ie dst = dst + src
	<a href="#"><u>vector3d Subtract(VECTOR3D* dst, const VECTOR3D* src)</u></a> Subtract two vectors.
	<a href="#"><u>vector3d Copy(VECTOR3D* dst, const VECTOR3D* src)</u></a> Set a vector to be a copy of another vector.



## Standard Function Summary

	<a href="#">vector3d Scale(VECTOR3D* v,double scale)</a> Multiply the X, Y and Z values of the vector by the given scale factor.
	<a href="#">vector3d CrossProduct(VECTOR3D*result, const VECTOR3D* v1, const VECTOR3D* v2)</a> Calculate the vector cross product.
double	<a href="#">vector3d AngleRadians(const VECTOR3D* v1, const VECTOR3D* v2)</a> Return the angle, in radians, between two vectors.

## Advanced Function Summary

double	<a href="#">vector3d LengthSquared(const VECTOR3D* vector)</a> Return the length squared of the vector.
double	<a href="#">vector3d DotProduct(const VECTOR3D* v1, const VECTOR3D* v2)</a> 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  $\text{SQRT}(x*x + y*y + z*z)$

---

### **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  $\text{dst} = \text{dst} + \text{src}$

---

### **vector3d\_Subtract**

`vector3d_Subtract(VECTOR3D* dst, const VECTOR3D* src)`

Subtract two vectors.  $\text{dst} = \text{dst} - \text{src}$ ;

---

### **vector3d\_Copy**

`vector3d_Copy(VECTOR3D* dst, const VECTOR3D* src)`

Set a vector to be a copy of another vector.  $\text{dst} = \text{src}$ .

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:  $\text{Length}(v1) * \text{Length}(v2) * \cos(\text{angle between them})$

If 'v1' and 'v2' have been normalised then their lengths are 1.0 and so the above simplifies to:  $\cos(\text{angle between them})$



## 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.

### Standard Function Summary

MATRIX3D

#### [MAKE\\_IDENTITY\\_MATRIX3D](#)

Create an identity matrix.

#### [matrix3d\\_Copy\(MATRIX3D\\* dst, const MATRIX3D\\* src\)](#)

Copy one matrix to another ie dst = src.

#### [matrix3d\\_Multiply\(MATRIX3D\\* dst, const MATRIX3D\\* src\)](#)

Multiply two matrices together.

#### [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\(MATRIX3D\\* matrix, double radians\)](#)

Set the matrix to be a rotation around the Y axis by the specified number of radians.

#### [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\(MATRIX3D\\* matrix, double scale\)](#)

Set the matrix to the given scaling factor in X, Y and Z.

#### [matrix3d\\_Transform\(VECTOR3D\\* dst, const VECTOR3D\\* src, const MATRIX3D\\* matrix\)](#)

Transform a VECTOR3D by the given matrix.



## Advanced Function Summary

double	<a href="#">matrix3d_Determinant(const MATRIX3D* matrix)</a> Returns the determinant of the matrix.
	<a href="#">matrix3d_Set(MATRIX3D* matrix,double m00,double m01,double m02,double m10,double m11,double m12,double m20,double m21,double m22)</a> 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.

---

## **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 ]

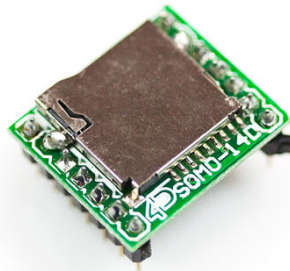
[ m20 m21 m22 ]



## Audio/SOMO14D.h

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 when the Axon is powered up:





```
#include <sys/axon.h>
#include <Audio/SOMO14D.h>

// Define the SOMO device
SOMO14D audio = MAKE_SOMO14D(F6,F5,NULL);

// Initialise the hardware
void appInitHardware(void) {
    // Initialise the SOMO
    somo14D_init(&audio);
}

// Initialise the software
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Play file '0000.ad4'
    somo14D_play(&audio,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 use the third parameter of the MAKE\_SOMO14 command.

```
#include <sys/axon.h>
#include <Audio/SOMO14D.h>

SOMO14D audio = MAKE_SOMO14D(F6,F5,F7);

// 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;

// Initialise the hardware
void appInitHardware(void) {
    // Initialise the SOMO
    somo14D_init(&audio);
}
```



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

```
// Start playing a sequence of sounds
void play(int16_t* effect){
    // Stop any current playback
    som14D_stop(&audio);
    seqPos = effect;

    // Start playing the first
    // sound in the sequence
    som14D_play(&audio,*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(!somo14D_isBusy(&audio)){
            // Move to next sound in sequence
            seqPos++;
            if(*seqPos==--1){
                // Sequence has finished
                seqPos = NULL;
            }else{
                // Play next sound in sequence
                somo14D_play(&audio, *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;
}

```

## Standard Function Summary

	<a href="#"><u>MAKE SOMO14D(const IOPin* clk,const IOPin* data,const IOPin* busy)</u></a> Create a SOMO14D device.
	<a href="#"><u>somo14D_init(const SOMO14D* device)</u></a> Initialise the SOMO14D device.
boolean	<a href="#"><u>somo14D_isBusy(const SOMO14D* device)</u></a> Returns TRUE if the device is currently playing a file.
	<a href="#"><u>somo14D_play(const SOMO14D* device, uint16_t fileNum)</u></a> Start playing the specified file number, in the range 0 to 511, on the device.
	<a href="#"><u>somo14D_stop(const SOMO14D* device)</u></a> Stop playing any current sound file.



---

## Standard Function Summary

	<a href="#"><code>somo14D_volume(const SOMO14D* device, uint8_t volume)</code></a>
--	--

	Set the volume level on the device to value between 0 (quietest) and 7 (loudest).
--	---

---

## Standard Function Detail

### MAKE\_SOMO14D

`MAKE_SOMO14D(const IOPin* clk, const IOPin* data, const IOPin* busy)`

Create a SOMO14D device.

The first two parameters are compulsory and specified the digital output pins to use to connect to the CLK and DATA pins of the SOMO14D. Note that if your micro controller has a Vcc of more than 3.3V then you will need to add a 100 ohm resistor on each of these lines.

The third parameter is optional and can either be NULL or a digital input pin to connect to the BUSY pin on the SOMO14D. This pin is only required if you want to use the `somo14D_isBusy` command.

---

### somo14D\_init

`somo14D_init(const SOMO14D* device)`

Initialise the SOMO14D device.

This must be called from `applnitHardware` before any other `somo14d` functions can be used.

---

### somo14D\_isBusy

`boolean somo14D_isBusy(const SOMO14D* device)`

Returns TRUE if the device is currently playing a file.

To use this command you must have specified a IO pin in the third parameter of your `MAKE_SOMO14D` command and connect this to the BUSY output of the device.

---

### somo14D\_play

`somo14D_play(const SOMO14D* device, uint16_t fileNum)`

Start playing the specified file number, in the range 0 to 511, on the device.

If another sound is currently being played then it will be stopped before playing the new file.



### **somo14D\_stop**

`somo14D_stop(const SOMO14D* device)`

Stop playing any current sound file.

---

### **somo14D\_volume**

`somo14D_volume(const SOMO14D* device, uint8_t volume)`

Set the volume level on the device to value between 0 (quietest) and 7 (loudest).



## Frequently Asked Questions

As with all FAQs - this is an ever-growing list of your questions.

**Q:** My program occasionally hangs. Background tasks like hardware PWM and interrupts continue to function but everything else dies. Why?

**A:** There is a known issue (thank you Admin at SoR) if you have defined I2C devices but they are not physically connected. It is recommended that any unused sensors are removed from your code in order to avoid this sort of issue. This will prevent the 'hanging' and, as important, stop your program from responding to 'fictitious' responses from the sensors.

**Q:** The compiler generates the error '**region text is full**'. What does this mean?

**A:** Your program is too big to fit on the processor. You will either need to make your program smaller, experiment with optimisation settings, or upgrade to a larger processor.

**Q:** The compiler generates the error '**multiple definition of `\_\_floatunsidf`**'. What does this mean?

**A:** There is a problem in your Library Configuration. You either haven't specified the required libraries or they are not listed in the correct order. See the section: Getting started with AVRStudio

**Q:** I've seen references to a **Writer** but I just don't understand what they are or what they are for?

**A:** A Writer is a 'place' that you can send text to. If you've used other programming languages and are familiar with the concept of an output stream then 'that' is a Writer. If not then imagine that you have 4 UARTs. Each of these are capable of writing stuff out to other devices such as a terminal emulation program running on your PC. But the bare bones of a Writer is to say 'output this byte'. Complex text formatting commands, such as those in `rprintf.h`, end up sending out each character to the Writer last set by `rprintfinit`. This command completes the 'plumbing' from `rprintf` to the 'Writer' All of these destinations have a 'GetWriter' function. So all the UARTs, Displays, (and anything else you may want to direct output to) support this function.

