FEN LOGIC LTD.

Gertboard User Manual

Gert van Loo and Myra VanInwegen

Revision 2.0

The Gertboard is an add-on GPIO expansion board for the Raspberry Pi computer. It comes with a large variety of components, including buttons, LEDs, A/D and D/A converters, a motor controller, and an Atmel AVR microcontroller. There are test programs for the Gertboard written in C and Python, which are freely available on the Web. This manual explains both how to set up the Gertboard for various control experiments and also explains at a high level how the test code works.

Copyright 2012 by Fen Logic Ltd. All rights reserved.

Contents

Gertboard Overview

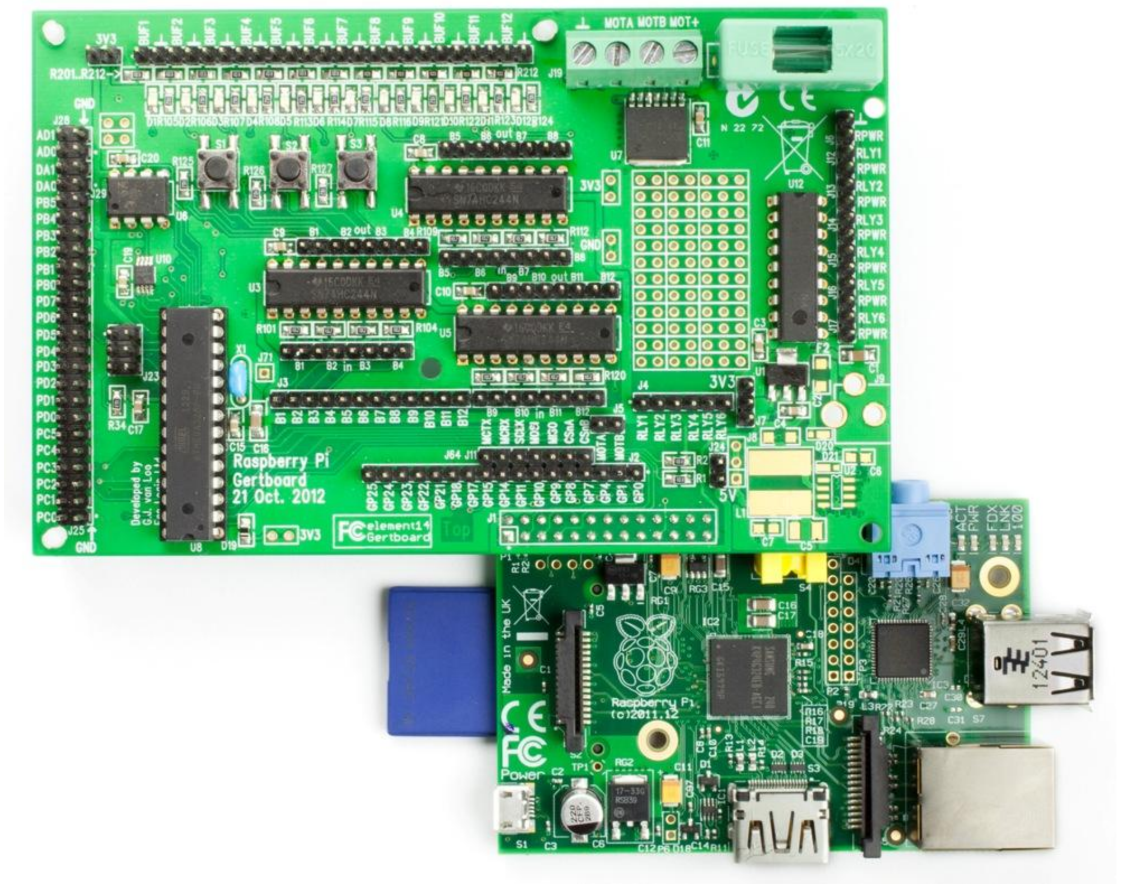


Figure 1: Gertboard and Raspberry Pi

The Gertboard is an input/output (I/O) extension board for the Raspberry Pi computer. It fits onto the GPIO (general purpose I/O) pins of the Raspberry Pi (the double row of pins on the upper left corner) via a socket on the back of the Gertboard. A bit of care is required when putting the two devices together. It is easy to insert just one row of pins into the socket, but all of the pins need to be connected. The Gertboard gets its power from those GPIO pins, so you will need a power supply for the Raspberry Pi (RPi) that is capable of supplying a current of at least 1A.

The Gertboard has collection of functional blocks (the major capabilities of the board) which can be connected together in a myriad of ways using header pins. The functional blocks are:

- 12x buffered I/O
- 3x pushbuttons
- 6x open collector drivers (50V, 0.5A)
- 18V, 2A motor controller
- 28-pin dual in line ATmega microcontroller
- 2-channel 8, 10, or 12 bit Digital to Analogue converter
- 2-channel 10 bit Analogue to Digital converter

The location of these blocks on the Gertboard is shown in Figure 2.

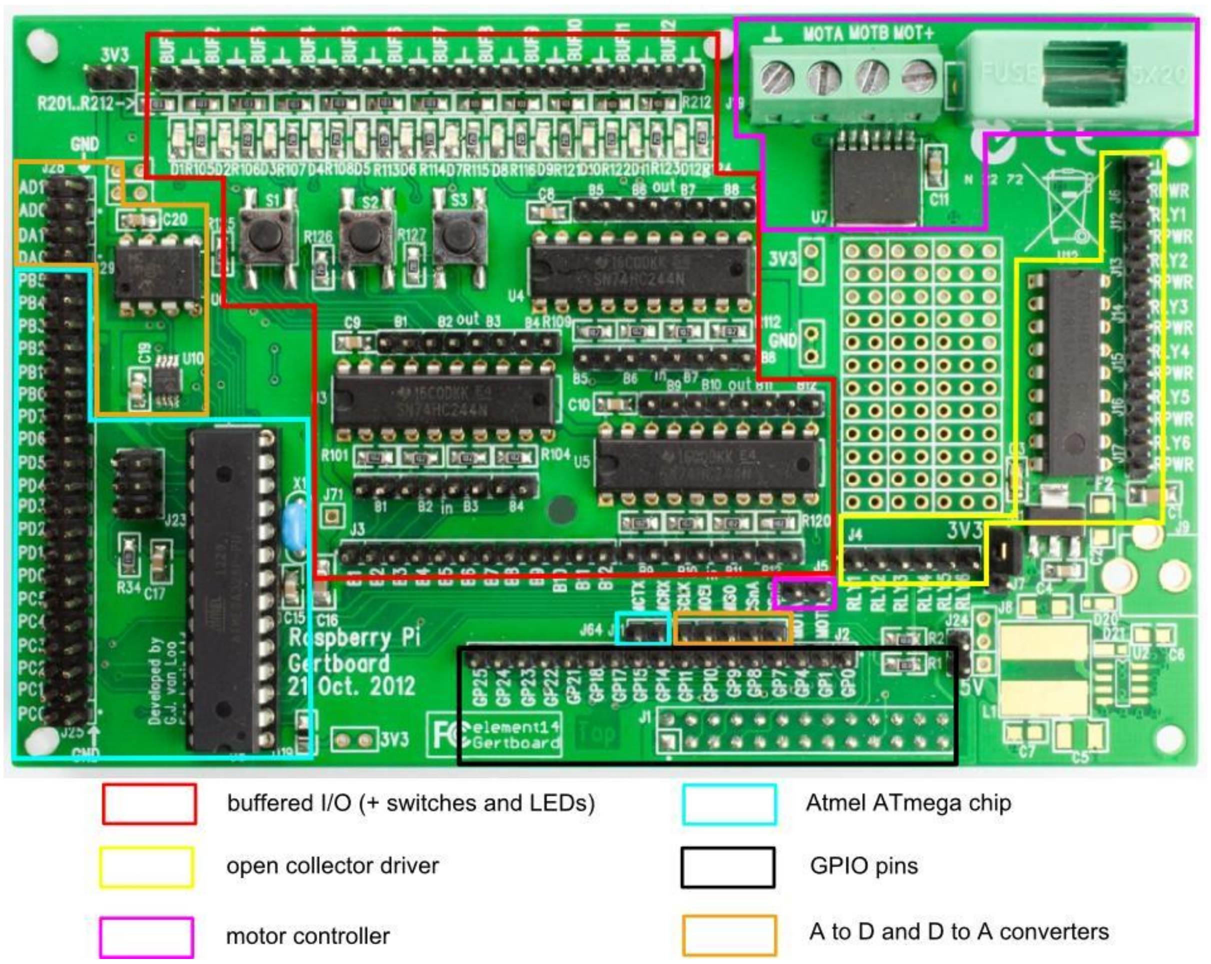


Figure 2: Functional blocks diagram: the key blocks are identified by coloured boundary marking. Please note that the appearance of some components can vary.

This annotated photo of a populated (fully assembled) Gertboard shows where the functional blocks are located. Some of the blocks have two areas marked. For example, the turquoise lines showing the Atmel ATmega chip not only surround the chip itself and the header pins next to it (on the lower left) but also surround two header pins near the bottom of the board, in the middle. These two pins are connected to the Atmel chip and provide an easy way to interface the GPIO signals from the Raspberry Pi (which are in the black box) with the Atmel chip.

There is no connection (other than power and ground) between the different functional blocks on the Gertboard. The many headers (the rows of pins sticking up from the board) allow you to make these connections, using straps and jumpers. See Figure 11 on page 18 for an example of how these are used to connect the various blocks together.

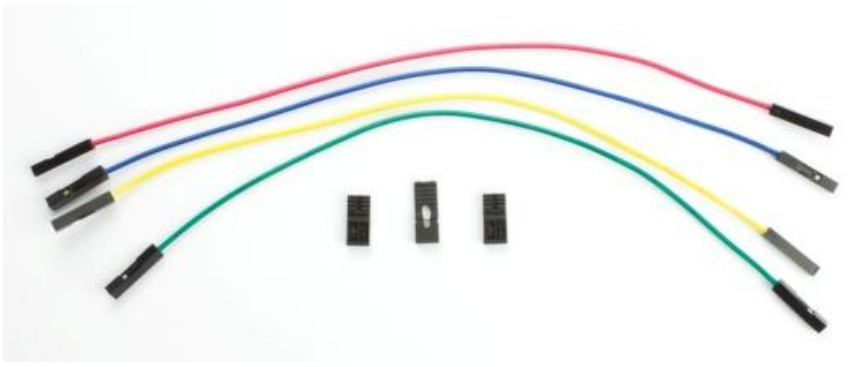


Figure 3: Photograph showing straps (the coloured wires) above, and jumpers below. Straps connect two parts of Gertboard together, whilst jumpers conveniently connect two adjacent pins together.

Labels and Diagrams

As you get to know the Gertboard and make connections between the various blocks, you will be guided extensively by the white labels on the circuit board. At this point, we would like to introduce the diagram, shown in Figure 5, which we will use to show you how to wire up your Gertboard for the test programs.

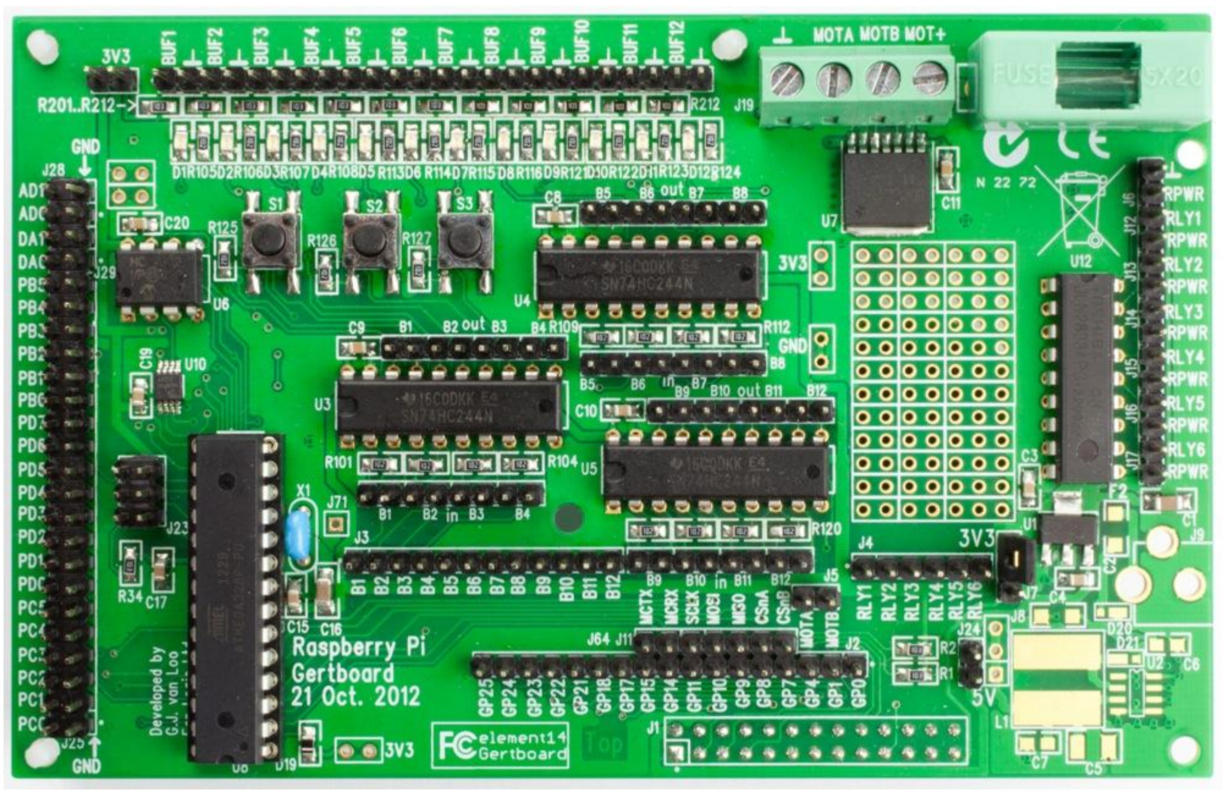


Figure 4: Photograph of the Gertboard

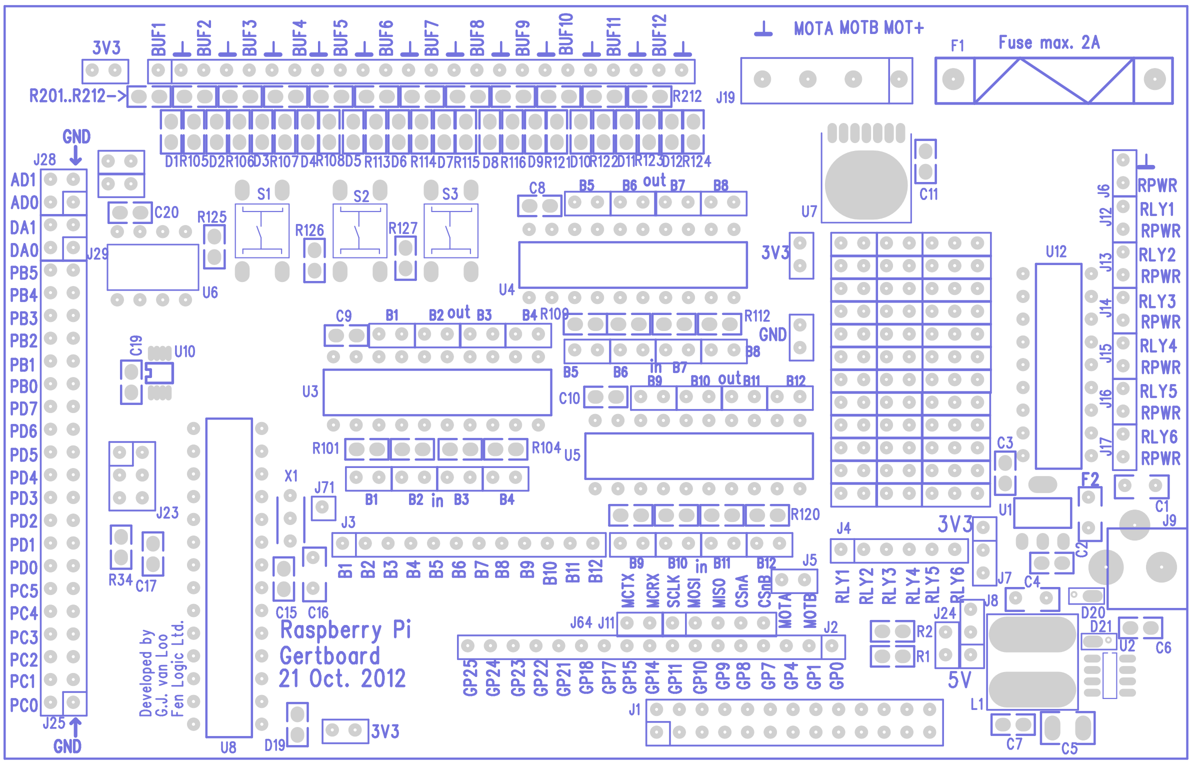


Figure 5: Diagram representing a bare Gertboard circuit board. The blue elements correspond to the white lines and text, and the gray elements correspond to the silver coloured pads.

The diagram in Figure 5 is created from the files that were used to design the Gertboard circuit board. The blue in the diagram is generated from the silkscreen file, which indicates where the white text and lines on the circuit board will be. The grey in the diagram is generated from the solder mask file, which roughly corresponds to where the silver conductive areas on the top of the circuit board will be. We will be using this blue and grey diagram as a basis for our wiring diagrams, which show you the pins that need to be connected together for each of the test programs. We use these diagrams because they are much clearer than a photo of the fully assembled board.

Have a close look at the white text on the photo of the Gertboard in Figure 4 (or the white text on your own board or the blue text in the diagram in Figure 5). These labels provide information that is required in order to connect together the various blocks of the Gertboard. Almost all of the components have labels, and more importantly, the pins in the headers have labels. It isn't necessary to be too concerned about many of the components, such as resistors and capacitors (labelled with R*n* and C*n*, where *n* is some number). However the labels for headers, integrated circuits, diodes, and switches are important.

Diodes are labelled D*n*. The ones that you will be interested in are D1 through D12, the LEDs (light emitting diodes). The LEDs are near the top of the board, on the left. The labels for them are a bit crowded. Each LED has a resistor next to it and thus each label D*n* has an R*m* next to it. The LEDs are easy to find when you have the board powered up, as they are a row of bright red lights. See below, in the section Power on the GertboardPower (page 9) for information on how to provide power to the Gertboard.

Pushbutton switches are labelled S1, S2, and S3 (they are located just beneath the LEDs).

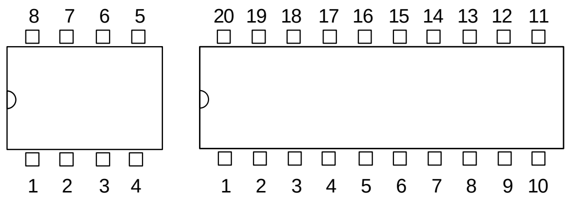


Figure 6: Two examples of ICs – an 8-pin and a 20-pin dual-inline package (DIP). In this package style, pin 1 is always identified as the first pin anticlockwise from the package notch marking.

Integrated circuits (also known as ICs or chips), are marked U*n*. For example the I/O buffer chips are U3, U4, and U5 (these are near the middle of the board), while the Atmel microcontroller is U8 (this is below and to the left of U3 to U5). It is important to understand IC pin numbering. If the chip is orientated so that the end with the semi-circle notch is to the left, then pin 1 is the leftmost pin in the bottom row. Pin numbers increase in an anti-clockwise direction from there, as shown in Figure 6. Knowing this means that the schematics in Appendix A can always be related to the pins on the ICs on the Gertboard.

Headers (the rows of pins sticking up from the board) will be a frequently used component on the Gertboard. They are labelled J*n*. For example, there is a collection of headers along the left edge of the board. They allow you to access the three chips on the left side of the board: J28 on top for the analogue to digital chip, J29 below that for the digital to analogue chip, and J25 below that for the Atmel microcontroller. It is a bit difficult to see the boundary between these headers on a fully assembled board; it's much clearer on the blue and grey diagram in Figure 5. On the Gertboard circuit board, each header with more than two pins has pin 1 marked with a square around it and a dot next to it. The dot is most useful on the assembled board, but these dots don't appear in the blue and grey diagram, so you can use the squares to find pin 1 there.

Not everything labelled J*n* is a collection of pins. J1, at the bottom of the board, is the location of the socket that connects the Gertboard to the Raspberry Pi. J19, at the top of the board (right of centre) is a block of screw terminals that allow you to easily connect wires from a power supply and a motor.

Power on the Gertboard

Power pins are marked with their voltage, e.g. 5V or 3V3 (this means 3.3V). A 5V power supply comes onto the board from the Raspberry Pi, and if you need this voltage it can be accessed from the lower pin (marked 5V) on header J24 on the lower right-hand corner of the board. Ground is marked with GND or a ⊥ symbol.

The supply voltage (the voltage that acts as high or logical 1 on the board) is 3.3V. This is generated from the 5V power pin in the J1 header by the components in the lower right corner of the board. To send the 3.3V power supply to the components on the Gertboard, you need to install a jumper over the top two pins of the header J7. It is near the lower right corner of the board; see the photo and diagram in Figure 7. The open collector and motor controllers can handle higher voltages and have points to attach external power supplies.

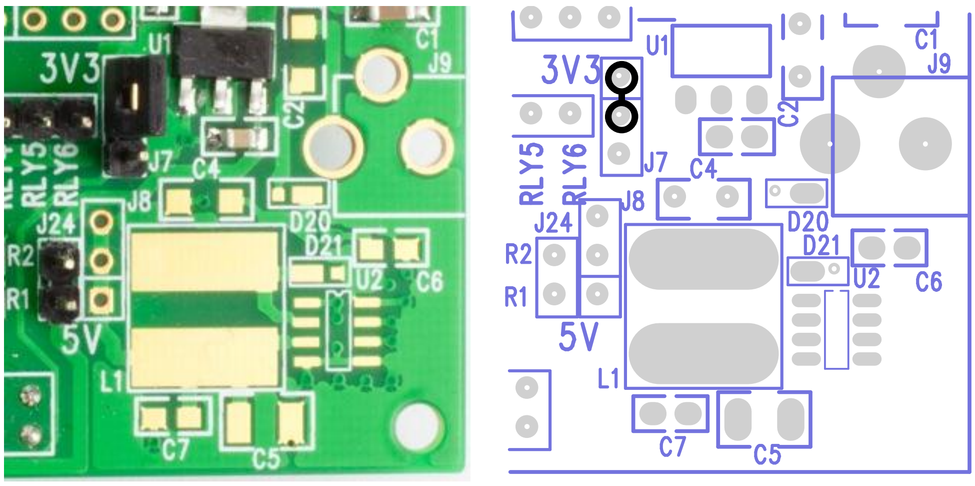


Figure 7: Power jumper installed in header J7: photo on left, diagram on right

The diagram on the right of Figure 7 above is our first example of a wiring diagram based on the blue and grey circuit board diagram. These diagrams indicate pins via black circles around the locations of pins on the board, and show connections as black lines between the circles. The diagram does not indicate directly whether the two pins should be joined by straps (wires) or jumpers. Generally, if the two pins are right next to each other, use a jumper, and if they are further apart, use a strap.

GPIO Pins

The header J2, to the right of the text 'Raspberry Pi Gertboard' on the board, provides access to all the I/O pins on the GPIO header. There are 26 pins in J1 (the socket which connects the Gertboard to the Raspberry Pi) but only 17 pins in J2: 3 of the pins in J1 are power (3.3V and 5V) and ground, and 6 are DNC (do not connect). The labels on these pins, GP0, GP1, GP4, GP7, etc, may initially seem a little arbitrary, as there are some obvious gaps, and the numbers do not correspond with the pin numbers on the GPIO header J1. These labels are important however: they correspond with the signal names used by the BCM2835, the processor on the Raspberry Pi (RPi). Signal GPIO*n* on the BCM2835 datasheet corresponds to the pin labelled GP*n* on header J2. At least, this was true of the first version of the Raspberry Pi ("rev1"). Starting in September 2012, revision 2 Raspberry Pis ("rev2") were starting to be shipped. On the rev2 RPis, some of the GPIO pins have been changed. The GPIO port that used to be controlled by GPIO21 is now controlled by GPIO27, and the ports that used to be controlled by GPIO0 and GPIO1 are now controlled by GPIO2 and GPIO3. The rest have remained the same. The first three columns of Table 1 below summarize the current situation.

Some of the GPIO pins have an alternate function that are made use of in some of the test programs. These are also shown in Table 1, in the last two columns. The ports that have nothing in the "Alt function" column are only used as general purpose input/output in the code. In the C test programs, we use macros to gain access to the alternative functions of the GPIO ports. This is explained in the section on analogue to digital and digital to analogue converts (D/A and A/D tests in C, page 35). In Python, we use packages to provide access to the alternative functions.

We mention the I^{2}C bus use of GPIO0 and GPIO1 (or GPIO2 and GPIO3 for rev2 RPis) in Table 1 not because the I^{2}C bus is used in the test programs, but because each of them has a 1800Ω pull-up resistor on the Raspberry Pi, and this prevents them from being used with the pushbuttons (see the section on Buffered I/O, LEDs, and Pushbuttons for more information).

Label on GB	Port on RPi1	Port on RPi2	Alt function (which alt)	Purpose
GP0	GPIO0	GPIO2	SDA	I^2C bus
GP1	GPIO1	GPIO3	SCL	
GP4	GPIO4	GPIO4		
GP7	GPIO7	GPIO7	SPI_CE1_N (alt 0)	SPI bus
GP8	GPIO8	GPIO8	SPI_CE0_N (alt 0)	
GP9	GPIO9	GPIO9	SPI_MISO (alt 0)	
GP10	GPIO10	GPIO10	SPI_MOSI (alt 0)	
GP11	GPIO11	GPIO11	SPI_SCLK (alt 0)	
GP14	GPIO14	GPIO14	TXD0 (alt 0)	UART
GP15	GPIO15	GPIO15	RXD0 (alt 0)	
GP17	GPIO17	GPIO17		
GP18	GPIO18	GPIO18	PWM0 (alt 5)	pulse width modulation
GP21	GPIO21	GPIO27		
GP22	GPIO22	GPIO22		
GP23	GPIO23	GPIO23		
GP24	GPIO24	GPIO24		
GP25	GPIO25	GPIO25		

Table 1: GPIO ports corresponding to "GP" labels, and alternative functions of GPIO ports (GB means Gertboard, RPi1 means Raspberry Pi rev1, RPi2 means Raspberry Pi rev2)

Schematics

Whilst there are some circuit diagrams, or schematics, in the main body of the manual for some of the functional blocks of the board, they are simplifications of the actual circuits. While these simplified diagrams and the explanations in the text will be good enough for most uses of the Gertboard, there will occasionally be questions that can only answered by knowing exactly what is on the board. Thus we have attached the full schematics at the end of this manual as Appendix A. These pages are in landscape format. The page numbers A-1, A-2, etc, are in the lower left corner of the pages (if you hold them so that the writing is the right way up).

Test Programs Overview

There are test programs for the Gertboard written in C and in Python. C provides the most direct access to the Gertboard functionality, but it is not a language that is very accessible to the beginner programmer. Several packages have been written to allow Python code to access the Raspberry Pi GPIO pins and alternative functions of these pins such as Serial Peripheral Interface (SPI) bus and pulse width modulation (PWM). Using these packages, you can access most of the functionality of the Gertboard with Python. At the time of writing, the only major functional block that (to our knowledge) cannot be programmed with Python is the Atmel microcontroller.

C Code Overview

To download the Gertboard C software, go to `http://www.element14.com` and search for "Gertboard" using the box at the top of the screen. The link you want will probably be called something like "Application Library for Gertboard". From there you can download the file containing the C code; it will have a name like `gertboard_sw_20120725.zip`. As you can tell by the file extension `.zip`, this is a zip file, which means that it is a compressed collection of different files, all packed together into a single file.

To retrieve the original software, put the file where you want your Gertboard software to end up on your Raspberry Pi computer, then extract the files by typing the following in one of the terminal windows on your RPi (substituting the name of the actual file you have downloaded for the file name we are using in this example):

```
unzip gertboard_sw_20120725.zip
```

A new directory, `gertboard_sw`, will be created. Change to this directory (by typing `cd gertboard_sw`) and list the contents (`ls`). You will see a set of C files and a makefile. C files are software files, but they need to be compiled to run on the processor on your system. In the case of Raspberry Pi, this is an ARM11. The `makefile` tells the computer how to compile the code, so all you need to do it type:

```
make all
```

This compiles the C code into executable programs. To run a program (for example the program `leds` which tests the LEDs), type:

```
sudo ./leds
```

The `sudo` is there because accessing the GPIO ports requires special privileges, and so you need to make an extra effort (by typing `sudo`) to execute it. The `./` before `leds` means that the program `leds` is in the current directory.

Each functional block has at least one test program that goes with it. Each test program is compiled from two or more C files. The file `gb_common.c` (which has an associated header file `gb_common.h`) contains code used by all of the functional blocks on the board. Each test has a C file that contains code specific to that test (thus you will find the `main` function here). Some of the tests use a special interface (for example the SPI bus), and these tests have an additional C file that provides code specific to that interface (these files are `gb_spi.c` for the SPI bus and `gb_pwm` for the pulse width modulator).

In each of the sections about the individual functional blocks, the code specific to the tests for that block is explained. Since all of the tests share the code in `gb_common.c`, an overview of that code will be given here. In order to use the Gertboard via the GPIO, the test code first needs to call `setup_io`. This function allocates various arrays and then calls `mmap` to associate the arrays with the devices that it wants to control, such as the GPIO, SPI bus, PWM (pulse width modulator) etc. The result of this is that it writes to these arrays control the devices or sends data to them, and reads from these arrays get status bits or data from the devices. At the end of a test program, `restore_io` should be called, which undoes the memory map and frees the allocated memory.

Macros

In `gb_common.h`, `gb_spi.h`, and `gb_pwm.h` there are a number of macros that give a more intuitive name to various parts of the arrays that have been mapped. These macros are used to do everything from setting whether a GPIO is used as input or output to controlling the clock speed of the pulse width modulator.

Macro name	T	Explanation	Page no.
`INP_GPIO(n)`	E	activates GPIO pin number *n* (for input)	13
`OUT_GPIO(n)`	E	used after above, sets pin *n* for output	13
`SET_GPIO_ALT(n, a)`	E	used after `INP_GPIO`, select alternate function for pin	29
`GPIO_PULL`	W	set pull code	19
`GPIO_PULLCCLK0`	W	select which pins pull code is applied to	19
`GPIO_IN0`	R	get input values	19
`GPIO_SET0`	W	select which pins are set high	21
`GPIO_CLR0`	W	select which pins are set low	21

Table 2: Commonly used macros, their type, purpose, and location within this manual.

Table 2 shows a summary of the more commonly used macros and gives the page number on which its use is explained in more detail. The T column below gives the 'type' of the macro. This shows how the macro is used. 'E' means that the command is executed, as in:

```
INP_GPIO(17);
```

'W' means that that the command is written to (assigned), as in:

```
GPIO_PULL = 2;
```

'R' means that that the command is read from, as in:

```
data = GPIO_IN0;
```

The macro `INP_GPIO(n)` must be called for a pin number *n* to allow this pin to be used. By default its mode is set up as an input. If it is required that the pin is used for an output, `OUT_GPIO(n)` must be called after `INP_GPIO(n)`.

Python Code Overview

The Python software (along with the bits of this manual describing the Python programs) was written by Alex Eames of Raspi.TV. We are very grateful to him for his help.

This software is written in Python 2.7 and is compatible with all current revisions of the Raspberry Pi computer and Gertboard. It requires sudo user privileges to use the SPI and GPIO ports.

The Python code relies on several packages to access the various functions of the Gertboard. To use the GPIO ports you need to have either RPi.GPIO or WiringPi for Python installed (or both). To use the digital to analogue and analogue to digital converters, which use the SPI bus, you need to install a module called py-spidev. Instructions for how to do this are included with the Python programs.

Downloading the Software

To obtain the Python Gertboard software with a browser or on a computer other than your Raspberry Pi, visit http://raspi.tv/downloads. To get it directly on your internet-connected Raspberry Pi, first change directory to where you want the programs installed, then from the command line type:

```
wget http://raspi.tv/download/GB_Python.zip
```

This should download the small file `GB_Python.zip`. Then type:

```
unzip GB_Python.zip
cd GB_Python
ls
```

The `ls` lists all the files in the directory. Most of them end in `.py` and are Python programs. The file `README.txt` contains (amongst other info) instructions on how to install the packages you need to run the Python programs.

Once you have the necessary packages installed, you can run the programs. For example, if want to run the program `leds-rg.py`, which tests the LEDs (using the RPi.GPIO package, see below) type:

```
sudo python leds-rg.py
```

Why Different Program Versions?

There are two General Purpose Input Output (GPIO) packages for Python: RPi.GPIO and WiringPi for Python. The programs that come in two versions (like `leds-rg.py` and `leds-wp.py`) are using these different packages.

It is desirable to have the both these packages because neither of them yet offers a fully finished set of capabilities (but most of the capabilities are covered between them). RPi.GPIO's weakness is the lack of hardware PWM (pulse width modulation) used with the motor program. WiringPi's weakness is the lack of pull-up facility required to use the buttons. If you want to use the full functionality of the board you will need to install both. For some of the programs (for example, `leds` and `ocol`) no special features are used, and you can use either package. The programs using the RPi.GPIO package are the ones called `*filename*-rg.py`, whilst the ones using the WiringPi for Python package are the ones called `*filename*-wp.py`.

Here is a list of all the Python test programs (at time of writing):

`buttons-rg.py` – buttons program using RPi.GPIO
`leds-rg.py` – leds program using RPi.GPIO
`leds-wp.py` – leds program using WiringPi
`butled-rg.py` – button and LED program using RPi.GPIO
`motor-rg.py` – motor program using software PWM and RPi.GPIO
`motor-wp.py` – motor program using hardware PWM and WiringPi
`ocol-rg.py` – relay switching program using RPi.GPIO
`ocol-wp.py` – relay switching program using WiringPi
`atod.py` – test for analogue to digital converter using SPI with spidev
`dtoa.py` – test for digital to analogue converter using SPI with spidev
`dad.py` – test for both D/A and A/D using SPI with spidev
`potmot.py` – test using A/D and motor using WiringPi and spidev

Buffered I/O, LEDs, and Pushbuttons

There are 12 pins which can be used as input or output ports. Each can be set to behave either as an input or an output, using a jumper. Note that the terms 'input' and 'output' here are always with respect to the Raspberry Pi: in input mode, the pin inputs data to the RPi; in output mode it acts as output from the RPi. It is important to keep this in mind as the Gertboard is set up: an output from the Gertboard is an input to the Raspberry Pi, and so the input jumper must be installed.

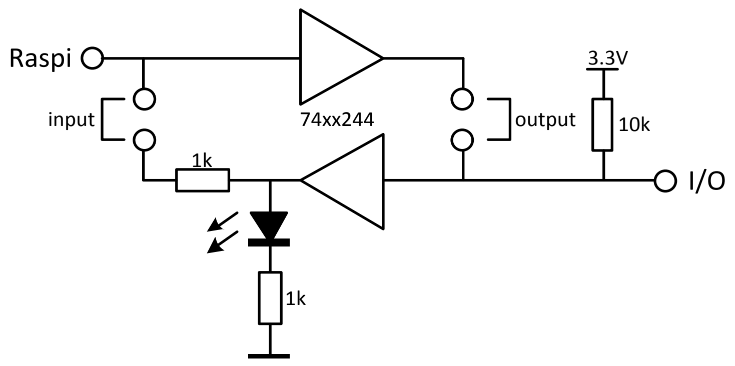


Figure 8: The circuit diagram for I/O ports 4-12.

The triangle symbols in the diagram above represent buffers; they propagate logical values (low and high) in the direction the triangle is pointing. The rectangles are resistors, the black triangle and line with arrows coming out is an LED, and the hollow circles are header pins. In order to make the port function as an input to the Raspberry Pi you install the input jumper (shown in the diagram as a staple shape labelled ‘input’): then the data flows from the ‘I/O’ point to the ‘Raspi’ point. To make the port function as an output, the output jumper must be installed: then the data flows from the ‘Raspi’ point to the ‘I/O’ point. If both jumpers are installed, it won’t harm the board, but the port won’t do anything sensible.

In both the input and output mode the LED will indicate what the logic level is on the ‘I/O’ pin. The LED will be on when the level is high and it will be off when the level is low. There is a third option for using this port: if neither the input nor output jumper is placed the I/O pin can be used as a simple logic detector. The I/O pin can be connected to some other logic point (i.e. one that is either at 0V or 3.3V) and the LED will show if the connect point is high or low.

The resistor on the right side of Figure 8 is a pull-up. If it were not there, the LED would turn off and on with the smallest of electronic changes, for example, when the board is simply touched. Turning the LED on when it is not being driven prevents this seemingly random behaviour and also serves as an indicator that your Gertboard is receiving power properly. Note that if the output jumper is installed but the ‘Raspi’ point is not driven, the random behaviour will return.

There is a series resistor between the input buffer (the left-pointing triangle) and the ‘Raspi’ point. This is to protect the BCM2835 (the processor on the Raspberry Pi) in case the user programs the GPIO as output but leaves the input jumper in place. The BCM2835 input is a high impedance input and thus a 1K series resistor will not produce a noticeable change in behaviour when it is used as input.

Pushbuttons

The Gertboard has three pushbuttons; these are connected to ports 1, 2, and 3. The circuit for these ports is shown in Figure 9. This circuit is essentially the same as that in Figure 8 with the addition of a pushbutton switch and resistor on the left side. When the button is pressed, the ‘Raspi’ point is connected to ground (through a resistor) and so reads low.

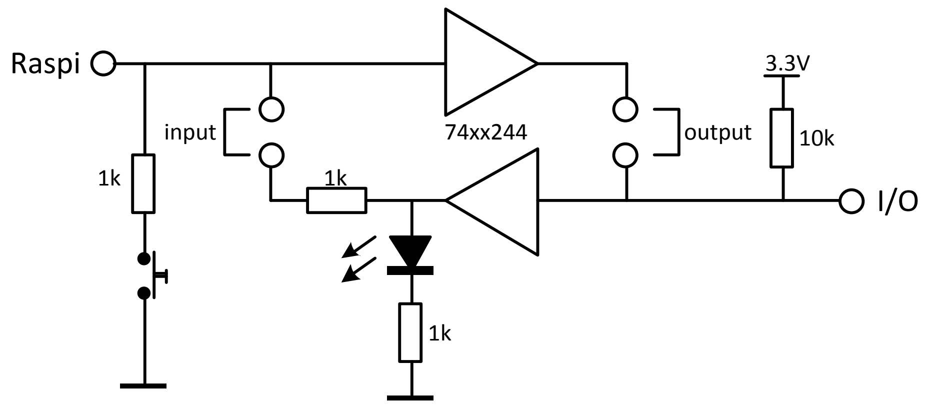


Figure 9: The circuit diagram for I/O ports 1, 2, and 3 (with pushbutton).

In order to use a pushbutton, the input jumper must ***not*** be installed, even if the intention is to use this as an input to the Raspberry Pi. If it is installed, the output of the lower buffer prevents the pushbutton from working properly. To make clear what state each button is in, the output jumper can be installed, and then the LED will now show the button state (LED on means button up, LED off means button down). To use the pushbuttons, a pull-up must be set on the Raspberry Pi GPIO pins used (described below, page 19) so that they are read as high (logical 1) when the buttons are not pressed.

Locating the I/O Ports on the Gertboard

In the functional blocks location diagram (Figure 2 on page 6), the components implementing the buffered I/O are outlined in red. The ICs containing the buffers are U3, U4, and U5 near the centre of the board. The LEDs are labelled D1 to D12; D1 is driven by port 1, D2 by port 2, etc. The pushbutton switches (the silver rectangular devices with black circles) are labelled S1 to S3; S1 is connected to port 1 and so on.

The pins corresponding to 'Raspi' in Figure 8 and Figure 9 above are B1 to B12 on the J3 header above and to the right of the words 'Raspberry Pi' on the board (B1 to B3 correspond to the 'Raspi' points in Figure 9, and B4 to B12 correspond to the 'Raspi' points in Figure 8). They are called 'Raspi' because these are the ones that are usually connected to the pins in header J2, which are directly connected to the pins in J1, and which are then finally connected to the GPIO pins on the Raspberry Pi. The pins corresponding to the 'I/O' point on the right of the circuit diagrams above are BUF1 to BUF12 in the (unlabeled) single row header at the top of the Gertboard.

On the Gertboard schematic, I/O buffers are on page A-2. The buffer chips U3, U4, and U5 are clearly labelled. It should be clear that ports 1 to 4 are handled by chip U3, ports 5 to 8 by chip U4, and ports 9 to 12 by chip U5. The 'Raspi' points in the circuit diagrams above are shown as the signals BUF_1 to BUF_12 on the left side of the page, and the 'I/O' points are BUF1 to BUF12 to the right of the buffer chips. The input jumper locations are the blue rectangles labelled P1, P3, P5, P7, etc to the left of the buffer chips, and the output jumper locations are the blue rectangles labelled P2, P4, P6, P8, etc, to the right of the buffer chips. The pushbutton switches S1, S2, and S3 are shown separately, on the right side of the page near the middle. Below the pushbuttons, the pull-up resistors are shown.

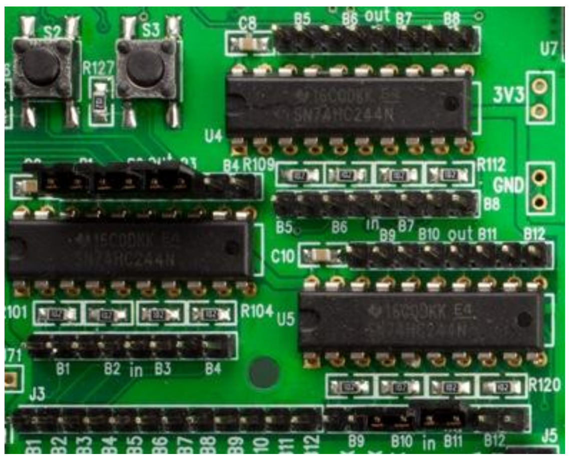
The buffered I/O ports can be used with (almost) any of the GPIO pins; they just have to be connected using straps. So for example, if you want to use port 1 with GPIO17 a strap is placed between the B1 pin in J3 and the GP17 pin in J2. Beware that the pushbuttons ***cannot*** be used with GPIO0 or GPIO1 (GP0 and GP1 in header J2 on the board) as those two pins have a 1800Ω pull-up resistor on the Raspberry Pi. When the button is pressed the voltage on the input will be

$$3.3V \times \frac{1000\Omega}{1000\Omega + 1800\Omega} = 1.2V$$

This is not an I/O voltage which can be reliably seen as low.

The output and input jumper locations are above and below the U3, U4, and U5 buffer chips. The input jumpers need to be placed on the headers below the chips (shown on the board with the ‘in’ text; they are separated from the chip they go with by four small resistors), and the ‘output’ jumpers need to be placed on the headers above the chips (with the ‘out’ text). If viewed closely (it is clearer on the blue and grey diagram), it is possible to see that each row of 8 header pins above and below the buffer chips is divided up into 4 pairs of pins. The pairs on U3 are labelled B1 to B4, the ones on U4 are B5 to B8, and the ones on U5 are B9 to B12. The B1 pins are for port 1, B2 for port 2, etc.

To use port *n* as an input (but not when using the pushbutton, if *n* is 1, 2, or 3), a jumper is installed over the pair of pins in B*n* in the row marked ‘in’ (below the appropriate buffer chip). To use port *n* as an output, a jumper is installed over the pair of pins in B*n* in the row marked ‘out’ (above the appropriate buffer chip).



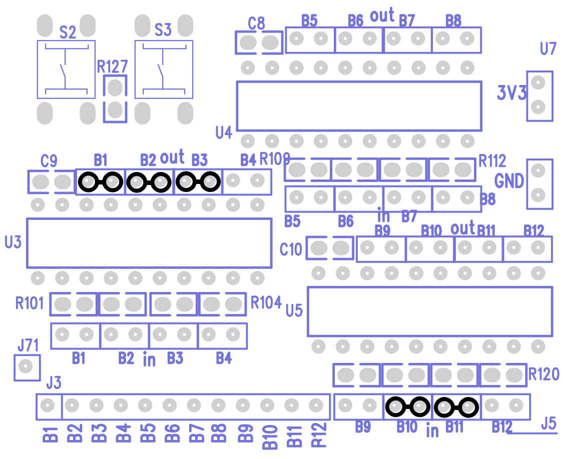


Figure 10: Example of port configuration where ports 1 to 3 are set to be outputs and ports 10 and 11 are set to be inputs.

As a concrete example, in Figure 10 ports 1, 2, and 3 are configured for output (because of the jumpers across B1, B2, and B3 on the ‘out’ side of chip U3). Ports 10 and 11 are configured for input (because of the jumpers across B10 and B11 on the ‘in’ side of U5). Figure 10 also demonstrates why we use diagrams instead of photos to show you how to wire up your Gertboard. The input jumper on B10 is almost impossible to see because it is shorter than the others.

In the test programs, the required connections are printed out before the tests are started. The input and output jumpers are referred to in the following way: U3-out-B1 means that there is a jumper across the B1 pins on the ‘out’ side of the U3 buffer chip. So the 5 jumpers in the picture above would be referred to as U3-out-B1, U3-out-B2, U3-out-B3, U5-in-B10, and U5-in-B11.

Testing the Pushbuttons

There are test programs for the buttons in both C and Python. The C version is called `buttons`, and the Python version is called `buttons-rg.py`. To run these tests, the Gertboard must be set up as in the photo in Figure 11. (The wiring diagram in Figure 12 shows the same thing more clearly.) There are straps connecting pins B1, B2, and B3 in header J3 to pins GP25, GP24, and GP23 in header J2 (respectively). Thus GPIO25 will read the leftmost pushbutton, GPIO24 will read the middle one, and

GPIO23 will read the rightmost pushbutton. The jumpers on the ‘out’ area of U3 (U3-out-B1, U3-out-B2, U3-out-B3) are optional: if they are installed, the leftmost 3 LEDs will light up to indicate the state of the switches.

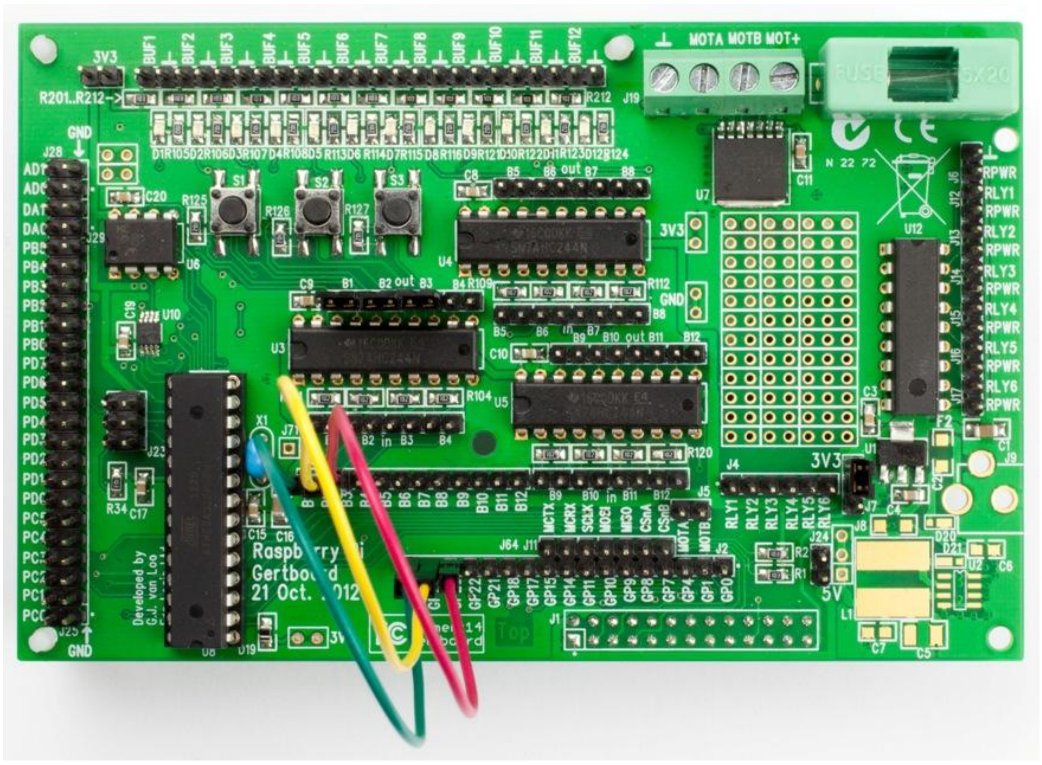


Figure 11: Photo showing connections for the buttons test. Whilst the image above is clear, it isn’t very good at showing exactly how the straps and jumpers are placed.

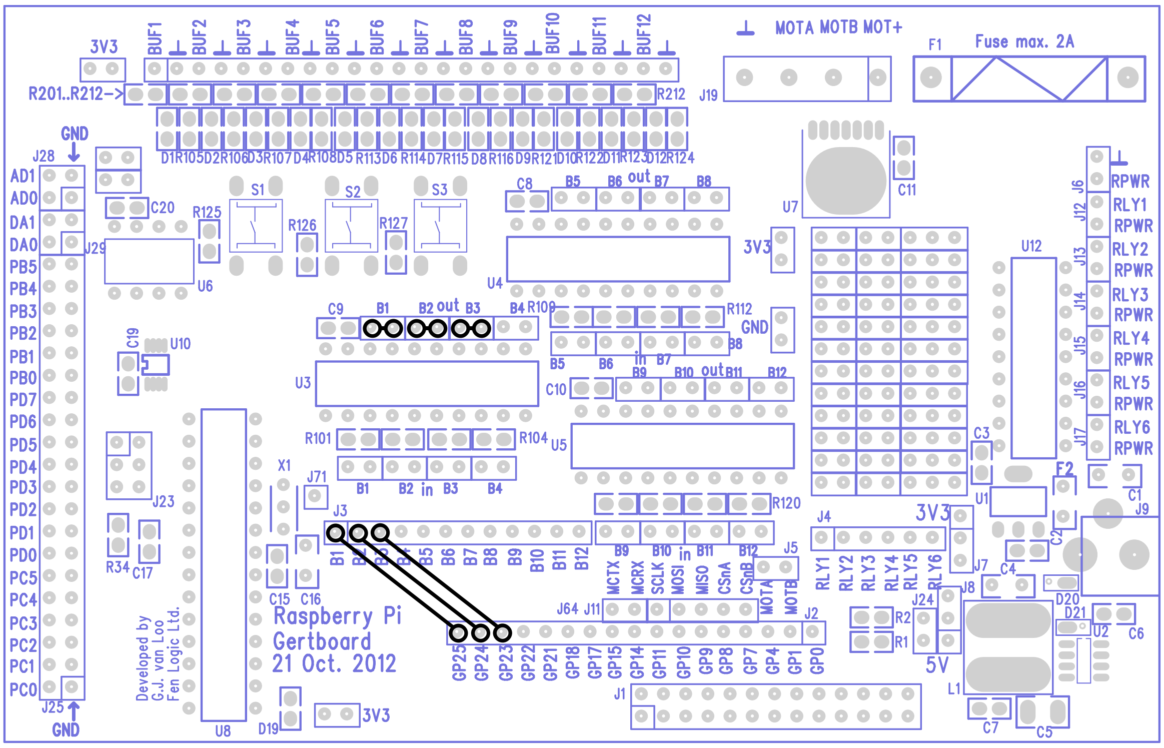


Figure 12: Wiring diagram for buttons test. This type of diagram is much more effective at showing the connections that need to be made, so from now on, we will use these diagrams to show wiring arrangements.

In the diagram in Figure 12, black circles show which pins are being connected, and black lines between two pins indicate that jumpers (if they are adjacent) or straps (if they are further apart) are used to connect them.

Buttons Test in C

The code specific to the `buttons` test is `buttons.c`. In the `main` routine, the connections required for this test are firstly printed to the terminal (a text description of the wiring diagram above). When the user verifies that the connections are correct, `setup_io` (described on page 12) is called to get everything ready.

`setup_gpio` is then called, which gets GPIO pins 1 to 3 ready to be used as pushbutton inputs. It does this by first using the macro `INP_GPIO(n)` (where *n* is the GPIO pin number) to select these 3 pins for input.

Then pins are required to be pulled high: the buttons work by dropping the voltage down to 0V when the button is pressed, so it needs to be high when the button is not pressed. This is done by setting `GPIO_PULL` to 2, the code for pull-up. Should it ever be required, the code for pull-down is 1. The code for no pull is 0; this will allows this pin to be used for output after it has been used as a pushbutton input. To apply this code to the desired pins, set `GPIO_PULLCCLK0 = 0X03800000`. This hexadecimal number has bits 23, 24, and 25 set to 1 and all the rest set to 0. This means that the pull code is applied to GPIO pins 23, 24, and 25. A `short_wait` allows time for this to take effect, and then `GPIO_PULL` and `GPIO_PULLCLK0` are set back to 0.

Back in the `main` routine, a loop is entered in which the button states are read (using macro `GPIO_IN0`), grabbing bits 23, 24, and 25 using a shift and mask logical operations, and, if the button state is different from before, it is printed out in binary: up (high) is printed as '1' and down (low) is printed as '0'. This loop executes until a sufficient number of button state changes have occurred.

After the loop, `unpull_pins` is called, which undoes the pull-up on the pins, then call `restore_io` in `gb_common.c` to clean up.

Buttons Test in Python

This program (`buttons-rg.py`) is only available using the RPi.GPIO package at the moment because the pull-up facility in WiringPi for Python is not yet available. The buttons cannot be used without this facility.

First the program imports the RPi.GPIO module it needs to handle GPIO control. Then the command `GPIO.setmode(GPIO.BCM)` sets up the BCM numbering scheme for the pins. The result of this is that the pin numbers in the Python code are the numbers that the BCM2835 (the Raspberry Pi processor) uses to refer to the pins. Otherwise, the numbers in the Python code will refer to the pin numbers in the J1 header (which is the same as their placement in the P1 header on the RPi).

The next two lines set up ports 23-25 with pull-up:

```
for i in range(23,26):
    GPIO.setup(i, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Note that because of the way Python `for` loops work, the end point of the loop must be set to one beyond the last item in the range that you want to cover. So in order to set port 25, the end point of the loop command must be set to 26.

Next, the wiring instructions appear on the screen. Once the user confirms that the wiring is ok, the initial values for variables `button_press` and `previous_status` are set.

Next a `while` loop runs until 19 button presses/releases have occurred. For each iteration of this loop, the status of each button is read and, if pressed, a 1 is stored in a list variable called `status_list`. If not pressed, a 0 is stored.

Then the `status_list` values are all checked against the `previous_status`. If there's a change, this line

```
   if current_status != previous_status:
```

executes a small section of the program that displays the new values on the screen, increments the value of the iterator `button_press`**,** and the loop starts again from the top. Now it's one button press closer to the end-point of 19.

The `while` loop is enclosed in a `try` block:

```
try:
   <while block>
except KeyboardInterrupt:
```

This enables the program to reset the ports if CTRL+C is pressed to terminate the program early.

If the program ends normally, after 19 button presses, the ports will be reset anyway. If we hadn't included the `try: except:` block, a keyboard interrupt (ctrl-c) would close the program but leave the ports open. This would give errors if you tried to use the ports again.

Suggested tweaks to experiment with. Try changing these one at a time and see what they do...

- `while button_press < 20:` – change the 20 to some other number
- `button_press += 1` – change the 1 to some other number

Testing the LEDs

The C test program for the LEDs is called `leds`. The Python versions are `leds-rg.py` and `leds-wp.py`. To set up the Gertboard to run this test, see the wiring diagram in Figure 13. Every I/O port is connected up as an output, so all the 'out' jumpers (those above the buffer chips) are installed. Straps are used to connect the following (where all the 'GP' pins are in header J2 and all the 'B' pins are in header J3): GP25 to B1, GP24 to B2, GP23 to B3, GP22 to B4, GP21 to B5, GP18 to B6, GP17 to B7, GP11 to B8, GP10 to B9, GP9 to B10, GP8 to B11, and GP7 to B12. In other words, the leftmost 12 'GP' pins are connected to the 'B' pins, except that GP14 and GP15 are missed out: they are already set to UART mode by Linux, so it's best if they are not touched.

If there aren't enough jumpers or straps to wire these connections all up at once, don't worry. Just wire up as many as possible and run the test. Once it's finished the straps/jumpers can be moved and the test can be run again. Nothing bad will happen if you write to an unconnected pin.

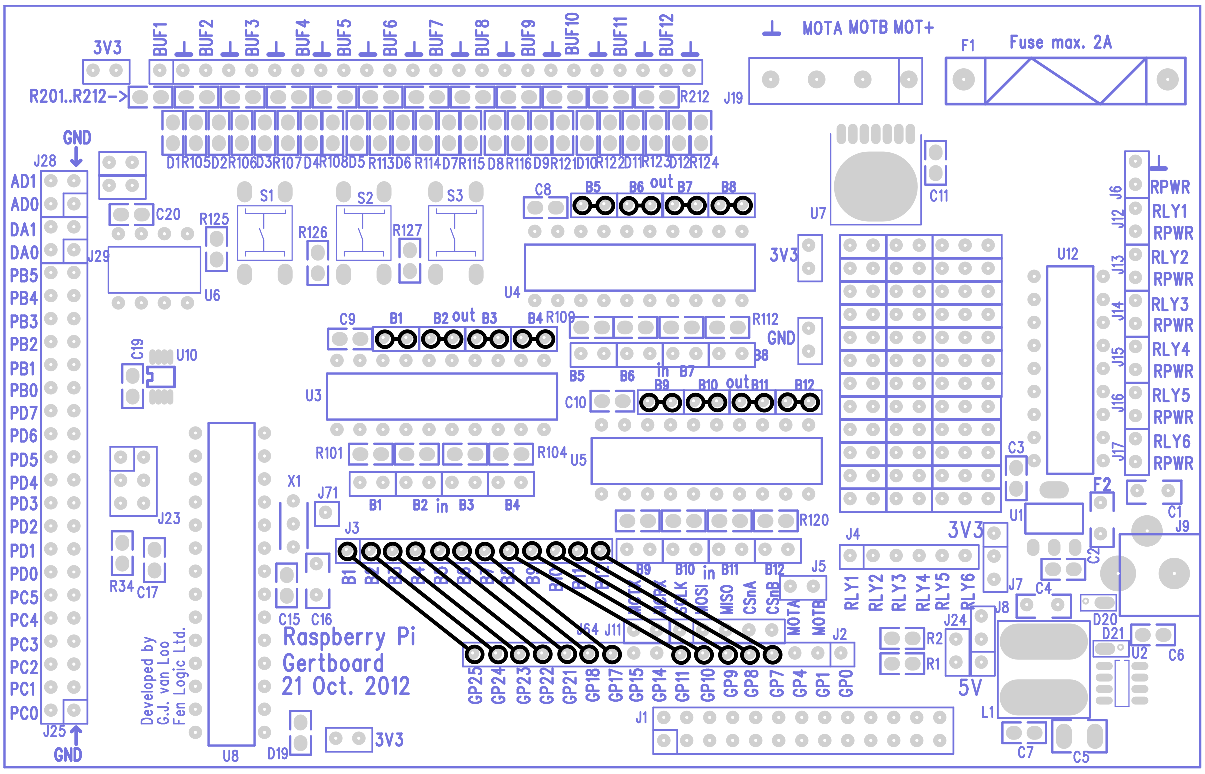


Figure 13: The wiring diagram for the LED test program

LEDs Test in C

The test code in `leds.c` first calls `setup_io` to get everything ready. Then `setup_gpio` is called, which prepares 12 GPIO pins to be used as outputs (as all 12 I/O ports will require controlling). All of the GPIO signals except GPIO 0, 1, 4, 14, and 15 are used. To set them up for output, first call `INP_GPIO(`*n*`)` (where *n* is the GPIO pin number) for each of the 12 pins to activate them. This also sets them up for input, so then call `OUT_GPIO(`*n*`)` afterwards for each of the 12 pins to put them in output mode.

LEDs are switched on using the macro `GPIO_SET0`: the value assigned to `GPIO_SET0` will set GPIO pin *n* to high if bit *n* is set in that value. When a GPIO pin is set high, the I/O port connected to that pin goes high, and the LED for that port turns on. Thus, the line of code "`GPIO_SET0 = 0x180;`" will set GPIO pins 7 and 8 high (since bits 7 and 8 are set in the hexadecimal number 0x180). Given the wiring setup above, ports 11 and 12 will go high (because these are the ports connected to GP7 and GP8), and thus the rightmost two LEDs will turn on.

To turn LEDs off, use macro `GPIO_CLR0`. This works in a similar way to `GPIO_SET0`, but here the bits that are high in the value assigned to `GPIO_CLR0` specify which GPIO ports will be set *low* (and hence which ports will be set low, and which LEDs will turn off). So for example, given the wiring above, the command "`GPIO_CLR0 = 0x100;`" will set GPIO8 pin low, and thus turn off the LED for port 11, which is the port connected to GP8. (In `leds.c` the LEDs are always all turned off together, but they don't have to be used this way.)

The test program flashes the LEDs in three patterns. The patterns are specified by a collection of global arrays given values using an initializer. The number in each of the arrays says which LEDs will be turned on at that point in the pattern – so, pattern value is submitted sequentially to produce the changing pattern, switching all the LEDs off between successive pattern values. Each pattern is run through twice. The first pattern lights the LEDs one at a time in sequence, left to right. The second pattern does the same but when it reaches the rightmost LED, it then reverses direction and lights them in sequence right to left. The third pattern starts at the left end and at each step switches on one more LED until they are all lit up, then starting at the left it switches them off one by one until they are all off.

Finally, the test program switches off all the LEDs and then finally calls `restore_io` to clean up all the LEDs to a predictable final state.

LEDs Test in Python

This test is available for both RPi.GPIO and WiringPi for Python. The only differences between the `leds-rg.py` and `leds-wp.py` programs are the ways the GPIO ports are driven and cleaned up, and the Raspberry Pi board revision checker.

In the programs, first the Raspberry Pi board revision is checked and, based on the result, the correct ports for the LEDs are defined in a list called `ports`. Since we need to go both forwards and backwards, we then make a copy of the list, name it `ports_rev` and reverse it. Then we set up the ports for output by iterating through the list `ports`.

After that, we define the main "engine" of this script, the function `led_drive()`, which requires three arguments (`reps, multiple, direction`).

1. `reps` defines how many times to run the process (1 or 3 in this demo)
2. `multiple` defines whether or not to switch an led off before switching on the next one (1 = leave it on, i.e. multiple LEDs lit)
3. `direction` defines whether to use the forward or reverse ports list (`ports` for forwards, `ports_rev` for reverse)

There are eight calls to the `led_drive()` function, so we've saved a lot of program lines by reusing the same code multiple times. All the calls to the `led_drive()` function are enclosed in a `try:` `except:` block. This ensures that if you exit via CTRL-C, the GPIO ports will be reset or "cleaned up". This avoids false warnings, the next time you want to use them, that the ports are already in use by another program.

Suggested tweaks to experiment with. Try changing these one at a time and see what they do...

- `led_drive(3, 0, ports)` – change the 3
- `led_drive(3, 0, ports)` – change `ports` to `ports_rev`
- `led_drive(3, 0, ports)` – change the 0 to a 1

Testing I/O

Our two examples so far have only used the ports to access the pushbuttons and LEDs. The next example, called `butled` (for BUTton LED) in C, or `butled-rg.py` (in Python), will show one of the ports serving just as an input port. The idea is that one port (along with its button) is used to generate a signal, and software then sends that signal to another port which it is used as just an input. We read both ports in and print them on the screen.

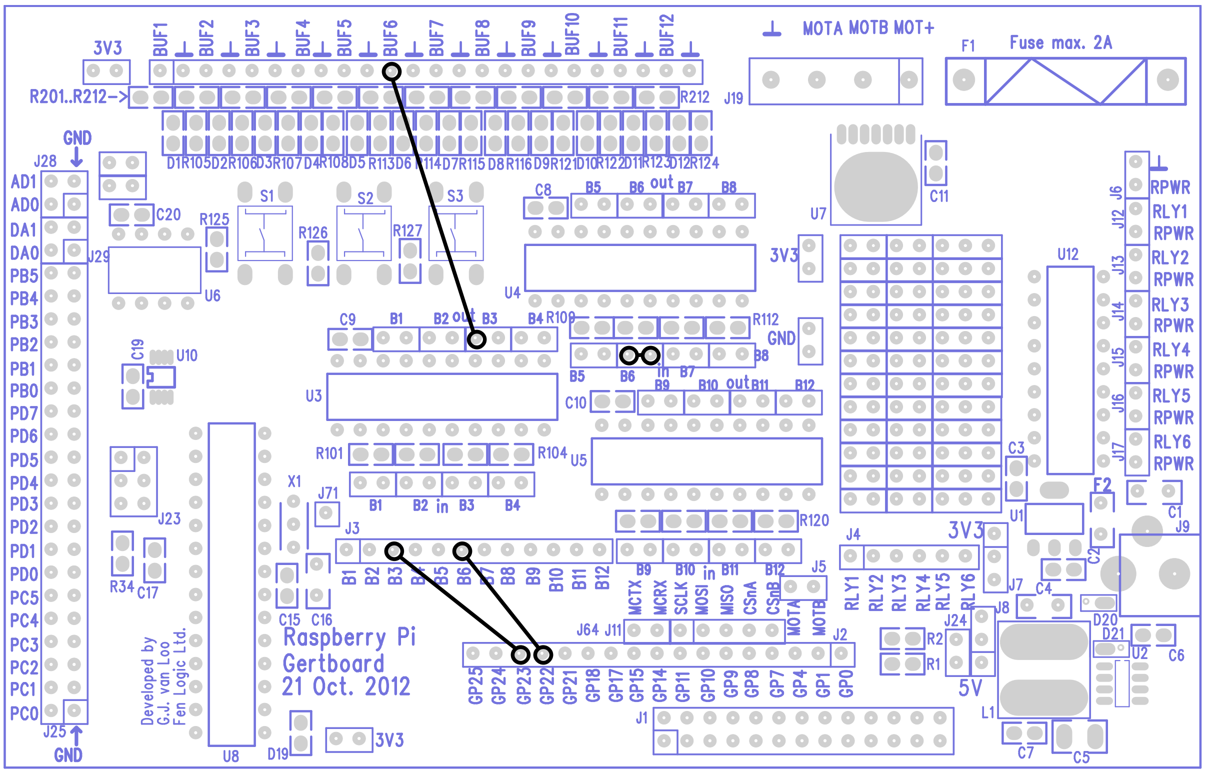


Figure 14: The wiring diagram for test program butled which detects a button press, and then displays that button state on the screen and on an LED.

The wiring for this test is shown above. Pin GPIO23 controls I/O port 3, and GPIO22 controls I/O port 6, so GP23 in header J2 is connected to pin B3 in header J3, and GP22 is connected to B6. Now for the interesting part. The pushbutton on port 3 is going to be used here, but the LED for port 3 should not be used, so therefore the output jumper for port 3 (which would be placed at U3-out-B3) is not installed.

Looking at the schematic on page A-2, it is clear that the output buffer for port 3 goes to pin 14 of buffer chip U3. This is connected to pin 1 of U3-out-B3 (shown as P6 in the schematic). It is not obvious which header pin on the board is pin 1, but as it's connected (on the circuit board) to pin 14 of the chip, we can guess that it's the one right above pin 14. A simple experiment shows that this is indeed the right pin, so we connect this pin to the BUF6 pin at the top of the board. This allows the switch to generate a signal which is then sent to port 6. A jumper is installed across U4-in-B6 to allow that signal to be input from the board. The value of the switch from port 3 is also read in, and these two should be the same (most of the time).

Butled Test in C

In `butled.c` we use `INP_GPIO` to set GPIO22 and GPIO23 to input and `GPIO_PULL` and `GPIO_PULLCLK0` to set the pull-up on GPIO23. This is described in more detail on page 19, in the `buttons` test. Then the GPIO values are repeatedly read in, and the binary values of GPIO22 and GPIO23 are printed out (with GPIO23 first), if they have changed since the last cycle. So if '01' is displayed on the monitor, we can see that GPIO23 is low and GPIO22 is high. (Note that the LED for port 6, labelled D6, should be off when switch 3 is pressed and on when switch 3 is up.)

Now, if the values for GPIO22 and GPIO23 are always the same, '00' and '11' will only ever be printed out. But occasionally we see '01' or '10'. Note that when this occurs, the first digit changes to the new value, and then immediately afterwards the second digit changes. But the values of both GPIO22 and GPIO23 are read out simultaneously, so why do we ever have different values on the two GPIO pins? The answer is that signal from the pushbutton (which is connected to GPIO23) takes a small amount of time to propagate through the buffers to get to GPIO22. Sometimes we take a reading after GPIO23 has changed, but insufficient time has passed for GPIO22 to change state and follow it!

Butled Test in Python

This program (`butled-rg.py`) also only works with RPi.GPIO at the moment. It is a very similar program to buttons, but with two less buttons in use and an LED wired into the circuit, which lights on button press.

Two GPIO ports are used. One (23) is pulled HIGH for the button and the other (22) is configured as a regular input. In the main `while` loop, the program polls both ports and, if there is a change, increments the variable `button_press` and displays the new values on the screen. The strap from U3-out-B3 pin 1 to BUF6 in the top header connects the LED to the button, causing it to light and the other input port (22) to go HIGH when the button is pressed. The values are displayed on the screen as with the buttons program.

Now, if the values for GPIO22 and GPIO23 are always the same, '00' and '11' will only ever be printed out. But occasionally we see '01' or '10'. Note that when this occurs, the second digit changes to the new value, and then immediately afterwards the first digit changes. This is opposite of the behaviour of the C program, where the first digit sometimes changes before the second. Why is it acting differently? In the Python code, the values are read in by this line of code:

```
    status_list = [str(GPIO.input(23)),str(GPIO.input(22))]
```

This causes GPIO23 to be read before GPIO22. If the button is pressed or released between these two reads, GPIO23 will still have the old value, but the new value will be read from GPIO22. The new value won't be read from GPIO23 until the next time through the `while` loop.

Open Collector Driver

The Gertboard uses six ports of a ULN2803a to provide open collector drivers. These are used to turn off and on devices, especially those that are powered by an external power supply and need a different voltage or higher current than that available on the Gertboard. Basically, an open collector connects the ground side of an external circuit to ground on the board, thus giving the circuit power. The ULN2803A can withstand up to 50V and drive 500mA on each of its ports. Each driver has an integrated protection diode (the uppermost diode in the circuit diagram in Figure 15).

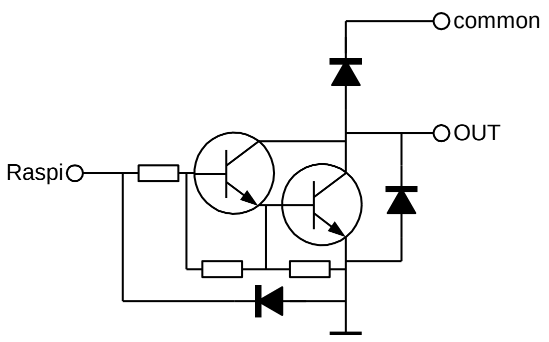


Figure 15: Circuit diagram of each open collector driver.

The 'common' pin is, as the name states, common for all open collector drivers. It is not connected to anything else on the Gertboard. As with all devices the control for the open collector drivers (the 'Raspi' point) can also be connected to the ATmega controller to, for example, drive relays or motors.

The open collector drivers are in the schematics on page A-3.

In the Gertboard functional block diagram (Figure 2 on page 6), the area containing the components for the open collector drivers are outlined in yellow. The pins corresponding to 'Raspi' in Figure 15 are RLY1 to RLY6 pins in the J4 header; the pins corresponding to 'common' are the ones marked RPWR in the headers on the right edge of the board; and the pins corresponding to 'OUT' are the RLY1 to RLY6 pins in the headers J12 to J17. How these are then used is demonstrated by the test wiring and code examples.

Testing the Open Collector Drivers

The C program `ocol` (for open collector) allows the functional testing of the open collector drivers. The Python version comes in two flavours, `ocol-rg.py` and `ocol-wp.py`.

We needed something for the driver to switch on and off, so we created a little circuit consisting of two large LEDs and a resistor in series. (This is the small circuit to the right of the Gertboard in Figure 16.) Once connected, the forward voltage across each of these LEDs is a little above 3V, so we used a 9V battery as a power supply and calculated a series resistance of around about 90Ω to set a suitable current flow through the LEDs. You can of course use any circuit you like to test this; just make sure that the voltage of your power supply is appropriate for your circuit (and that it is within the 50V, 500mA limit of the driver).

To turn the circuit off and on using the open collector driver (say you want to use driver 1), first check that your circuit works with the external power supply you are using. Then, leave the positive side of your circuit attached to the positive terminal of the power supply, but in addition connect it to one of the RPWR pins in the headers on the right edge of the board (they are all connected together). Disconnect the ground side of the circuit from the power supply and connect it instead to RLY1 in header J12 on the right of the board. Attach the ground terminal of the power supply to any GND or ⊥ pin on the board. Now, we need a signal to control the driver. For the `ocol` test we use GPIO4 to control the open collector (you could of course use any logic signal), so connect GP4 in header J2 to RLY1 in J4. (To test a different driver, say *n*, connect the ground side of the circuit up to RLY*n* in the headers on the right of the board and connect GP4 in header J2 to RLY*n* in J4.)

Now, when RLY1 in J4 is set low, the circuit doesn't receive any power and thus is off. When RLY1 in J4 goes high, the open collector driver uses transistors to connect the 'ground' side of the circuit to the ground on the board, and since this is connected to the ground terminal on the power supply, the power supply powers your circuit: it is just turned off and on by the open collector driver.

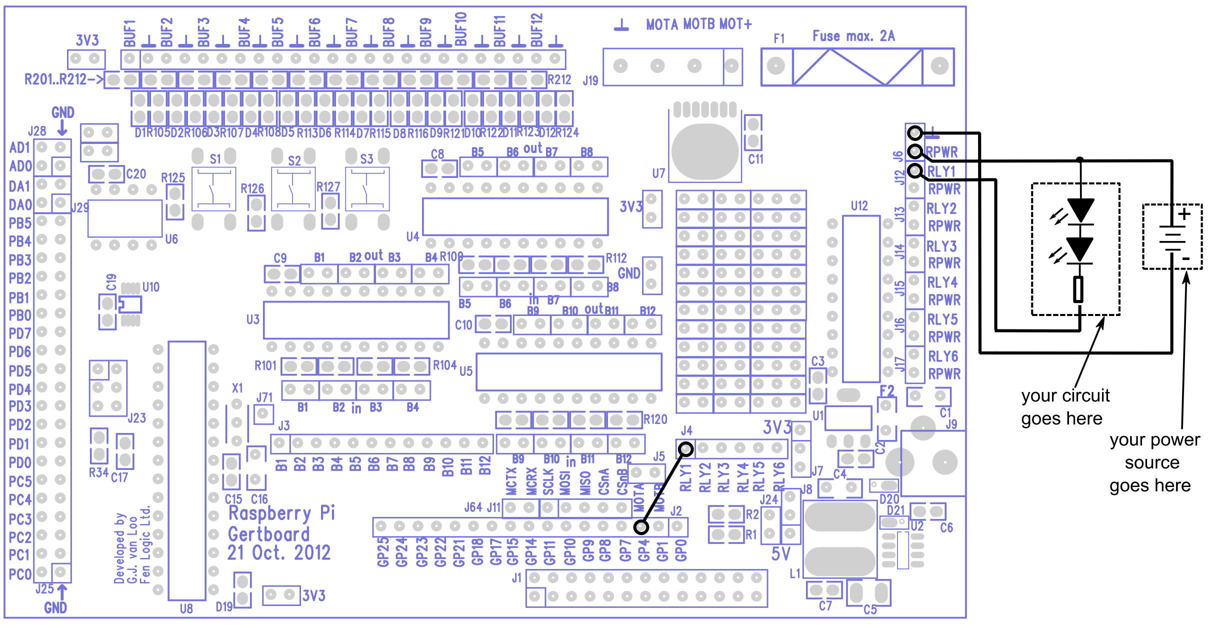

your circuit goes here
your power source goes here
Raspberry Pi Gertboard 21 Oct. 2012
Developed by G.J. van Loo Fen Logic Ltd.

Figure 16: Wiring diagram to test the open collector drivers. On the right is a small test circuit made up of two LEDs in series with a 90 Ω resistor, and a 9V battery acting as power supply.

You may wonder why you need to connect the positive terminal of the power supply to the open collector driver (via the RPWR pin). The reason for this is that if the circuit happens to contain an component that has electrical inductance, for example a motor or a relay, when the power is turned off this inductance can cause the voltage on RLY*n* pin on the right of the board to quickly rise to a higher voltage than the positive terminal of the power supply, dropping quickly afterwards. The chip itself has an internal diode connecting the RLY*n* pin to RPWR (this is the diode at the top of Figure 15). This allows current to flow to the top (positive side) of your circuit, allowing the energy to dissipate and preventing damage.

Open Collector Test in C

The `ocol` test is very simple. First, it prints out the connections required on the board (and with your external circuit and power supply), and then it calls `setup_io` to get the GPIO interface ready to use and `setup_gpio` to set pin GPIO4 to be used as an output (using the commands `INP_GPIO(4); OUT_GPIO(4);` as described on page 13). Then in it uses `GPIO_SET0` and `GPIO_CLR0` (described on page 21) to set GPIO4 high then low 10 times. Note: the test asks which driver should be tested, but it only uses this information to print out the connections that need to be made. Otherwise it ignores your response.

Open Collector Test in Python

The two open collector programs `ocol-wp.py` and `ocol-rp.py` are identical apart from the GPIO system used. In the programs, the function `which_channel()` handles the user channel selection. Once a suitable input is obtained, correct wiring instructions are displayed. Once **enter** is pressed, to confirm the wiring is done, the program switches the chosen open collector port on and off ten times with a 0.4 second delay between each change. At the end, or on (CTRL-C) keyboard interrupt, the GPIO ports are reset.

Suggested safe tweaks to experiment with. Try changing these one at a time and see what they do...

- vary the 0.4 in `sleep(0.4)`
- change the 10 in `for i in range(10):`
- see what happens if you try to tell it you want to use driver 7 (when using the program)
- change the error message displayed when choosing the wrong port number to an "alternative" one of your choice

Motor Controller

The Gertboard has a ROHM BD6222HFP motor controller. The motor controller is for brushed DC motors and can handle a maximum voltage of 18V and max current of 2A.

The controller has two input pins, A and B (labelled MOTA and MOTB on the board). The pins can be driven high or low, and the motor responds according to the table below. The speed of the motor can be controlled by applying a pulse-width-modulated (PWM) signal to either the A or B pin.

A	B	Motor action
0	0	no movement
0	1	rotate one way
1	0	rotate opposite way from above
1	1	no movement

Table 3: Truth table showing the behaviour of the motor controller under different logic combinations.

The motor controller IC has internal temperature protection. Current protection is provided by a fuse on the Gertboard. The motor controller is in the schematics on page A-4.

On the Gertboard functional block diagram (Figure 2 on page 6), the area containing the components for the motor controller are outlined in purple. The motor controller and screw terminals are near the top of the board, and there are two pins for the control signals in J5, a small header just above GP4 and GP1 in header J2. The MOTA and MOTB pins in J5 are the ***inputs*** to the motor controller – these are digital signals (low and high). The screw terminals at the top of the board labelled MOTA and MOTB are the ***outputs*** of the motor controller: they actually provide the power to the motor. The motor will probably need more power (a higher voltage or current) than that provided by the Gertboard. The screw terminals at the top labelled MOT+ and ⊥ allow the connection of an external power supply to provide this: the motor controller directs this power to the MOTA and MOTB screw terminals, modulating it according to the MOTA and MOTB inputs in J5.

If you just want to turn the motor off and on, in either direction, this is achieved by simply choosing two of the GPIO pins and installing straps between them to the MOTA and MOTB motor controller inputs. Then, to control the motor, the pins are set high or low as in Table 3. To control the *speed* of the motor however, pulse width modulation (PWM) is required. This is a device that outputs a square wave that flips back and forth from on to off very rapidly, as shown in Figure 17.

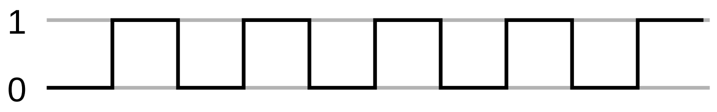


Figure 17: An example of a PWM output. Here the output is on for 50% of the time, so it has a duty cycle of 50%.

With a PWM, you can control the amount of time the output is high vs. when it is low. This is called the duty cycle and is expressed as a percentage. Figure 17 above shows a 50% duty cycle; the one in Figure 18 below is 25%.

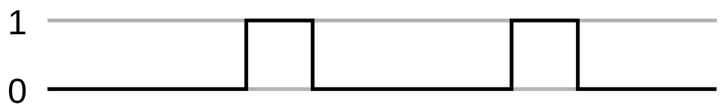


Figure 18: In this PWM example, the duty cycle is 25%.

There is a PWM in the BCM2835 (the Raspberry Pi processor), and its output can be accessed via GPIO18 (it is alternate function 5). If this is connected to one of the motor controller inputs (MOTA has been used in our `motor` test), and the other motor controller input (MOTB in our test) is set to a steady high or low, the speed and direction of the motor can be controlled.

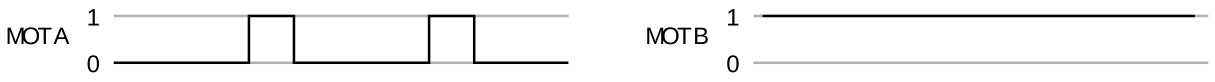


Figure 19: The motor direction is set by MOTB. Whilst MOTA has a duty cycle of 25%, the motor receives power whenever MOTA and MOTB are different, thus it receives power for 75% of the time.

For example, in Figure 19 above we are alternating between A low/B high and A high/B high (the second and fourth lines of the table above). When A is low, the motor will receive power making it turn one way; when A is high it will not receive power. The end result for the 25% duty cycle shown here is that the motor will turn one way at roughly ¾ speed.

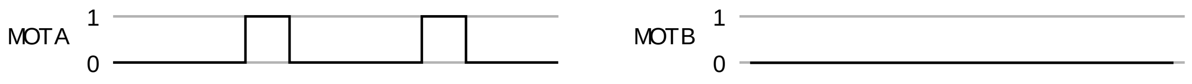


Figure 20: In this example, the motor will run in the opposite direction at around 25% speed.

If on the other hand you set MOTB low, as in Figure 20 above, then when A is high the motor will receive power making it turn in the other direction, and when A is low the motor will not receive power. The result for the 25% duty cycle is that it will turn in the other direction at about ¼ speed.

Testing the Motor Controller

The C test program for the motor controller is called `motor`. In Python there are two versions, `motor-rg.py` and `motor-wp.py`. To set up Gertboard for this, connect GP17 in J2 to the MOTB pin (the MOTB pin in J5, not the one at the top of the board), and GP18 to MOTA in J5. The motor leads need to be connected to the MOTA and MOTB screw terminals at the top of the board, and the power supply for the motor needs to be connected to the MOT+ and ⊥ screw terminals. This is shown in Figure 21.

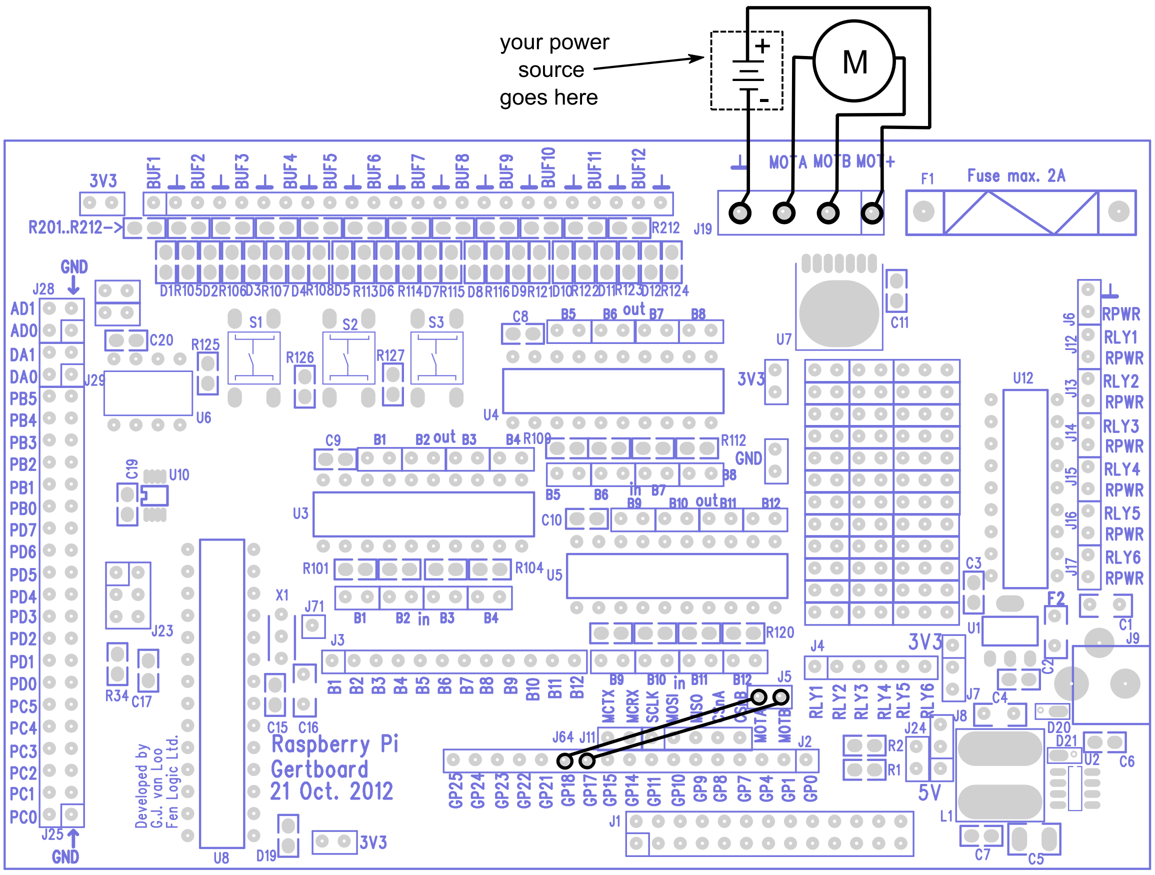

your power source goes here
+
-
M
MOTA
MOTB
MOT+
Fuse max. 2A
Raspberry Pi Gertboard 21 Oct. 2012
GP18
GP17
J5
MOTA
MOTB

Figure 21: The wiring diagram for the test program motor.

Motor Test in C

The PWM is controlled by a memory map, like the GPIO and SPI bus. This memory map is part of the `setup_io` function in `gb_common.c`, so that is whether the PWM is used or not. Further setup code is found in, `gb_pwm.c`, with an associated header file `gb_pwm.h`. The function `setup_pwm` in `gb_pwm.c` sets the speed of the PWM clock, and sets the maximum value of the PWM to 1024: this is the value at which the duty cycle of the PWM will be 100%. It also makes sure that the PWM is off. The two routines `set_pwm0` and `force_pwm0` set the value that controls the duty cycle for the PWM. `set_pwm0` sets the value (first checking that it is between 0 and 1024), but as there are only certain points in the PWM cycle where a new value is picked up, if a second value is written again quickly the first will have no effect. The `force_pwm0` routine takes two arguments, a new value and a new mode. It disables the PWM, then sets the value, then re-enables it with the given mode setting, with delays in strategic places to allow the new values to be picked up. The `pwm_off` routine simply disables the PWM.

The code for the **`motor`** program is in `motor.c`. In the `main` routine, first the connections that must be made on the board to run this program are printed out, then call `setup_io` to get the GPIO interface ready for use. `setup_gpio` is then called to set GPIO18 up for use as the PWM output and GPIO17 up for normal output. For the latter, both `INP_GPIO` and `OUT_GPIO` are used, see page 13 for more info. To set up GPIO18, first use `INP_GPIO(18)` to activate the pin. One of the alternate functions for GPIO18 is to act as the output for the PWM; this is alternative 5. Thus use the macro `SET_GPIO_ALT(18, 5)` to select this alternate use of the pin. (See table Table 6-31 from the BCM2835 datasheet, or the online version at http://elinux.org/RPi_BCM2835_GPIOs, for more details about alternative functions of the GPIO pins. For a summary of the alternate function of GPIO pins used on the Gertboard, see Table 1 on page 11.)

We set the output of GPIO17 low (to make sure that the motor doesn't turn) and then initialize the PWM by calling `setup_pwm`. We enable the PWM by setting the mode to `PWM0_ENABLE` using `force_pwm0`. Since GPIO17 (motor controller B input) is set low, when the duty cycle on the PWM (motor controller A input) is high enough, the motor will turn the 'opposite way' as described in the motor table on page 27.

A loop now starts where the PWM is started, first with a very low duty cycle (because the value passed to `set_pwm0` is low), then gradually increasing this to the maximum (which is set to 0x400 – 1024 – in `setup_pwm`). Then the value sent to the PWM is decreased to slow the motor down. Then GPIO17 is set high, so that the motor will get power on the low phase of the PWM signal. The PWM is re-enabled with the mode `PWM0_ENABLE|PWM0_REVPOLAR`. The reverse polarization flag flips the PWM signal, so that a low value sent to the PWM results in a signal that is high most of the time (rather than low most of the time). That way the same code can be used to slowly ramp up the speed of the motor (but in the 'one way' direction as in the table on page 27), then slow it down again. Finally the PWM is switched off, and the GPIO interface is closed down.

Motor Tests in Python

The `motor-rg.py` and `motor-wp.py` programs are rather different because the RPi.GPIO package does not yet support hardware pulse-width modulation (PWM), but WiringPi for Python does (it works, although it's still undocumented). Thus `motor-rg.py` (the RPi.GPIO version) uses software PWM, which is found in the function `run_motor()`. If you try both programs, you'll probably get smoother results with `motor-wp.py` (the WiringPi for Python version).

Both versions use the Python 3 `print()` function, which is imported like this

```
from __future__ import print_function
```

Without this function, it would be difficult to prevent line-breaks and spaces in the on-screen output. It means that all print statements need to be written as `print()` instead of just `print` because they are now function calls and not statements.

motor-rg.py (software PWM)

After importing the required modules, we define which ports to use (18 and 17), how many times to run each loop (`Reps`), PWM cycle time (`Hertz`), frequency for motor loop time period (`Freq`). Then the ports are set up as outputs and set to OFF (0).

Next the function

```
run_motor(Reps, pulse_width, port_num, time_period)
```

is defined. This controls the GPIO port switching on and off for precise periods of time. The port `port_num` is switched on for time `pulse_width`, then switched off for time `time_period`, and all of that is repeated `Reps` times (400 in this case). The main loop in `run_motor()` is inside a `try: except:` block. This is an important safety precaution to ensure the motor is switched off on keyboard interrupt. (This is even more important with hardware PWM in the other version). The `run_motor()` function is called many times from the `run_loop()` function. Each loop of 400 repetitions is just one step up or down in motor speed: 400 repetitions at 2000 Hertz represents just 0.2 seconds.

Then we define the `run_loop()` function

```
run_loop(startloop, endloop, step, port_num, printchar)
```

The arguments `startloop` and `endloop` are the % time for the switched port to be ON at the start and end of the loop. `step` is the size of the increment for each successive loop. `port_num` is the port we're switching (17 or 18). `printchar` is the acceleration/deceleration indication character (`'+'` or `'-'`). The `run_loop()` function handles the repeated calls to the `run_motor()` function causing it to run its 400 cycles for each set of values defined.

After the functions are defined, wiring instructions are printed out and the computer waits for a key press of **enter** to confirm the user is ready. Then the `run_loop()` function is called four times to drive the motor speed from 5% to 95% and back again in each direction. After this, both ports are set to OFF (False) and then reset.

Suggested safe tweaks to experiment with. Try changing these one at a time and see what they do...

- `run_loop(5, 95, 1, 18,'+')` – change + to something else
- `run_loop(5, 95, 1, 18,'+')` – change 5 to a higher number < 95
- `run_loop(5, 95, 1, 18,'+')` – change 95 to a lower number (but still greater than what you changed the 5 to)

motor-wp.py (hardware PWM)

The Raspberry Pi has one available hardware PWM port (GPIO18). We will use port 17 to control motor direction and port 18 to handle the pulsing.

The first part of the program imports the required modules, including the Python 3 `print()` function (explained above in the section on `motor-rg.py`). After initialising WiringPi, port 18 is set to PWM mode with `wiringpi.pinMode(18,2)` and port 17 is set up for normal output. Then both ports are set to 0 (OFF). Wiring instructions are then printed out and the computer waits for user confirmation. When the user is ready, the program defines three functions that are used repeatedly.

`display(printchar)` handles the correct display of the motor acceleration/deceleration indicators using the imported Python 3 `print()` function.

`reset_ports()` handles the resetting of the ports on program exit. This is not built into WiringPi for Python so it needs to be defined here. This is particularly important in the motor program as it avoids an uncontrollable "motor running" situation on program exit. This is an important safety consideration if you're going to run things with propellers etc.

`loop(start_pwm, stop_pwm, step, printchar)` handles the main PWM control loop. As before, this is called in four different ways (increasing speed then decreasing speed with motor going one way, then increasing and decreasing speed with motor going the other way). `start_pwm` is the 0-1024 PWM value to start the loop with. `stop_pwm` is the 0-1024 PWM value to end the loop with. `step` is the incremental/decremental PWM value for each successive loop. `printchar` is the character to denote motor accelerating or decelerating.

The main body of the program contains four calls to `loop()` to demonstrate acceleration and deceleration in each of two different rotational directions, with a short "aesthetic" pause in between each loop, defined by `rest = 0.013` (0.013s). When port 17 is 0, the motor rotates in one direction. When port 17 is 1, it rotates the other way. As usual, the main body is contained in a `try: except:` block to enable safe port reset, before exit, on keyboard interrupt.

Suggested safe tweaks to experiment with. Try changing these one at a time and see what they do...

- `rest = 0.013` – change 0.013 to 0 and see why it's there
- `loop(140, 1024, 1, '+')` – change 140 to a positive number nearer 0
- `loop(140, 1024, 1, '+')` – change 1024 to a lower number > 140
- `loop(140, 1024, 1, '+')` – change + to another character

Digital to Analogue and Analogue to Digital Converters

In the Gertboard functional blocks diagram (Figure 2 on page 6), the components implementing the converters are outlined in orange. Both the digital to analogue converter (D/A) and analogue to digital converter (A/D) are 8-pin chips from Microchip, although oddly they are in different sorts of packages. The A/D (labelled U6 on the circuit board) is in a dual in-line package whilst the A/D (U10) is surface mounted. Each supports 2 channels.

Both use the SPI bus to communicate with the Raspberry Pi. The SPI pins on the two chips are connected to the pins labelled SCLK, MOSI, MISO, CSnA, and CSnB in the header just above J2 on the board (thus in the functional blocks diagram, these pins are also outlined in orange). SCLK is the clock, MOSI is the output from the RPi, and MISO is the input to the RPi. CSnA is the chip select for the A/D, and CSnB is the chip select signal for the D/A (the 'n' in the signal name means that the signal is 'negative', thus the chip is only selected when the pin is low). Both A/D and D/A chips have a 10K pull-up resistor on their chip-select pins, so the devices will not be accessed if the chip-select pins are not connected.

The SPI pins are conveniently located just above GP7 to GP11 in header J2, because one of the alternate functions of these pins is to drive the SPI signals. For example, the 'ALT0' (alternative 0) function of GPIO9 is SPI0_MISO, which is why the pin labelled MISO is just about the pin labelled GP9. Thus to use the A/D and D/A, simply put jumpers connecting pins GP7 to GP11 to the SPI pins directly about them (although technically you only need CSnA for the A/D and CSnB for the D/A).

In the schematics, the D/A and A/D converters are on the upper left of page A-6.

Digital to Analogue Converter

The Gertboard uses an MCP48x2 digital to analogue converter (D/A) from Microchip. The device comes in three different types: 8, 10 or 12 bits. The one you get on your board is determined by availability of parts. To see which one you have, look very closely at the small chip in location U10. It should have a number 48x2 stamped on it, where x is either 0, 1, or 2.If x is 0, you have the 8-bit version. If it's 1 you have the 10 bit version, and if it's 2 you have the 12-bit one. These chips are all pin-compatible and are written to in the same way. In particular, the routine that writes to the D/A assumes that writes are in 12 bits, so it is important that the value is selected appropriately (details are below in the section “Testing the D/A and A/D”). The maximum output voltage of the D/A – the output voltage when you send an input of all 1s – is 2.04V.

The analogue outputs of the two channels go to pins labelled DA0 (for channel 0) and DA1 (for channel 1) in the J29 header on the left edge of the board. Just next to these pins are ground pins (GND) to provide a reference.

Analogue to Digital Converter

The Gertboard uses a MCP3002 10-bit analogue to digital converter from Microchip. It supports 2 channels with a sampling rate of ~72k samples per second (sps). The maximum value (1023) is returned when the input voltage is 3.3V.

The analogue inputs for these two channels are AD0 (for channel 0) and AD1 (for channel 1) in the J28 header. Just next to these pins are ground pins (GND) to provide a reference.

Testing the D/A and A/D

According to the data sheet for the D/A, the value on the output pin, Vout, is given by the following formula (assuming the 8-bit MCP4802):

$$Vout = \frac{D_{in}}{256} \times 2.048V$$

Vout for channel 0 is DA0 in J29; for channel 1 it’s DA1.

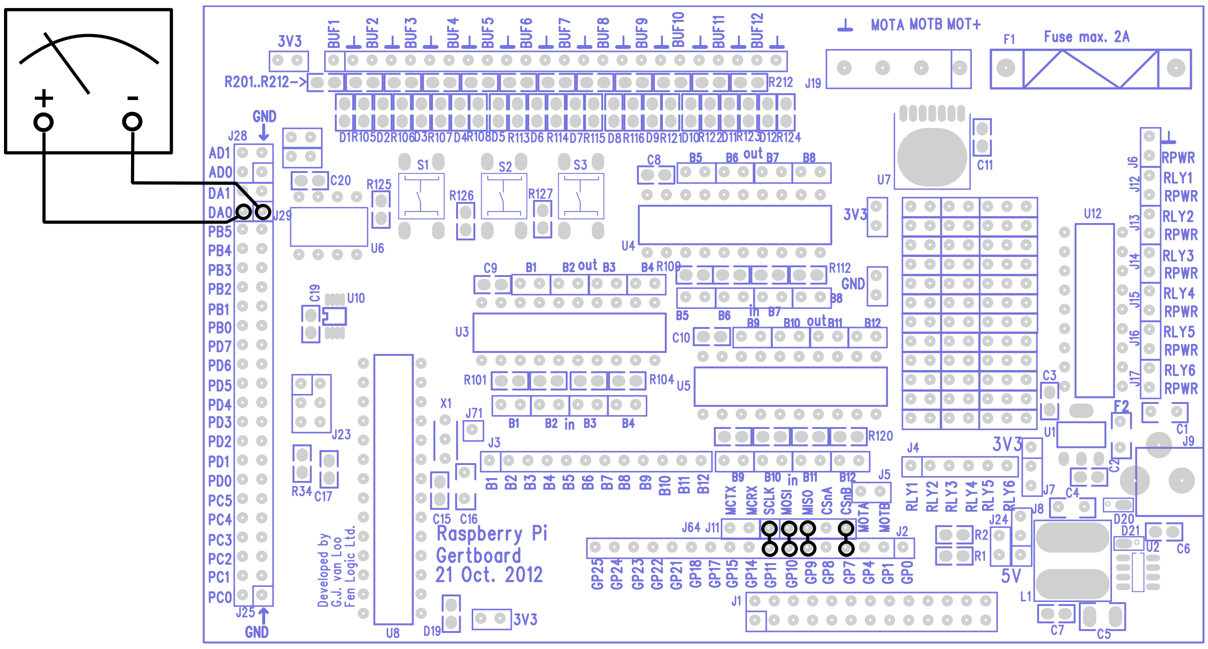


Figure 22: The wiring diagram for the dtoa test.

To test the D/A, a multi meter is required. The C test program for this is `dtoa`. The Python version is `dtoa.py`. To set up Gertboard for this test, jumpers are placed on the pins GP11, GP10, GP9, and GP7 connecting them to the SPI bus pins above them. Attach the multi meter as follows: the black lead needs to be connected to ground. You can use any of the pins marked with ⊥ or GND for this.

The red lead needs to be connected to DA0 (to test the D/A channel 0 which is shown below) or DA1 (for channel 1). Switch the multimeter on, and set it to measure voltages from 0 to around 5V. All this is shown in Figure 22.

The C test program for the A/D is called `atod`; the Python version is `atod.py`. To run this test a voltage source on the analogue input is required. This is most easily provided by a potentiometer (a variable resistor). The two ends of the potentiometer are connected, one side to high (3.3V, which you can access from any pin labelled 3V3) and the other to low (GND or ⊥), and the middle (wiper) part to AD0 (for channel 0 as shown below) or AD1 (for channel 1). To use the SPI bus jumpers should be installed on the pins GP11, GP10, GP9, and GP8 connecting them to the SPI bus pins above them. This is shown in Figure 23.

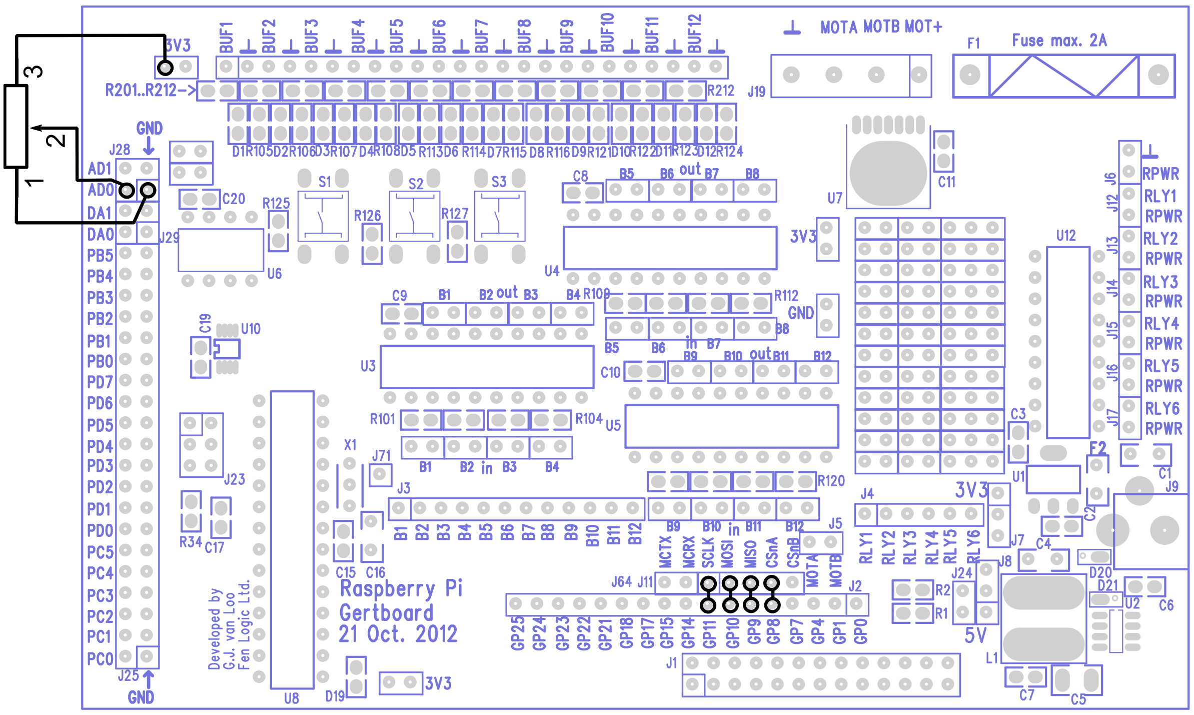


Figure 23: Wiring diagram for the test program atod.

Even without a multi meter or a potentiometer, it is still possible to test the A/D and D/A by sending the output of the D/A to the input of the A/D. The test that does this is called `dad`, for digital-analogue-digital. To set the Gertboard up for this test, hook up all the SPI bus pins (connecting GP11 though GP7 with jumpers to the pins above them) and put a jumper between pins DA1 and AD0, as in the diagram below.

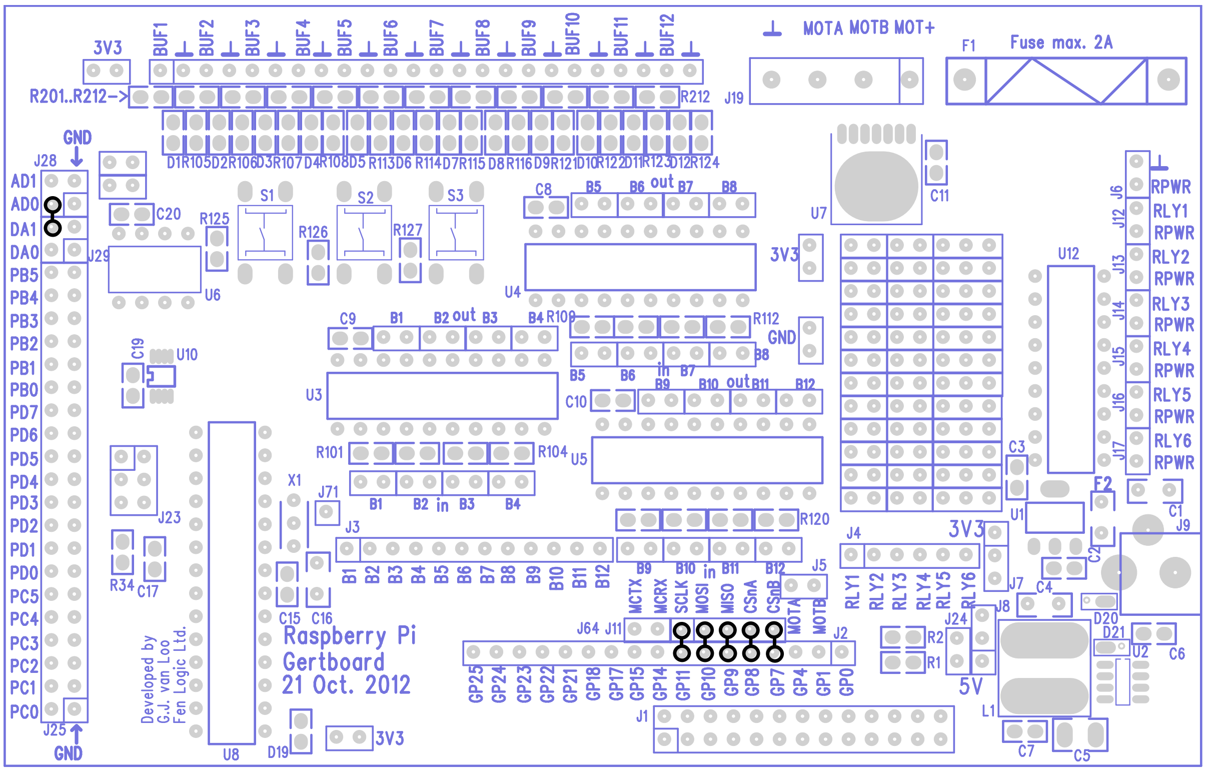


Figure 24: The wiring diagram for the dad test, which allows you to test the A/D and D/A converters together, without the aid of a multimeter or potentiometer.

D/A and A/D tests in C

Since the D/A and A/D converters both use the SPI bus, the common SPI bus code has been placed into a separate file, `gb_spi.c`. There is also an associated header file, `gb_spi.h`, which contains many macros and constants needed for interacting with the SPI bus, as well as the declarations for the functions in `gb_spi.c`. These functions are `setup_spi`, `read_adc`, and `write_dac`. `setup_spi` sets the clock speed for the bus and clears status bits. `read_adc` takes an argument specifying the channel (should be 0 or 1) and returns an integer with the value read from the A/D converter. The value returned will be between 0 and 1023 (i.e. only the least significant 10 bits are set), with 0 returned when the input pin for that channel is 0V and 1023 returned for 3.3V.

The `write_dac` routine takes two arguments, a channel number (0 or 1) and a value to write. The value written requires some explanation. The MCP48xx family of digital to analogue converters all accept a 12 bit value. The MCP4822 uses all the bits; the MCP4812 ignores the last two; and the MCP4802 ignores the last four. Since any of those chips might appear on the Gertboard (depending on availability), `write_dac` is written in so that it will work with all three, so it simply sends to the D/A the value it was given. If Gertboard is fitted with the MCP4802, it can only handle values between 0 and 255, but these must be in bits 4 through 11 (assuming the least significant bit is bit 0) of the bit string it is sent. Thus if the desired number to be sent to the D/A is between 0 and 255, it must be multiplied by 16 (which effectively shifts the information 4 bits to the left) before sending this value to `write_dac`.

dtoa

To test the D/A, the `dtoa` program first asks which channel to use and prints out the connections needed to make on Gertboard to run the program. Then it calls `setup_io` to get the GPIO ready to use, then calls `setup_gpio` to choose which pins to use and how to use them. In `setup_gpio`, as usual `INP_GPIO(n)` (where *n* is the pin number) is used to activate the pins. This also sets them up to be used as inputs. They should however, be used as an SPI bus, which is one of the alternative functions for these pins (it is alternate 0). Thus we use `SET_GPIO_ALT(n, a)` (where *n* is the pin number and *a* is the alternate number, in this case 0) to select this alternate use of the pins. Then the program sends different values to the D/A and asks for real verification, using the multimeter, that the D/A converter is generating the correct output voltage.

atod

To test the A/D, the `atod` program first asks which channel should be used and prints out the connections required on Gertboard to run the program. Then it calls `setup_io` to get the GPIO ready, then calls `setup_gpio` to choose which pins will be used, and how they will be used. The `setup_gpio` used in `atod` works the same way as the one in `dtoa` (except for activating GPIO8 instead of GPIO7).

Then `atod` repeatedly reads the 10 bit value from the A/D converter and prints out the value on the terminal, both as an absolute number and as a bar graph (the value read is divided by 16, and the quotient is represented as a string of '#' characters). One thing to be aware of is that even if the potentiometer is not moved, exactly the same result may not appear on successive reads. With 10 bits of accuracy, it is very sensitive, and even the smallest changes, such as house current running in nearby wires, can affect the value read.

dad

To test both the D/A and A/D at the same time, the `dad` test sends 17 different digital values to the D/A (0 to 255 in even jumps, then back down to 0). The resulting values are then read in from the A/D. Both the original digital values sent and the values read back are printed out, as is a bar graph representing the value read back (divided by 16 as in `atod`). The bar graph printed out should be a triangle shape: the lines will start out very short, then get longer and longer as larger digital values are read back, then will get shorter again.

D/A and A/D tests in Python

The analogue to digital and digital to analogue converters are connected to the (SPI) ports on the Raspberry Pi. These are the alternative functions of GPIO ports 7 and 8.

In order to make use of `atod.py`, `dtoa.py` and `dad.py` you must have SPI enabled, and you must install a Python module called **py-spidev**. Instructions on how to do this are in the `README.txt` included with the Python programs that you downloaded.

atod.py

The user chooses which channel to use on the analogue to digital converter (A/D), then wiring instructions are printed out. The function `get_adc(channel)` uses **spidev** to read the A/D. It receives three bytes of data and extracts the result – a number between 0 and 1023, where 0 means 0V and 1023 means 3.3V.

The main loop runs 600 times with a "sleep" of 0.05s, so the program takes about 30 seconds (600 * 0.05) to run. During each iteration of the loop, the program reads the A/D and displays this reading and a number of # symbols proportional to the value read, which depends on the position of the potentiometer. It uses the `display(char, reps, adc_value, spaces)` function to achieve this.

Moving the potentiometer while the program is running changes both the numerical readout and the number of # characters displayed.

Suggested tweaks to experiment with. Try changing these one at a time and see what they do...

- `char = '#'` – change value of char from # to a symbol of your choice (line 30)
- `sleep(0.05)` – change the 0.05 and see what happens (line 56)

dtoa.py

The digital to analogue converter (D/A) is controlled by writing 2 binary bytes (16 bits in total) to it via the SPI interface. The program uses a Python module called **spidev** to handle the SPI communication. We have to give spidev two base 10 numbers, which it converts into 8-bit binary bytes, which it sends to the D/A. The D/A outputs a voltage according to the value given to it. An input of 0 gives 0V, 255 gives 2.048V, and it's linear in between, so 128 should give 1.02V.

The pre-determined values we send to the D/A are stored in two list variables...

- `num_list[]` – holds the first byte (different for each channel).
- `common[]` – holds the second byte of data (same for each channel).

The user chooses the channel and then the main loop takes the first number from each list and combines byte1 with byte2. Then it sends them to the D/A, which immediately outputs the desired voltage until **enter** is pressed. It iterates through all five values, setting the D/A and waiting for a key press. Both channels are reset to zero at the end of the program.

Suggested tweaks to experiment with.

- None this time. If you change the numbers in the lists you will break the program (the voltage output will no longer agree with the numbers that the test program prints out). If you want to see how they are derived, you can look in the `dad.py` program in the `dac_write` function

dad.py

This program uses spidev to control the A/D and the D/A using both of the SPI ports on the Raspberry Pi. First the wiring instructions are printed out on the screen. Then the main loops (one for each direction) iterate from 0 to 256 and back again in increments of 32. This value is sent to the D/A using `dac_write(DAC_value)`, and the appropriate voltage appears on the ouput pin DA1.This voltage goes by a jumper to AD0, an input of the A/D, and the `get_adc(adc_channel)` function is called to read this voltage using the A/D, returning a number between 0 and 1023. Both numbers are then printed out along with a bar chart (using the # character) representing the A/DC.

Suggested safe tweaks to experiment with.

- Change the 32 in both loops `for DAC_value in range(0,257,32)` and `for DAC_value in range(224,-1,-32)`

- Change the 4 in `adc_string = "{0:04d}"`
- Change the 3 in `print "%s %s %s" % ("{0:03d}")`
- Remove the jumper between AD0 and DA1 and see what happens

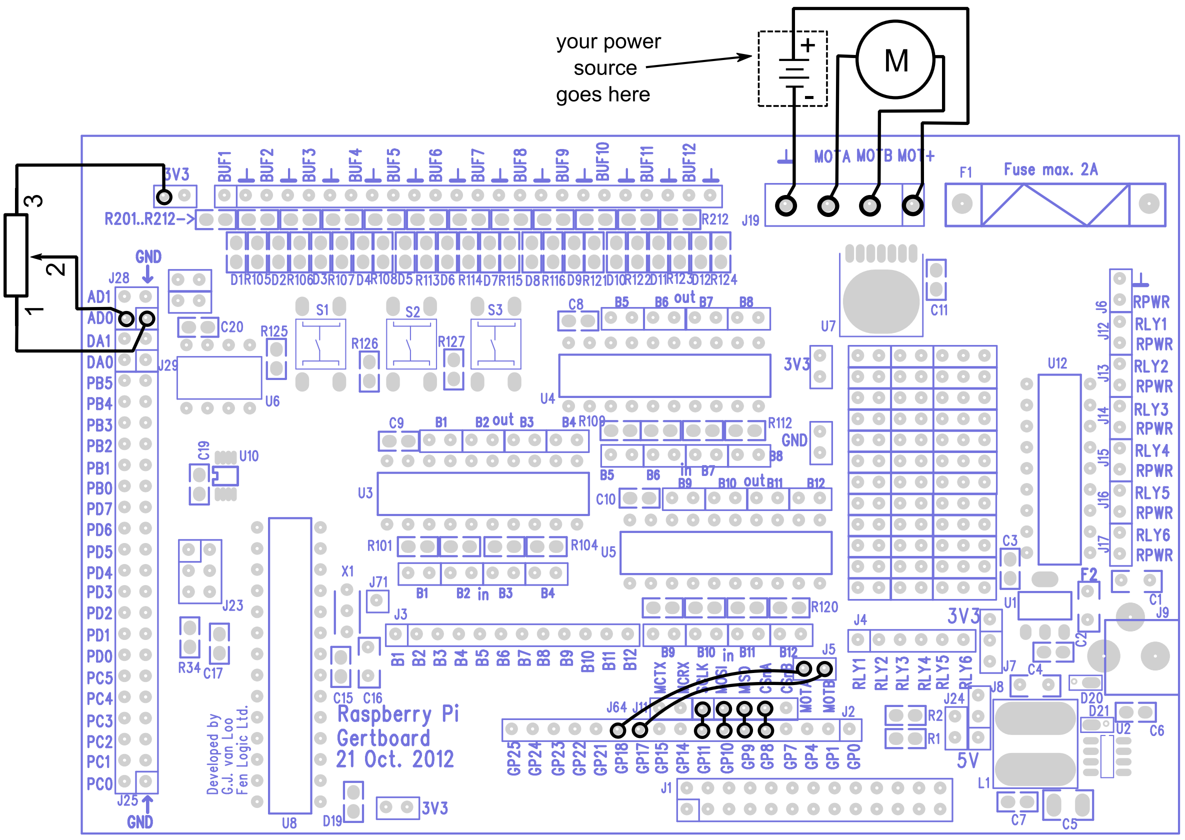
Combined Tests

This section shows some examples of using more than one functional block at a time.

A/D and Motor Controller

In the `potmot` (for potentiometer-motor) test we use a potentiometer ("pot") connected to the analogue to digital converter (A/D) to get an input value, and this value is used to control the speed and direction of the motor. It is set up so that at one extreme, the motor is going at top speed in one direction, as you move the wiper towards the middle it slows, at the middle the motor stops, and as you continue to move the wiper along, the motor speeds up again but in the other direction.

To wire up the Gertboard for this example, you combine the wiring for the A/D and motor tests. Jumpers connect GP8 to GP11 to the pins directly above them to allow us to control the SPI bus using GPIO8 to GPIO11. You must attach your potentiometer to the AD0 input. GPIO17 controls the motor B input and GPIO18 controls the motor A input using the pulse width modulator (PWM). Thus GP17 must be connected via a strap to MOTB, and GP18 must be connected to MOTA. The motor and its power source must be connected to the screw terminals in J19 at the top of the board. See the wiring diagram below.



your power source goes here
M
+
-
3
2
1
MOTA MOTB MOT+
Fuse max. 2A
J19
AD1
AD0
DA1
DA0
GND
3V3
Raspberry Pi Gertboard 21 Oct. 2012
Developed by G.J. van Loo Fen Logic Ltd.
GP25 GP24 GP23 GP22 GP21 GP18 GP17 GP15 GP14 GP11 GP10 GP9 GP8 GP7 GP4 GP1 GP0

Figure 25: Wiring diagram for the potmot test.

Potmot Test in C

The `main` routine for this is in `potmot.c`. Functions from `gb_spi.c` and `gb_pwm.c` are used to control the SPI bus (for reading the A/D) and the pulse width modulator (for controlling the speed of the motor).

In the `main` routine for `potmot`, first we print to the terminal the connections that need to be made on the Gertboard to run this example, then we call `setup_io` to set up the GPIO ready for use. Then we call `setup_gpio` to set the GPIO pins the way we want them. In this, we set up GPIO8 to GPIO11 to use the SPI bus using `INP_GPIO` and `SET_GPIO_ALT` as described in the section on the converters (D/A and A/D tests in C, page 35). GPIO17 is set up as an output (using `INP_GPIO` and `OUT_GPIO`), and GPIO18 is set up as a PWM using as `INP_GPIO` and `SET_GPIO_ALT` as described in the section on the motor controller (Motor Test in C, page 29). Back in `main`, we call `setup_spi` and `setup_pwm` to get the SPI bus and PWM ready for use and get the motor ready to go.

Then we repeatedly read the A/D and set the direction and speed of the motor depending on the value we read. Lower A/D values (up to 511 – recall that the A/D chip used returns a 10 bit value so the maximum will be 1023) result in the motor B input being set high, and thus the motor goes in the "rotate one way" as in the motor controller table (Table 3, page 27). Confusingly, this motor direction is called "backwards" in the comments of the program! Higher A/D values (512 to 1023) result in the motor B input being set low, and the motor goes in the "rotate opposite way" direction. This is called "forwards" in the comments of the program. Simple arithmetic is used to translate A/D values near 511 to slow motor speeds and A/D values near the endpoints of the range (0 and 1023) to fast motor speeds by varying the value sent to the PWM.

Potmot test in Python

This program, `potmot-wp.py`, uses spidev to control the A/D and WiringPi for Python to control the motor with the hardware PWM. Essentially `potmot` is a simplified combination of the `atod.py` and `motor-wp.py` programs. It is simplified in that there is no on-screen display of the A/D reading or motor direction.

The potentiometer position (read by the ADC) determines motor direction and speed (PWM value) as follows: middle value (511) results in no movement, 1023 results in max speed one way, 0 results in max speed the other way.

First the program imports the required modules, spidev and wiringpi, then sets up GPIO ports 17 and 18 as digital output and PWM output respectively. Then two functions are defined; `get_adc()` reads the voltage at the potentiometer using the A/D; `reset_ports()` ensures we can safely exit the program with the ports switched off. Then the initial values of variables are set and the wiring instructions are printed out on the screen. The program then waits for user input before proceeding.

Once the user hits **enter**, the SPI port is opened to read the potentiometer voltage using the A/D. The A/D value is read and if above 511 we set port 17 to 0, which sets motor direction one way. Otherwise direction is set the other way. Then the PWM value sent to port 18 is calculated, based on the value read from the ADC. This determines how fast the motor will spin. After the PWM value is written to port 18, the program waits 0.05 seconds and then repeats the main loop, reading the A/D value again. This occurs 600 times, so the program runs for about 30 seconds.

The main loop is wrapped in a `try: except:` block to enable safe resetting of the ports in the event of a CTRL+C keyboard interrupt.

Suggested safe tweaks to experiment with.

- none this time

Decoder

The decoder implemented by the `decoder` program takes the three pushbuttons as input and turns on one of 8 LEDs to indicate the number with the binary encoding given by the state of the buttons. Switch S1 gives the most significant bit of the number, S2 the middle bit, and S3 the least significant bit. For output, the LED D5 represents the number 0, D6 represents 1, and so on, so D12 represents 7. Recall that the pushbuttons are high (1) when up and low (0) when pushed, so LED D12 is lit up when no buttons are pressed (giving binary 111 or 7), D6 is lit up when S1 and S2 are pressed (giving binary 001), etc.

There is quite a bit of wiring for this one, as we are using all but one of the I/O ports.GPIO25 to GPIO23 are reading the pushbuttons, so you need to connect GP25 to B1, GP24 to B2, and GP23 to B3. The 8 lowest-numbered GPIO pins are used with I/O ports 5 to 12, so you need to connect GP11 to B5, GP10 to B6, GP9 to B7, GP8 to B8, GP7 to B9, GP4 to B10, GP1to B11, and GP0 to B12. In addition, since we are using I/O ports 5 to 12 for output, you need to install all the out jumpers for buffer chips U4 and U5 (recall that the out jumpers are those above the chips).

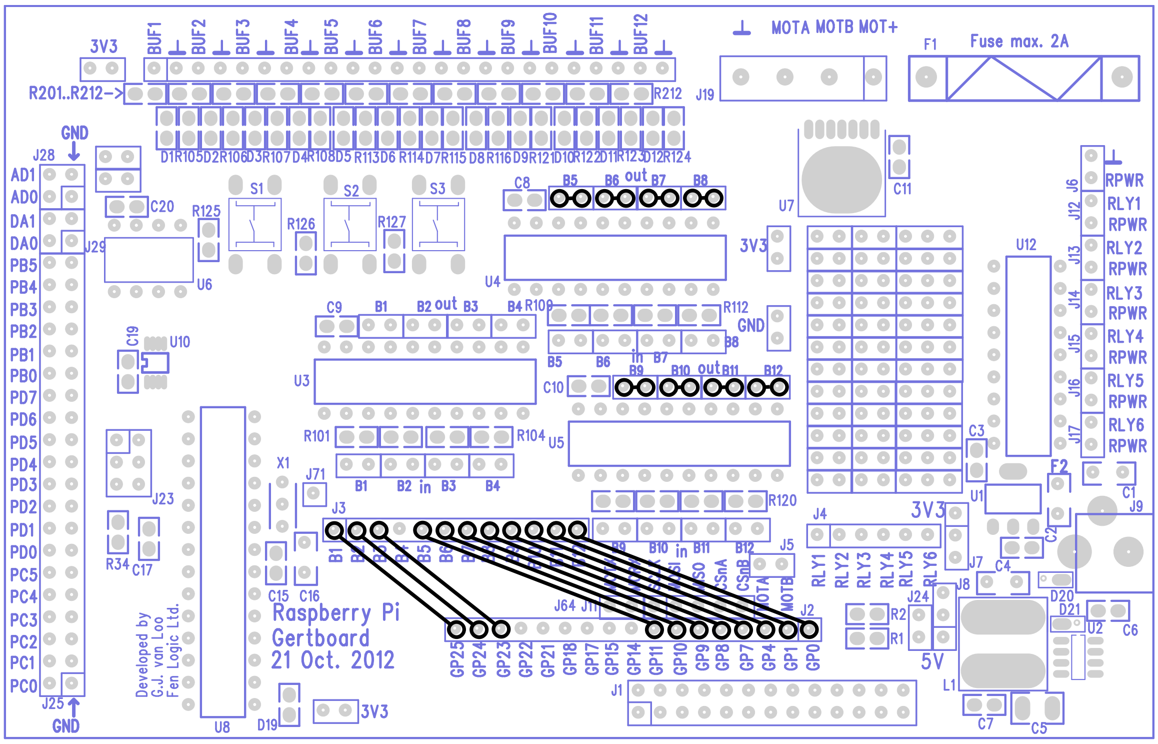


Figure 26: Wiring diagram for the decoder test.

Decoder Test in C

In the `main` routine for `decoder`, as always we start out by printing out to the terminal the connections that need to be made on the Gertboard. Then we call `setup_io` to set up the GPIO ready for use. Then we call `setup_gpio` to set GPIO25 to 23 for use with the pushbuttons (by selecting them for input and enabling a pull-up, as described on page 19) and to set GPIO11 to GP7, GPIO4, GPIO1, and GPIO0 up as outputs (as described on page 13). Then we enter a loop where we read the state of the pushbuttons and light up the LED corresponding to this number (after turning off the LED previously set). We turn the LEDs on and off using `GPIO_SET0` and `GPIO_CLR0` as described on page 21.

At the time of writing, there is no decoder test in Python.

ATmega Device

The Gertboard can hold an Atmel AVR microcontroller, a 28-pin ATmega device, at location U8 on the lower left of the board. This can be any of the following: ATmega48A/PA, 88A/PA, 168A/PA or 328/P in a 28-pin DIP package. Usually the 168 or 328 is fitted. The device has a 12MHz ceramic resonator attached to pins 9 and 10. All input/output pins are brought out to header J25 on the left edge of the board. There is a separate 6-pin header (J23 on the left side of the board) that can be used to program the device.

The PD0/PD1 pins (ATmega UART TX and RX) are brought out to pins placed adjacent to the Raspberry Pi UART pins so you only need to place two jumpers to connect the two devices.

Note that the ATmega device on the Gertboard operates at 3.3Volts. That is in contrast to the 'Arduino' system which runs at 5V. (This is the reason why the device does not have a 16MHz clock. In fact at 3V3 the maximum operating frequency according to the specification is just *under* 12MHz.) ***Warning***: many of the Arduino example sketches (programs) mention +5V as part of the circuit. Because we are running at 3.3V, you must use 3.3V instead of 5V wherever the latter is mentioned. If you use 5V you risk damaging the chip.

The ATmega device and the headers connected to it are in the schematics on page A-6.

Programming the ATmega

Programming the ATmega microcontroller is straightforward once you have all the infrastructure set up, but it requires a fair bit of software to be installed on your Raspberry Pi. We are very grateful to Gordon Henderson, of Drogon Systems, for working out what needed to be done and providing the customized software. Using his system, you can use the Arduino IDE (Integrated Development Environment) on the Raspberry Pi to develop and upload code for the ATmega chip on the Gertboard. All the software needed, along with instructions, is available at

https://projects.drogon.net/raspberry-pi/gertboard/

For the rest of this section, we assume that you have downloaded and successfully installed and configured the Arduino IDE, as described at Gordon's website, and we proceed from there.

To get going with the ATmega chip, start up the Arduino IDE. This should be easy: if the installation of the Arduino package was successful, you will have a new item "Arduino IDE" in your start menu, under "Electronics". The exact version of the IDE you get with depends on the operating system you are using. The vast majority of Raspberry Pi users are using Raspbian, which is based on Debian wheezy, so from now on we'll assume that you're running this. The version number is given in the title bar; for wheezy it's 1.0.1. First you will need to configure the IDE to work with the Gertboard. Go to the Tools > Board menu and choose the Gertboard option with the chip you are using (there are options for the ATmega168 and ATmega328, the ones most commonly used on the Gertboard). Then go to the Tools > Programmer menu and choose “Raspberry Pi GPIO”.

Arduino Pins on the Gertboard

All the input and output pins of the ATmega chip are brought out to header J25 on the left edge of the board. They are labelled PC*n*, PD*n*, and PB*n*, where *n* is a number. These labels correspond to the pinout diagrams of the ATmega168/328 chips. However, in the Arduino world, the pin numbers of the chips are not referred to directly. Instead there is an abstract notion of digital and analogue pin numbers, which is independent of the physical devices. This allows code written for one Arduino board to be easily used with another Arduino board, which may have a chip with a different pinout. Thus, in order to use your Gertboard with the Arduino IDE, you need to know how the Arduino pin number relates to the labels on your Gertboard. The table below shows this correspondence (“GB” means Gertboard).

Arduino Pin	GB pin	Arduino Pin	GB pin	Arduino Pin	GB pin
0	PD0	7	PD7	A0	PC0
1	PD1	8	PB0	A1	PC1
2	PD2	9	PB1	A2	PC2
3	PD3	10	PB2	A3	PC3
4	PD4	11	PB3	A4	PC4
5	PD5	12	PB4	A5	PC5
6	PD6	13	PB5		

Table 4: The relationship between Arduino pin numbering and pins on the Gertboard.

In sketches digital pins are referred to with just a number. For example

```
digitalWrite(13, HIGH);
```

will set pin 13 (PB5 on the Gertboard) to logical 1. (In the Arduino world, `LOW` refers to logical 0, and `HIGH` refers to logical 1.)

The analogue pins are referred to as A0 to A5. So to read from analogue pin 0 (PC0 on the Gertboard) you would use the command

```
value = analogRead(A0);
```

A Few Sketches to Get You Going

A *sketch* is the name that Arduino uses for a program. It's the unit of code that is uploaded to and run on an Arduino board (or, in our case, an ATmega microcontroller on a Gertboard). Let’s have a look at a simple one, `Blink`, which makes an LED turn on and off. This is accessible from the Arduino IDE from the File > Examples > Basics menu. When you select this, a new window pops up with the `Blink` code. There are only two functions in the code, `setup` and `loop`. These are required for all Arduino programs: `setup` is executed once at the very beginning, and `loop` is called repeatedly, as long as the chip has power. Note that you do not need to provide any code to call these functions; this is added automatically as part of the compilation and uploading process. The language used in these sketches is based on C, so the syntax in programs should look familiar if you have been looking at the C test programs for the Gertboard.

Uploading Sketches using the SPI Bus

In order to get your sketch running on the ATmega chip on the Gertboard, it has to be transferred over to the chip somehow (this is called *uploading* the sketch). There are various methods used to program ATmega chips, but we are going to use the SPI bus available on GPIO pins 8 through 11. This is possible because your Arduino IDE is using the special downloader/uploader (`avrdude`) that you got from projects.drogon.net. To set this up you need to connect the GPIO pins used for the SPI bus to the 6-pin header J23, as in the diagram below. Here you are simply connecting the SPI pins in the GPIO to the corresponding SPI pins in the header. The arrangement of the pins in J23 is shown in the schematics, on page A-6.

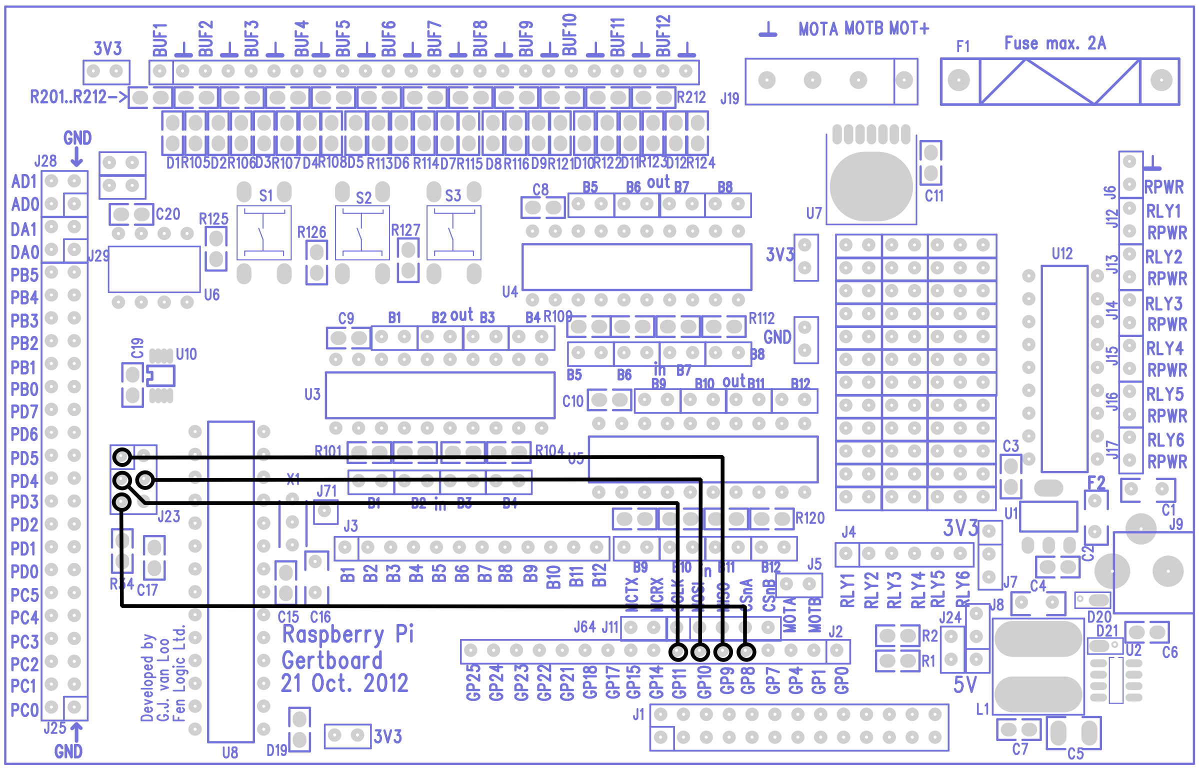


Figure 27: The wiring diagram for uploading sketches to the ATmega microprocessor.

To upload your sketch to the chip in Arduino IDE choose File > Upload Using Programmer. It will take a bit of time to compile and upload, and then your sketch is running on the microcontroller.

Blink Sketch

The `Blink` sketch is under the File > Examples > Basics menu. Bring it up and upload it to the ATmega chip using the instructions above. The sketch is now running, but nothing is happening! On most Arduino boards, pin 13 (the digital pin used by this sketch) has an LED attached to it, but not the Gertboard. You have to wire up the LED yourself. Looking at Table 4 above, we see that digital pin 13 is labelled PB5 on the Gertboard, so you need to connect PB5 to one of the I/O ports. In the section on Buffered I/O, LEDs, and Pushbuttons, we explained that you can make an LED show the value of a signal by connecting that signal to one of the pins labelled BUF1 to BUF12 in the (unlabeled) single row header at the top of the Gertboard. So if you connect PB5 to BUF1, as below, the first LED will start to blink.

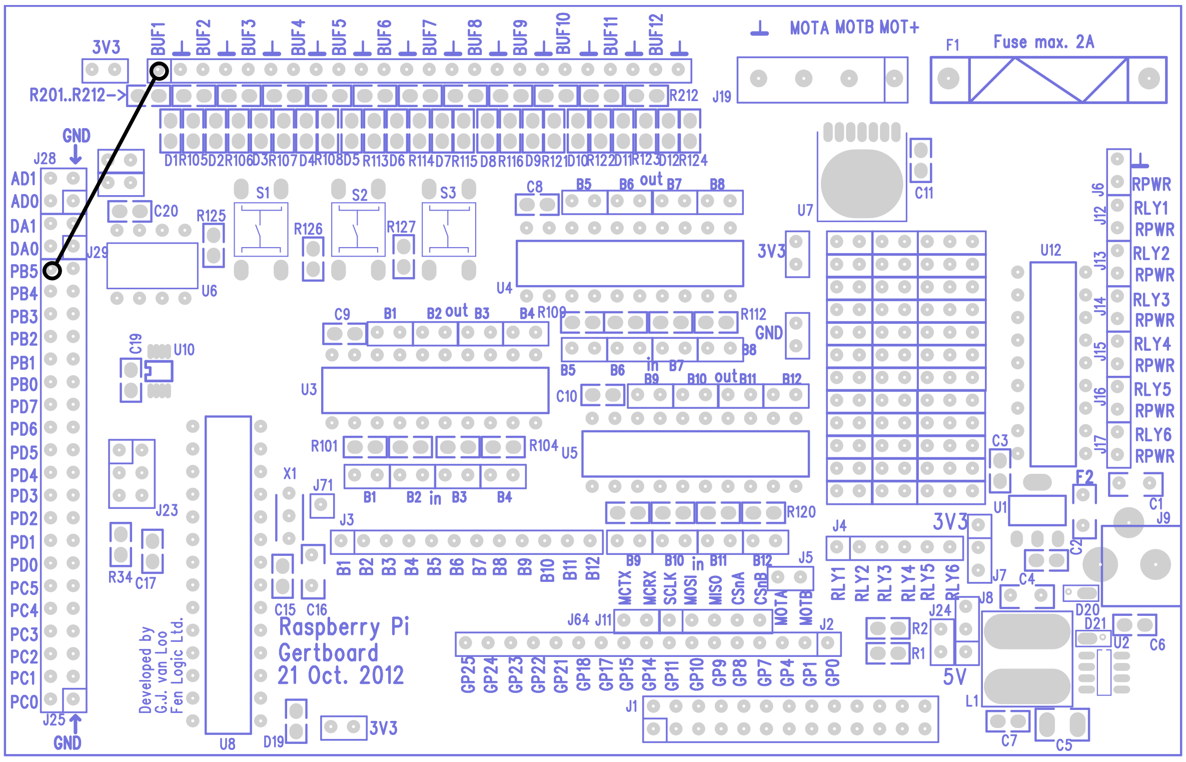


Figure 28: Wiring diagram for the sketch Blink.

Note that in this diagram we have not shown the connections to the SPI pins. Once you have uploaded the code, you no longer need them and can remove the straps. On the other hand, if you want you can leave them in place, and this is a good idea if you are planning on uploading some other sketches later.

Button Sketch

Let’s look at another fairly simple sketch called `Button`, located under File > Examples > Digital menu in both 0018 and 1.0.1. The comments at the beginning of the sketch read

```
The circuit:
* LED attached from pin 13 to ground
* pushbutton attached to pin 2 from +5V
* 10K resistor attached to pin 2 from ground
```

Assuming that you have `Blink` working, your LED is already wired up, but what about the button? As mentioned above, since the ATmega chip on the Gertboard runs at 3.3V, we must replace the 5V with 3.3V. So they suggest using a circuit like the one below, where the value read at pin 2 is logical 0 if the button is not pressed (due to the 10K pull-down resistor) and logical 1 if the button is pressed.

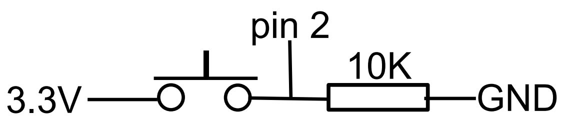


Figure 29: Suggested switch circuit for use with Button sketch.

However, the buttons on the Gertboard are used like this:

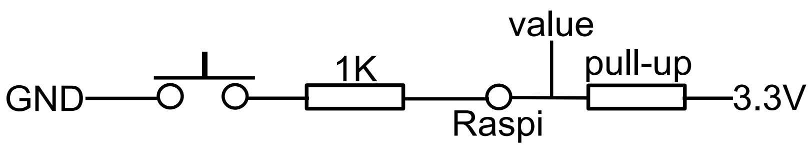


Figure 30: Circuit in use on the Gertboard, showing an additional 1k resistor to protect the input to BCM2835.

The 1K resistor between the pushbutton and the 'Raspi' point is to protect the BCM2835 (the processor on the Raspberry Pi) if you accidentally set the GPIO pin connected to 'Raspi' to output instead of input. The circuit to the right of the 'Raspi' point happens on the Raspberry Pi: to use the pushbutton we set a pull-up (shown as a resistor in the circuit above) on the pin so that the value read is logical 1 when the button is not pressed (see page 19 for more info on the pull-up). The Gertboard buttons are connected directly to ground so they cannot be made to read logic 1 when pressed. If you want to use a Gertboard button with an Arduino sketch that assumes that the button reads 1 when pressed, the best approach is to modify the sketch, if needed, so that it will invert the value it reads from the button. For the pull-up, we can take advantage of the pull-ups in the ATmega chip. To do this, find the lines below in the sketch

```
// initialize the pushbutton pin as an input:
pinMode(buttonPin, INPUT);
```

and insert the following two lines after them:

```
// set pullup on pushbutton pin
digitalWrite(buttonPin, HIGH);
```

To invert the value read from the button, find the line below:

```
buttonSate = digitalRead(buttonPin);
```

and insert a `!` (the negation operator in C) as follows:

```
buttonSate = !digitalRead(buttonPin);
```

Now upload this modified sketch, as described for `Blink`. We still need to attach Arduino digial pin 2 (PD2 on the Gertboard, as you can see from the table) to a button, say button 3.The 'Raspi' pin in the circuit diagram above, which is where we want to read the value, is in the J3 header.

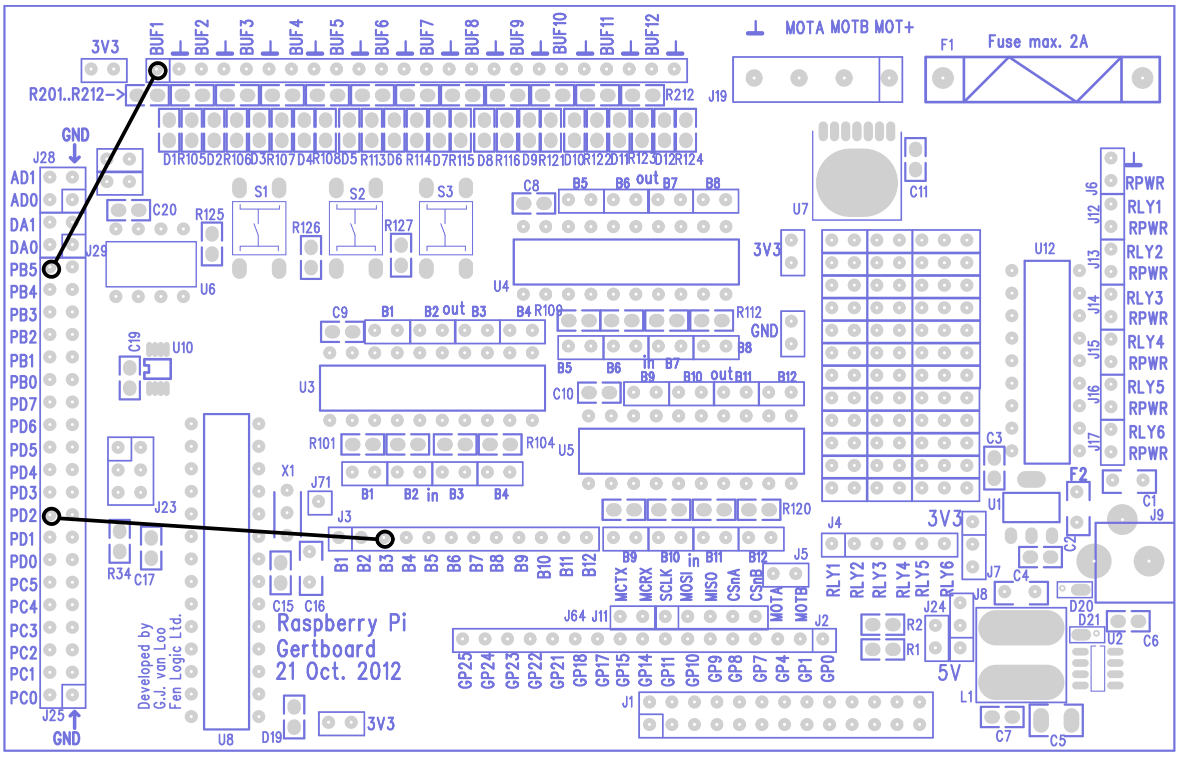


Figure 31: Wiring diagram for the sketch Button.

When you have done this, the first LED will be on when the third button is pressed, and off when the third button is up.

AnalogInput Sketch

Now let’s try using an analogue pin. Find the `AnalogInput` sketch under File > Examples > Analog. This reads in a value from analogue input 0 (which has already been converted by the internal A/D to a value between 0 and 1023), then uses that number as a delay between turning an LED on and off. Thus, the lower the voltage on the analogue pin, the faster the LED flashes. To run this example, you’ll need a potentiometer. The one used to test the A/D will work fine here. The comments for `AnalogInput` say to connect the potentiometer so that the wiper is on analogue pin 0 (PC0 on the Gertboard) and the outer pins are connected to +5V and ground. Remember, you must use 3.3V instead of 5V as we’re running the chip at 3.3V here. The diagram below shows how to connect up the Gertboard to make this sketch work after it is uploaded.

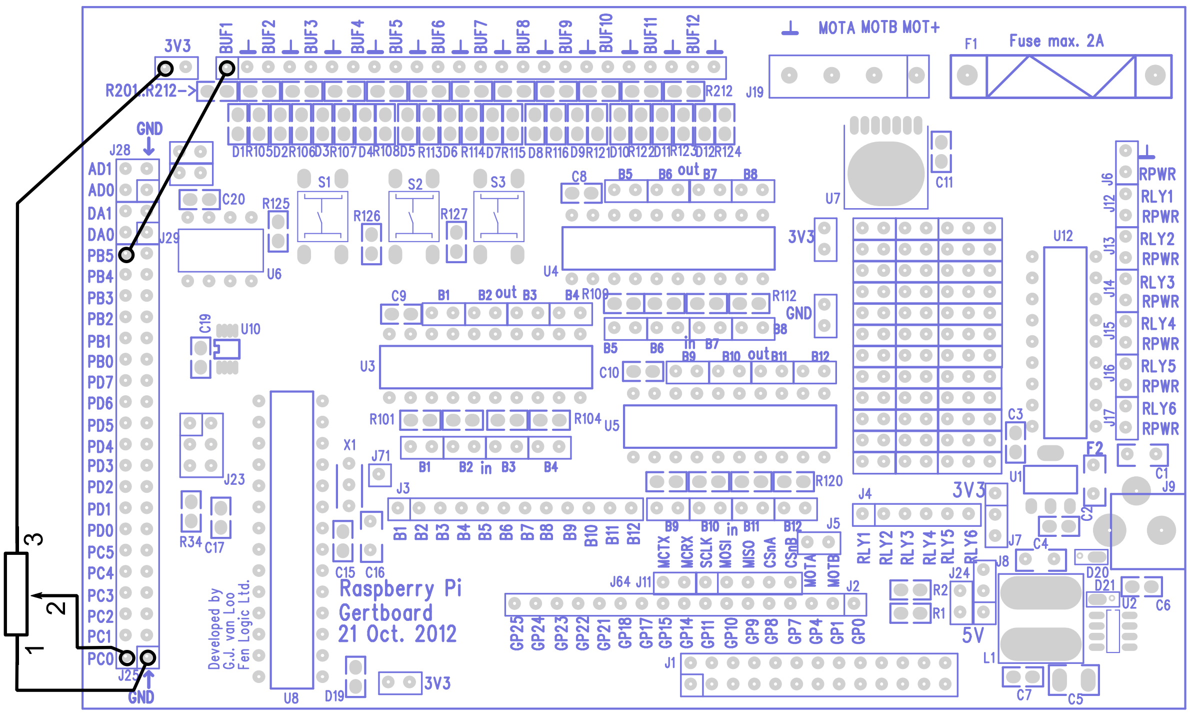


Figure 32: Wiring diagram for the AnalogInput sketch.

AnalogReadSerial Sketch Using Minicom

Some of the Arduino sketches involve reading or writing data via the serial port, or UART. An example is `AnalogReadSerial` which is in File > Examples > Basics. This sketch sets the baud rate to 9600, then repeatedly reads in a value from analogue pin 0 and prints this value to the serial port (also called UART). The value read in is between 0 and 1023; 0 means that the input pin is at 0V and 1023 means that it is at the supply voltage (3.3V for the Gertboard).

To set up your Gertboard for this sketch, you need the potentiometer attached to analogue input 0 as for the `AnalogInput` sketch. In addition you need to connect the ATmega chip's UART pins to the Raspberry Pi. Digital pin 0 (PD0 on the Gertboard) is RX (receive), and digital pin 1 (PD1 on the Gertboard) is TX (transmit). These signals are also brought out to the pins labelled MCTX and MCRX just above the GP15 and GP14 pins in header J2 on the Gertboard. Thus you can use two jumpers to attach the ATmega's TX to GP15 and RX to GP14, as shown below.

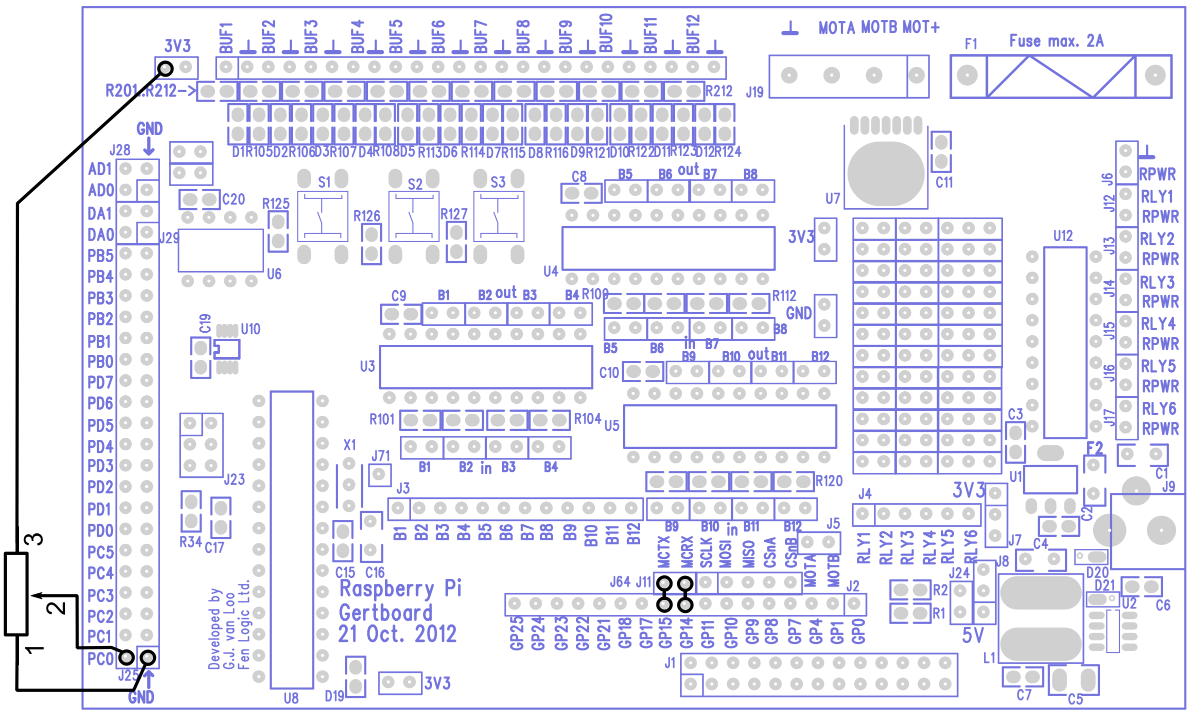


Figure 33: Wiring diagram for the sketch AnalogReadSerial.

GPIO14 and GPIO15 are the pins that the Raspberry Pi uses for the UART serial port. If you refer back to the table of alternate functions (Table 1, page 11), you will see that GPIO14 is listed as TX and GPIO15 as RX. This is not a mistake! This swapping is necessary: the data that is transmitted by the ATmega is received by the Raspberry Pi, and vice versa.

Now, how to we get the Raspberry Pi to read and show us the data that the ATmega is sending out on the serial port? There is a button labelled Serial Monitor on the toolbar of the Arduino IDE, but it doesn't work on the Raspberry Pi. It assumes that you are talking to an Arduino board over USB, not talking to a Gertboard over GPIO. The easiest way to retrieve this data is to use the minicom program. You can install this easily by typing into a terminal this command:

```
    sudo apt-get install minicom
```

You can use menus to configure minicom (by typing `minicom -s`). Alternatively, included with the Gertboard software is a file `minirc.ama0` with the settings you need to read from the GPIO UART pins at 9600 baud. Copy this file (which was provided by Gordon Henderson) to `/etc/minicom/` (you'll probably need to `sudo` this) and invoke minicom by typing

```
    sudo minicom ama0
```

Now if you upload the sketch to the ATmega chip, you should see the value from the potentiometer displayed in your minicom monitor.

LEDmeter Sketch

This example is one that we created specifically for the Gertboard, based on the `AnalogInput` sketch described above. In the `gertboard_sw` directory, along with all the C files, is one called `LEDmeter.ino`. This is an Arduino sketch that makes use of all 12 LEDs to create a bar graph showing the voltage from an analogue input such as a potentiometer. First you need to put this sketch in the right place. You should have a directory called `sketchbook` in your home directory. Make a subdirectory beneath that called `LEDmeter`, and copy `LEDmeter.ino` into that directory. To do this, you can type the following from your `gertboard_sw` directory:

```
mkdir ~/sketchbook/LEDmeter
cp LEDmeter.ino ~/sketchbook/LEDmeter
```

When that is done, the sketch will now be in the File > Sketchbook menu on the Arduino IDE. You can bring it up to inspect the code. In this example, we created two extra functions, `turn_on_leds` and `turn_off_leds`, mostly to demonstrate that you can use ordinary C code (including functions) in sketches. The pin numbers are stored in an array to make it easy to turn on or off a specific LED. If you want to access LED 4, use Arduino pin `led_pins[4]`. In order to do this, we put a `0` into the first location of the array, because there is no LED 0. One feature of C that you may not be familiar with is the use of `static` in the definition:

```
static int old_max_led = 0;
```

The keyword `static` means that the value of the variable should be kept the same between function calls. So if we assign it a value at the end of one call to `loop()`, the next time `loop()` is called `old_max_led` will still have that value. The initial value, `0`, is only assigned once, before `loop()` is called the first time, not every time `loop()` is called. The wiring diagram for `LEDmeter` is below.

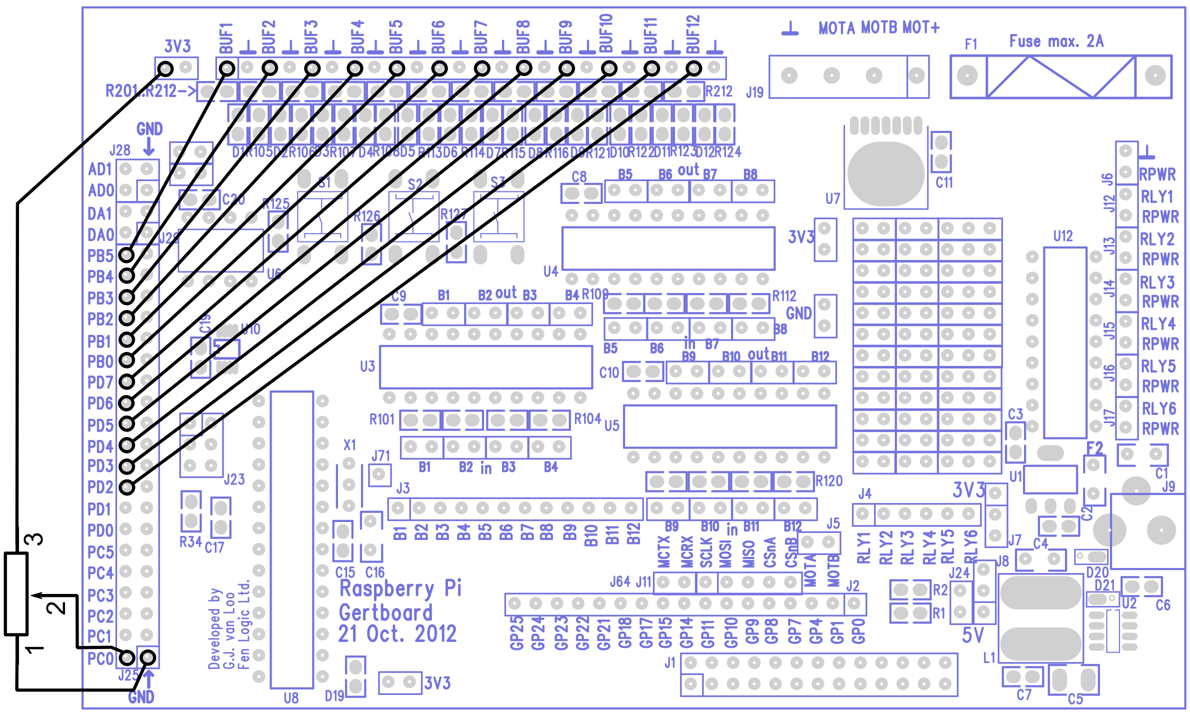


Figure 34: Wiring diagram for the LEDmeter sketch

Upload it the usual way, and the LEDs will respond to the position of the potentiometer.

Going Forward

These examples have only just scratched the surface of the wonderful world of Arduino. Check out http://arduino.cc/en/Tutorial/HomePage for much, much more.

For More Information

For further information on the Raspberry Pi and its GPIO ports, the datasheet for the processor can be found here:
http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

Appendix A: Schematics

We have included the schematics for the Gertboard in the pages that follow. They are numbered A-1, A-2, etc. The page number is located in the lower left hand of each page.