

Gertboard User Manual

Gert van Loo and Myra VanInwegen

Revision 1.0

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 is a suite of test/example programs for the Gertboard, written in C, which is freely available at www.element14.com/raspberrypi 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.

Contents

- Gertboard Overview..... 4
 - Labels on the circuit board..... 5
 - Location of the building blocks on the Gertboard 7
 - Jumpers and straps 8
 - GPIO pins 8
 - Schematics 9
 - Test programs overview..... 9
 - Macros..... 10
- Buffered I/O, LEDs, and pushbuttons..... 11
 - Push buttons..... 12
 - Locating the relevant sections of the Gertboard 12
 - Testing the pushbuttons 14
 - Testing the LEDs 16
 - Testing I/O 18
- Open Collector Driver..... 19
 - Testing the open collector drivers..... 20
- Motor Controller 22
 - Testing the motor controller..... 23
- Digital to Analogue and Analogue to Digital Converters 25
 - Digital to analogue converter..... 25
 - Analogue to Digital converter..... 26
 - Testing the D/A and A/D 26
- ATmega device 29
 - Programming the ATmega..... 30
 - Arduino pins on the Gertboard..... 30
 - A few sketches to get you going 31
 - Minicom..... 36
- Combined Tests 38
 - A/D and motor controller..... 38
 - Decoder 39
- For More Information 40
- Appendix A: Schematics..... 40

Gertboard Overview

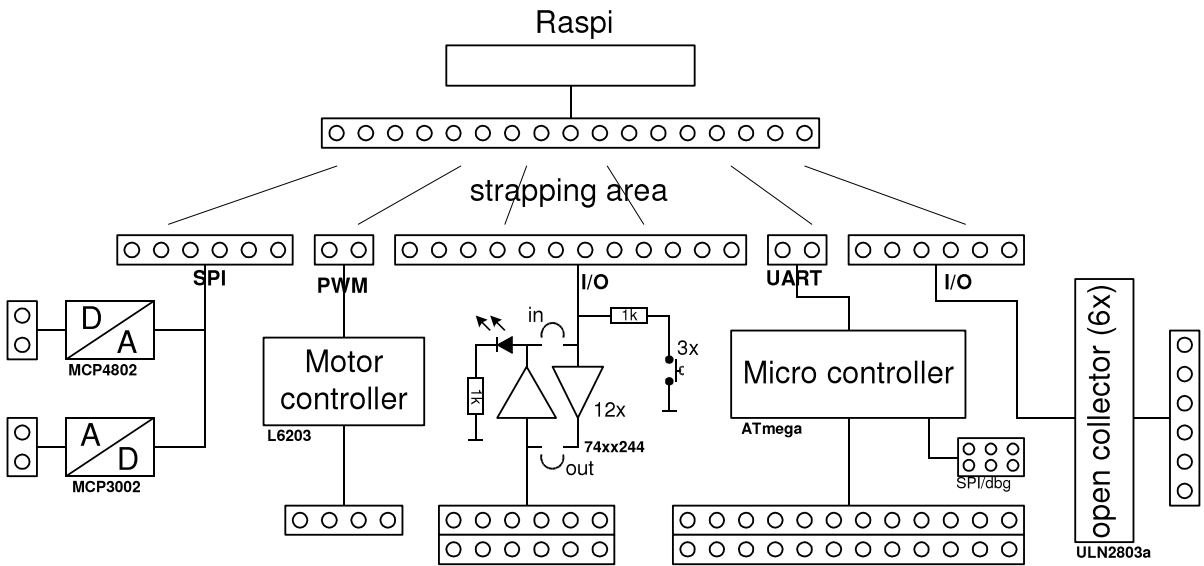


Fig. 1: The principle, high level diagram of the Gertboard. In this view it is possible to see how flexible Gertboard is, by being able to connect various parts of the board together.

Above is a principle diagram¹ of the Gertboard. Each circle in the diagram represents a header pin. These headers give you access to a wide range of control combinations. As you begin experimenting with the board, you will probably use the strapping area to connect various components on the Gertboard to the Raspberry Pi. This flexibility even allows you, for example, to connect the motor controller input pins to the Atmel ATmega device (an AVR microcontroller). The ATmega device has a separate 6-pin header, which allows it to be programmed by the Raspberry Pi using the (Serial Peripheral Interface) SPI bus.

The major building blocks are:

- 12x buffered I/O
- 3x push buttons
- 6x open collector drivers (50V, 0.5A)
- 48V, 4A motor controller
- 28-pin dual in line ATmega microcontroller
- 2-channel 8/10/12 bit Digital to Analogue converter
- 2-channel 10 bit Analogue to Digital converter

Each of these building blocks has a section below.

¹ A 'principle diagram' is a coarse overview of the most important parts of the system. It is not correct in all details. For that you must look at the board schematics.

Labels on the circuit board

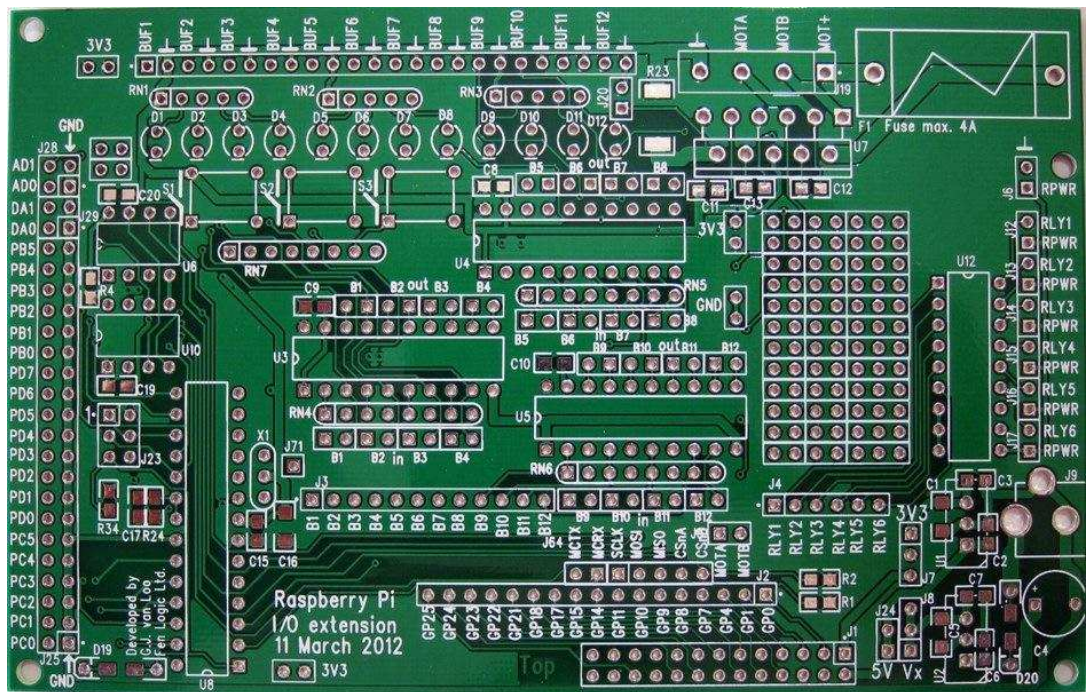


Fig. 2: A photograph of the unpopulated Gertboard viewed from above, showing the silver coloured holes and pads that eventually will be home to the components, as well as the legends printed in white epoxy ink, and green solder resist coating.

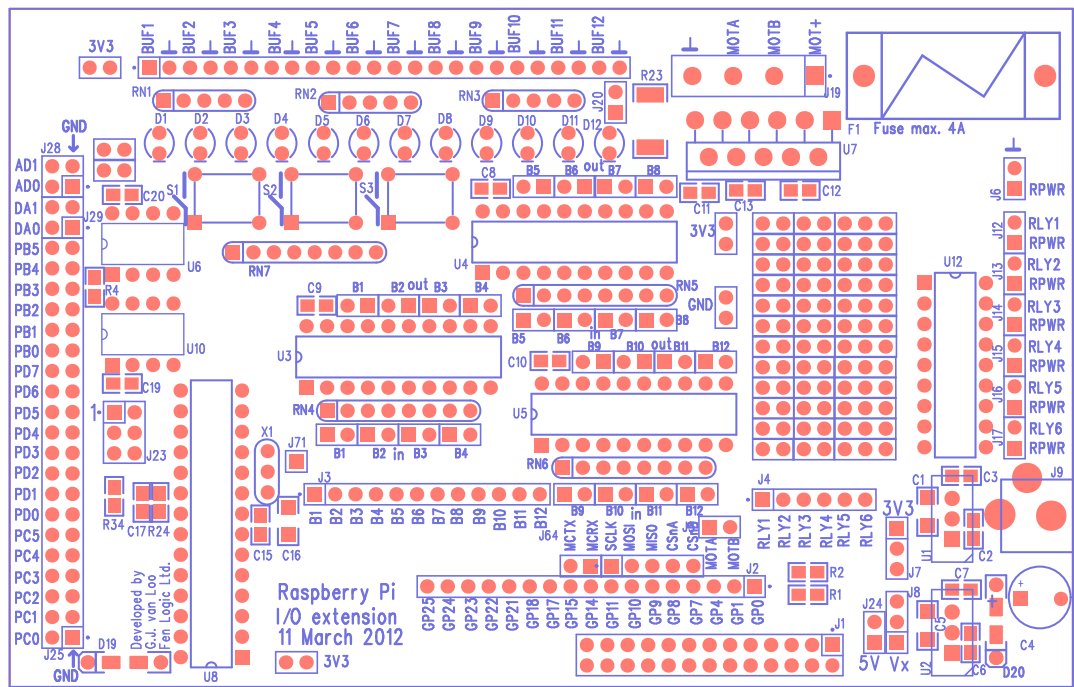


Fig. 3: This image is a diagrammatic representation of the same photograph shown in Fig. 2 above. It was generated from the same files that were used to create the physical printed circuit board. The blue elements in the diagram correspond to the white text and lines on the photo and the red elements correspond to the silver pads and holes on the photo.

From now onwards in this guide, because it is much clearer to see, the diagram shown in Fig. x will be used in preference to show you how to wire up the Gertboard, and to run the test and example programs.

It is useful to be able to look at the bare board in order to see the labels (the white text in the photo and the blue text in the diagram) on the board without the components getting in the way. These labels provide essential information that is required in order to use Gertboard to its full potential. 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 the majority of the components; such as resistors and capacitors (labelled with C_n and R_n , where n is some number). These are fairly simple devices that don't have a 'right way round' when they are assembled to the board. Diodes on the other hand, do need assembling the right way round (covered later) - all the diodes are labelled D_n ; of these, the ones that you will be interested in are D1 through D12, the light emitting diodes (LEDs; they are located near the top of the board on the left). Pushbutton switches are labelled S1, S2, and S3 (they are located just beneath the LEDs).

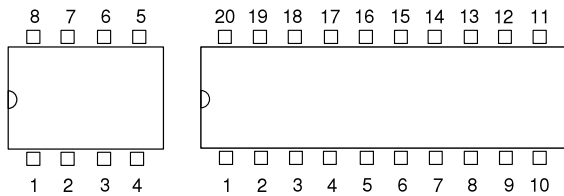


Fig. 4: Two examples of ICs – an 8-pin and a 20-pin dual-inline (DIL) package. In this package style, pin 1 is always identified as the first pin anticlockwise from the package notch marking.

Integrated circuits, or ICs, are marked U_n , so 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). For the ICs, it is very important to know which is pin 1. If the IC 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. On the Gertboard, the location of pin 1 is always marked with a square pad. Pin numbers increase in an anti-clockwise direction from there, as shown in the diagram. Knowing this means that the schematics in Appendix A can always be related to the pinning 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 , so for example the header to the ribbon cable from the Raspberry Pi is attached, is J1. Pin 1 on the headers is again marked with a square pad.

Power pins are marked with their voltage; for example there are a few positions marked 3V3. This is a commonly used notation in electronics, and in this case it means 3.3 volts. A 5V power supply comes onto the board via the GPIO connector, but the standard Gertboard assembly instructions do not require that a header is installed to access this. If 5V is really required, and spare header pins are available, a header can be soldered in location J24 in the lower right-hand corner of the board, and then a 5V supply can be picked up from the lower pin (next to the text '5V'). Ground is marked with GND or a \perp symbol.

Location of the building blocks on the Gertboard

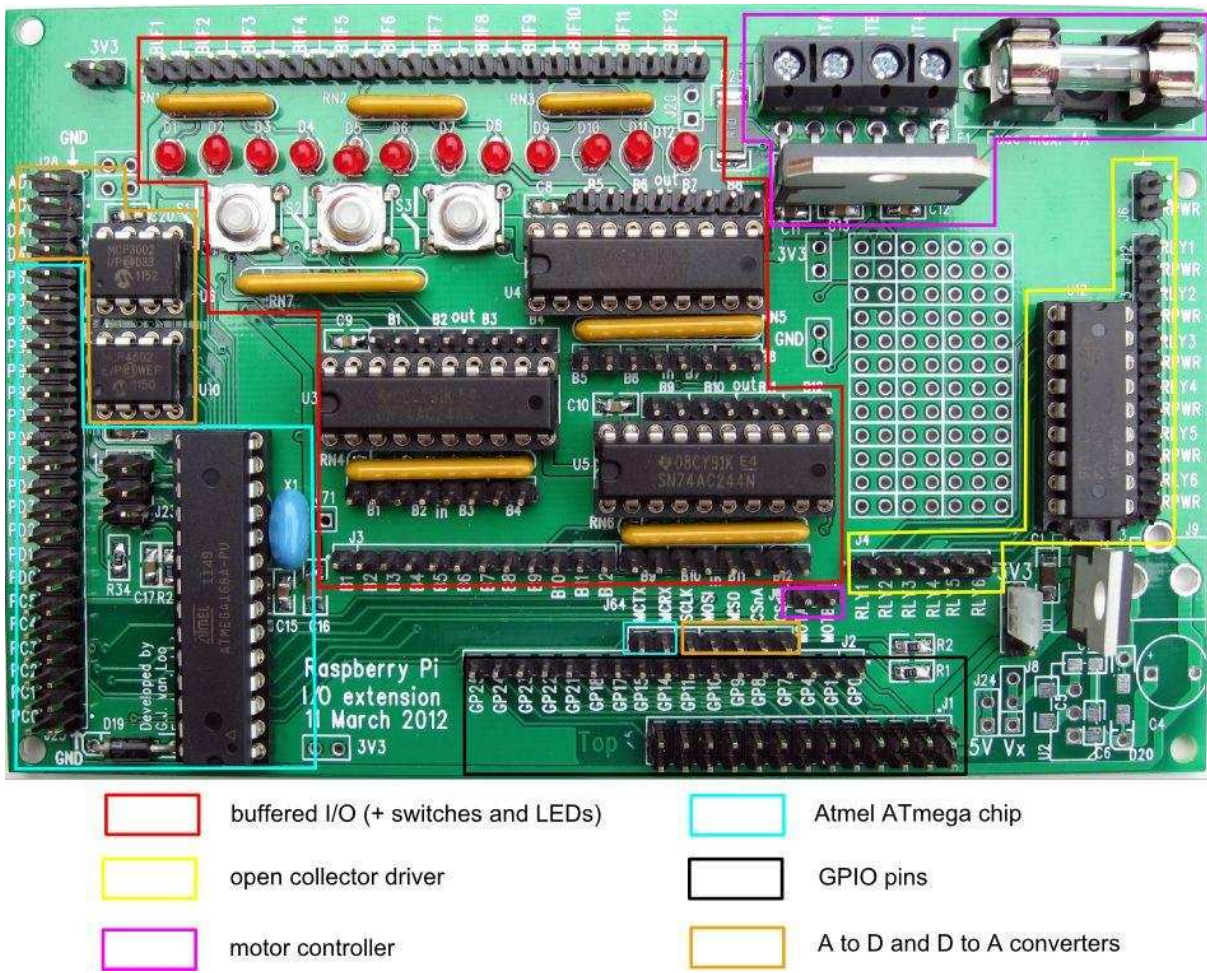


Fig. 5: Photograph of an assembled Gertboard, with key functional blocks identified by coloured boundary marking. This image serves as a good reference point for a board that has been successfully assembled from bare board and components. Please note that the appearance of some components can vary.

This annotated photo of a populated Gertboard shows where the building blocks (the major capabilities of the board) are located. Some of the building blocks have two areas marked. For example, the turquoise lines showing the Atmel ATmega chip not only surround the chip itself (on the lower left) but also surround two header pins near the bottom of the board, in the middle. These 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.

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 (the one where the ribbon cable to the Raspberry Pi is attached) by the components in the lower right corner of the board. The open collector and motor controllers can handle higher voltages and have points to attach external power supplies.

Jumpers and straps

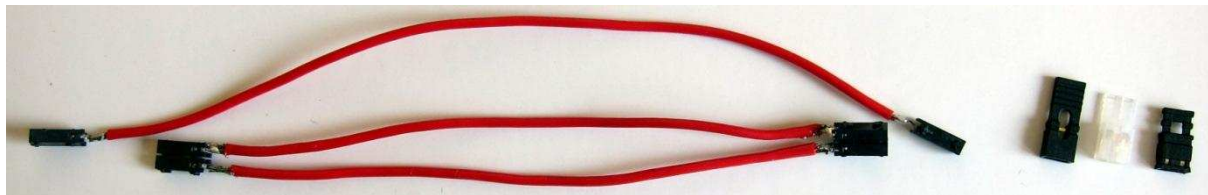


Fig. 6: Image showing straps on the left hand side, and jumpers on the right. Straps connect two parts of Gertboard together, whilst jumpers conveniently connect two adjacent pins on the same header, together. The Gertboard Kit contains materials to produce single straps, although the double strap also shown can also be useful.

To work properly, and get the maximum flexibility from the Gertboard a number of straps and jumpers are essential. On the left of the photo are straps: they consist of wires that connect the small metal connector and plastic housing, that slip over the header pins. They are meant for connecting header pins that are further apart. It is sometimes useful to have straps that connect two or three adjacent pins to the same number of adjacent pins elsewhere on the board. This is useful for example when you want to use several LEDs. On the right of the above photo are jumpers: they are used to connect two header pins that are right next to each other.

There is one jumper that should be in place at all times on the board: the one connecting pins 1 and 2 in header J7. This is the jumper that connects power from the power input pins to the rest of the board. It is near the lower right corner of the board and is the jumper connecting the two pins below the text 3V3 in the photo below.

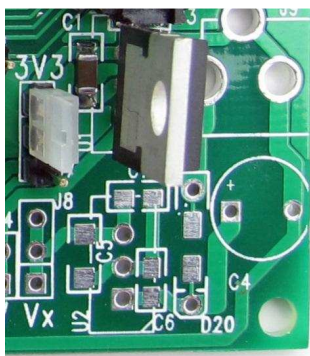


Fig. 7: Image showing header J7 with translucent jumper in place. J7 is located just above J8 (J7 legend is obscured in this image)

GPIO pins

The header J2, to the right of the text ‘Raspberry Pi’ on the board, provides access to all the I/O pins on the GPIO header. There are 26 pins in J1 (the GPIO header which is connected to the Raspberry Pi through the ribbon cable) but only 17 pins in J2: 3 of the pins in J1 are power 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. Signal GPIO*n* on the BCM2835 datasheet corresponds to the pin labelled GP*n* on header J2 (so for example, GPIO17 on the data sheet can be found at the pin labelled GP17 on the board). The numbers in the labels allow us to specify which pins are required in the control programs to be run later.

Some of the GPIO pins have an alternate function that are made use of in some of the test programs. These are shown in the table below. The rest are only used as general purpose input/output in the code. On page 27 there is a description of how to gain access to the alternate functions of GPIO pins.

GPIO0	SDA0 (alt 0)	I ² C bus
GPIO1	SLC0 (alt 0)	
GPIO7	SPI_CE1_N (alt 0)	SPI bus
GPIO8	SPI_CE0_N (alt 0)	
GPIO9	SPI_MISO (alt 0)	
GPIO10	SPI_MOSI (alt 0)	
GPIO11	SPI_SCLK (alt 0)	
GPIO14	TXD0 (alt 0)	UART
GPIO15	RXD0 (alt 0)	
GPIO18	PWM0 (alt 5)	pulse width modulation

Table 1: Table showing the GPIO pins on the Gertboard, and what their alternative function is.

We mention the I²C bus use of GPIO0 and 1 above not because the I²C 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 page 134).

Schematics

Whilst there are some circuit diagrams, or schematics, in the main body of the manual for some of the building blocks of the board, they are simplifications of the actual circuits on the board. To truly understand the board and the connections you need to make on it, you need to be a little familiar with the schematics. 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

When you download the Gertboard test/example code (available at www.element14.com/raspberrypi), you will have a file with a name something like `gertboard_sw_10_07_12.tar.gz`. This is a compressed (hence the `.gz` suffix, which means it was compressed using the `gzip` algorithm) archive (hence the `.tar`), where an archive is a collection of different files, all stored in 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 uncompress it by typing the following in one of the terminal windows on your Pi (substituting the name of the actual file you have downloaded for the file name we are using in this example):

```
gunzip gertboard_sw_10_07_12.tar.gz
```


Typing a directory command, `ls`, should then show the newly uncompressed archive file `gertboard_sw_10_07_12.tar`. So now, to extract the files from the archive, type

```
tar -xvf gertboard_sw_10_07_12.tar
```

A new directory, `gertboard_sw`, will be created. In it is 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. To compile all the code to run on Raspberry Pi, first change directory to `gertboard_sw` by typing:

```
cd gertboard_sw
```

And then in that directory, type:

```
make all
```

Each building block has at least one test program that goes with it. Currently the test programs are written in C; but they'll be translated into Python in the near future. 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 building blocks on the board. Each test has a C file that contains code specific to that test (thus you will find `main` 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 building 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. In the chart below is a summary of the purpose of the more commonly used macros and give 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;
```

Macro name	T	Explanation	Page no.
INP_GPIO(<i>n</i>)	E	activates GPIO pin number <i>n</i> (for input)	11
OUT_GPIO(<i>n</i>)	E	used after above, sets pin <i>n</i> for output	11
SET_GPIO_ALT(<i>n</i> , <i>a</i>)	E	used after INP_GPIO, select alternate function for pin	24
GPIO_PULL	W	set pull code	16
GPIO_PULLCCLK0	W	select which pins pull code is applied to	16
GPIO_IN0	R	get input values	16
GPIO_SET0	W	select which pins are set high	17
GPIO_CLR0	W	select which pins are set low	17

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

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)`.

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 Pi; in output mode it acts as output from the Pi. 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 to implement this.

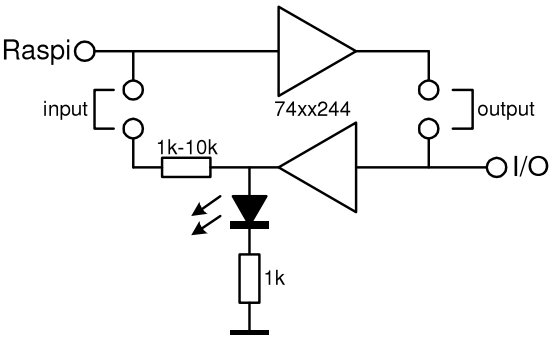


Fig. 8: The circuit diagram for I/O ports 4-12

The triangles symbols in the diagram above represent buffers. In order to make the port function as an input to the Raspberry Pi you install the ‘input’ jumper: 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 use the LED to check if the connect point is seen as high or low.

Depending on the type of 74xx244 buffer chosen, the LED could behave randomly if the port is not driven properly. In that case it may easily switch state, switching on or off with the smallest of electronic changes, for example, when the board is simply touched.

There is a series resistor between the input buffer and the GPIO port. This is to protect the BCM2835 (the processor on the Raspberry Pi) in case the user programs the GPIO as output and also leaves the ‘input’ jumper in place. The BCM2835 input is a high impedance input and thus even a 10K series resistor will not produce a noticeable change in behaviour when it is used as input.

Push buttons

The Gertboard has three push buttons; these are connected to ports 1, 2, and 3. Thus the first three I/O ports look like this:

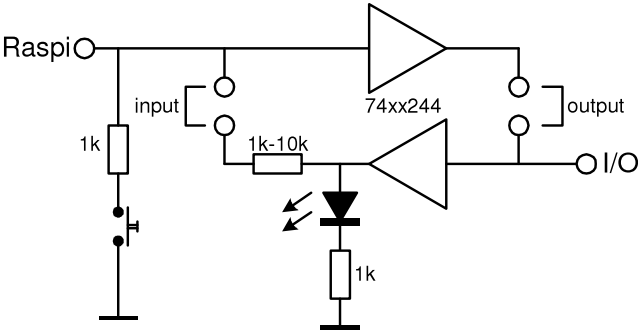


Fig. 9: Circuit diagram showing one of the three push buttons I/Os. There is a circuit like this for ports 1 to 3.

In order to use a push button, 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 push buttons, a pull-up must be set on the Raspberry Pi GPIO pins used (described below, page 16) so that they are read as high (logical 1) when the buttons are not pressed.

Locating the relevant sections of the Gertboard

In the building blocks location diagram on page 7, 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 (the round translucent red plastic devices) are labelled D1 to D12; D1 is driven by port 1, D2 by port 2, etc. The pushbutton switches (the silver rectangular devices with circular depressions in the middle) are labelled S1 to S3; S1 is connected to port 1 and so on. The long thin yellow components with multiple pins, are resistor arrays.

The pins corresponding to ‘Raspi’ in the circuit diagrams above are B1 to B12 on the J3 header above the words ‘Raspberry Pi’ on the board (B1 to B3 correspond to the ‘Raspi’ points on the second circuit diagram with the pushbutton, and B4 to B12 correspond to the ‘Raspi’ points on the first circuit diagram). They are called ‘Raspi’ because these are the ones that should be connected to the pins in header J2, which are directly connected to the pins in J1, and which are then finally connected via the ribbon cable to 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 apparent 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 bottom.

The buffered I/O ports can be used with (almost) any of the GPIO pins; they just have to be connected up using the 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 push buttons *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 a yellow resistor array), 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 bare board), 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).

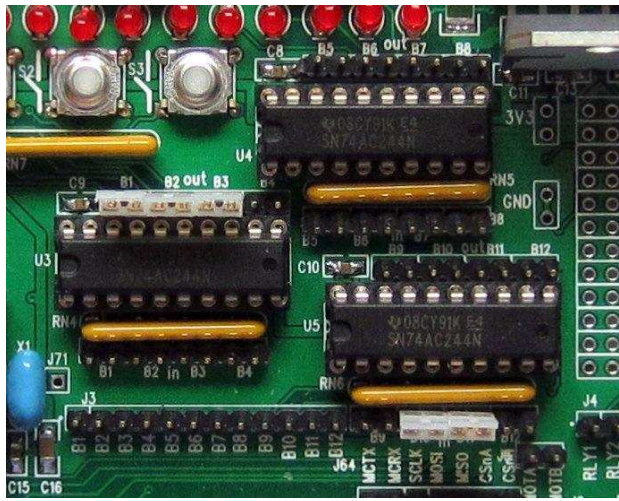


Fig. 10: Example of port configuration where ports 1 to 3 are set to be outputs and ports 10 and 11 are set to be inputs.

As a concrete example, in the picture above, 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).

In the test programs, the required connections are printed out before starting the tests. 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

The test program for the pushbutton switches is called `buttons`. To run this test, the Gertboard must be set up as in the image below. 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.

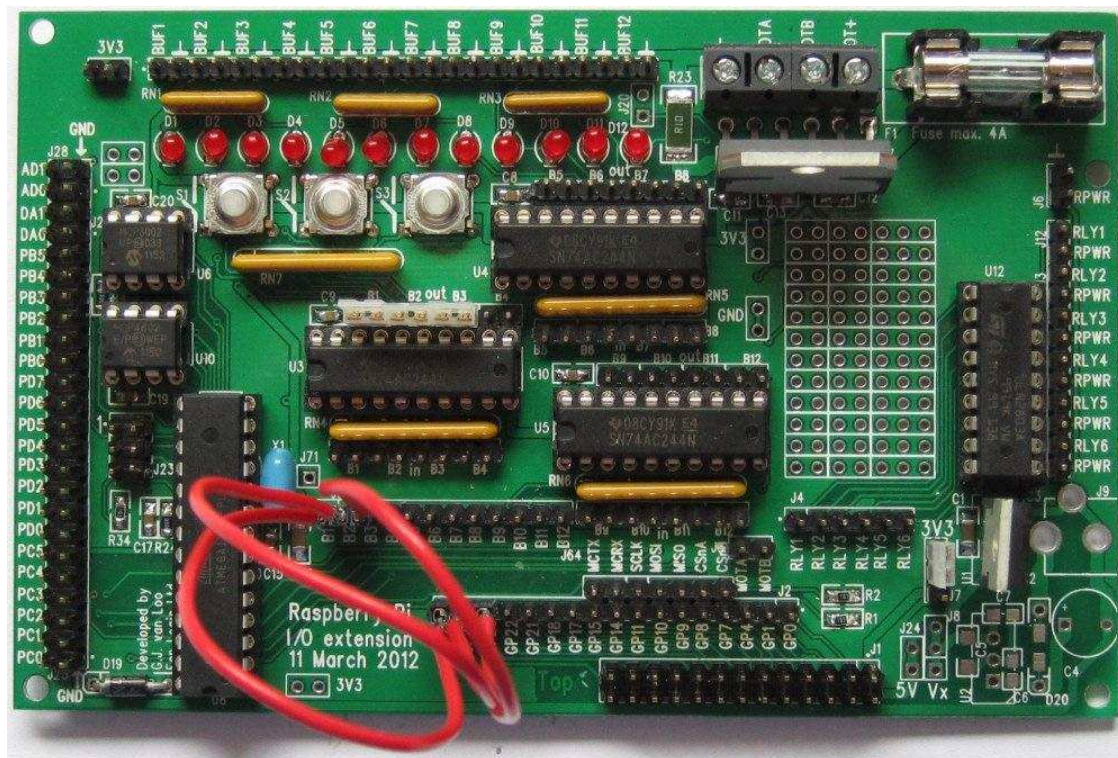


Fig. 11: Whilst the image above is clear, it isn't very good at showing exactly how the straps are connected, and between which pins on the board.

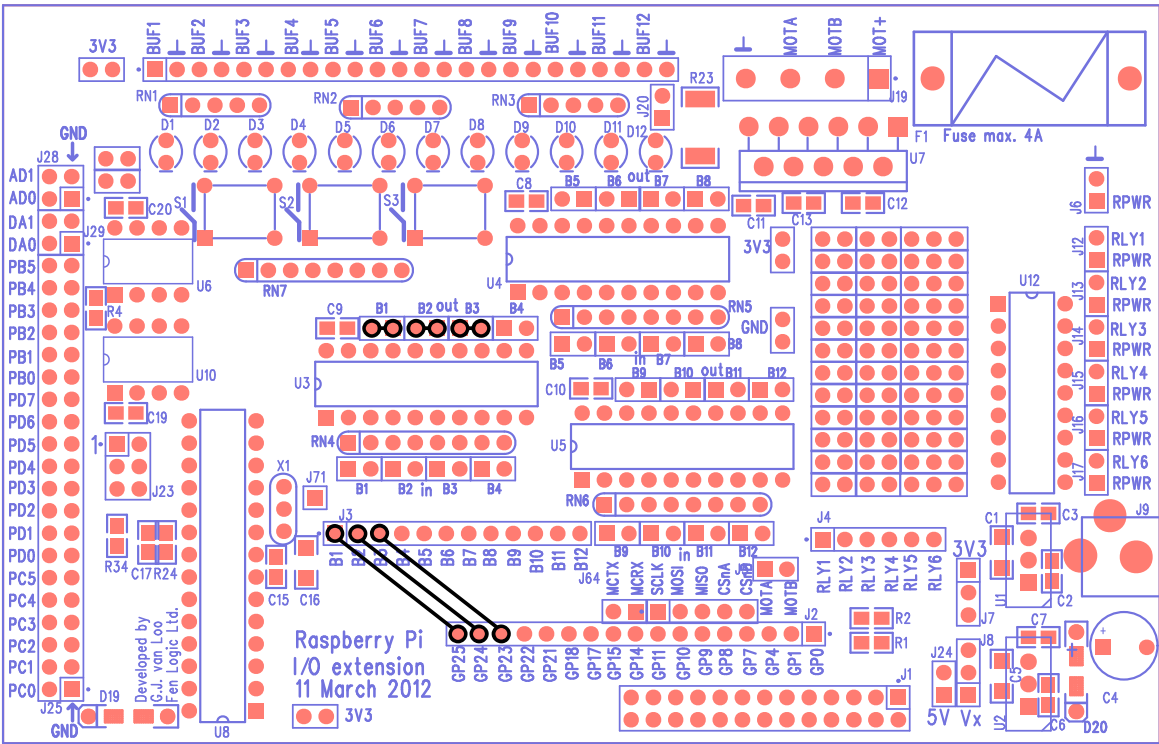


Fig. 12: This type of diagram is much more effective at showing how straps connect pins together on the board, so from now onwards, we will use these type of diagrams to show wiring arrangements.

In the diagram, 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.

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` is called (described on page 10) 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 allow 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_PULLCLK0 = 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.

Testing the LEDs

The test program for the LEDs is called `leds`. To set up the Gertboard to run this test, see the wiring diagram below. 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 a pin is written to that has nothing connected to it.

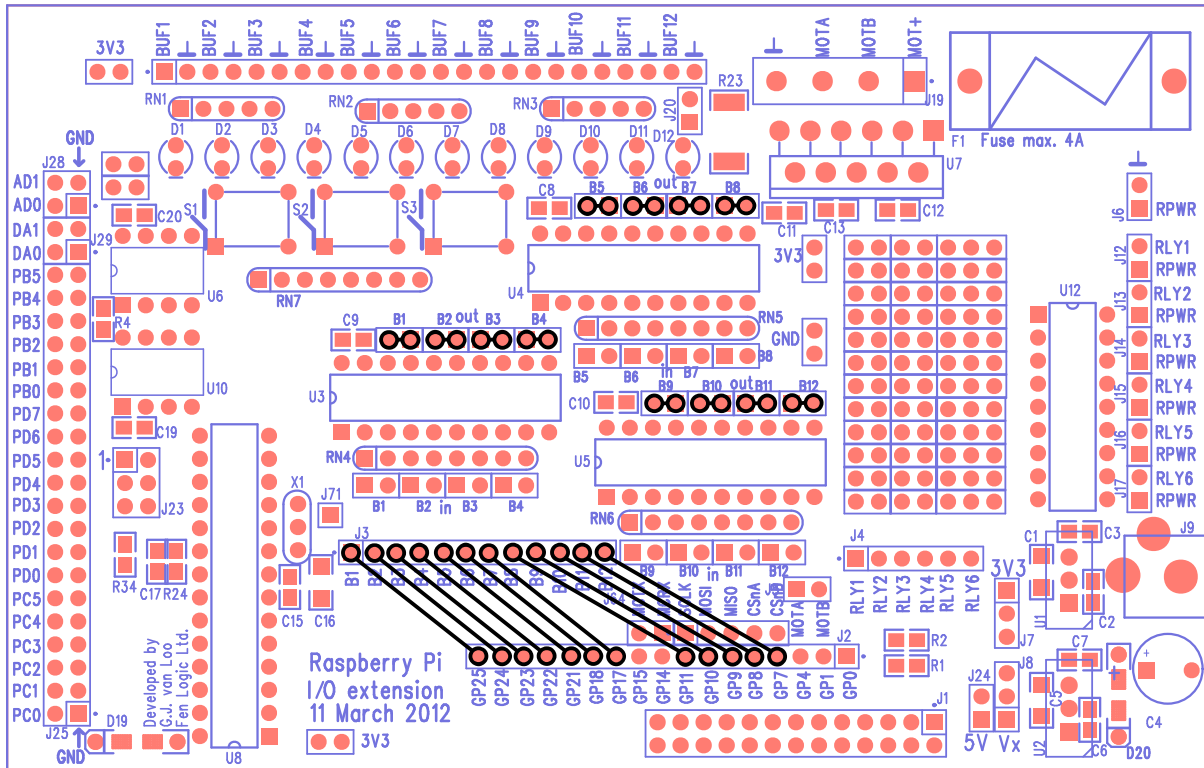


Fig. 13: The wiring diagram necessary to run the Gertboard LED test program, 1eds

The test code in `1eds.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 `1eds.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.

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) 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.

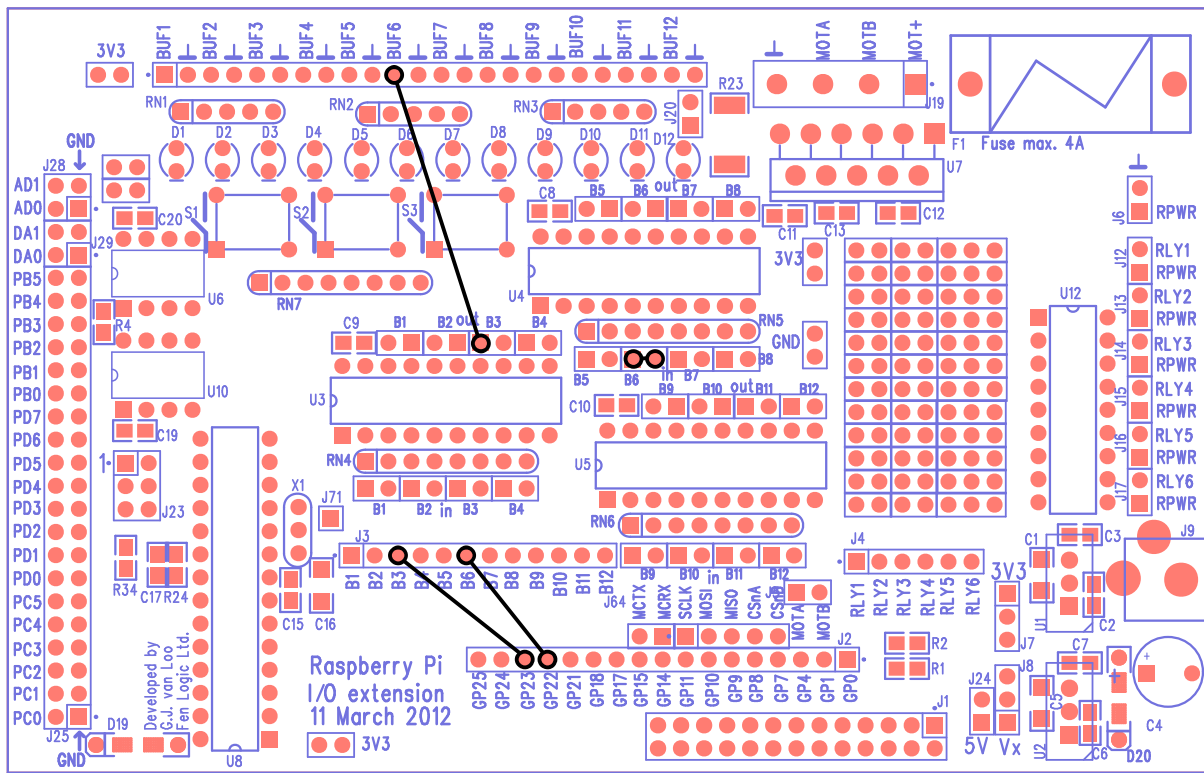


Fig. 14: The wiring diagram for test program `butLed` which detects a button press, and then display that button state on the screen. This is to test all the I/O on the Gertboard.

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 is not installed (which would be placed at U3-out-B3).

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 the U3-out-B3 header pin just above pin 14 on the chip (it is pin 1 of U3-out-B3; this is clear from the schematic and from the fact that this pin has a square pad on the bare circuit board), so that pin is connected 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).

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 16, in the `buttons` test. Then the `GPIO` values are repeatedly read in, and the binary values of `GPIO22` and `GPIO23` are printed out, if they have changed since the last cycle. So if '01' is displayed on the monitor, it can be deduced 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 if the test is started with button 3 up (so '11' is displayed), and then the button is pushed down, occasionally '01' might be seen, followed very quickly by '00'. The reason for this differs between the Python and C implementations. In the C version, both values are read at the same time, and the signal from the push button (which is connected to `GPIO23`) takes a small amount of time to propagate through the buffers to get to `GPIO22`.

It may even be possible to get one reading in after `GPIO23` has changed, but insufficient time has passed for `GPIO22` to change state and follow it! In the Python code, the read of `GPIO22` occurs before the read of `GPIO23` (the button). Thus if the button is pressed or released between these two reads, the new value will be read in for the button (`GPIO23`), but the new value of the other input (`GPIO22`) won't change 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 need a different voltage or higher current than that available on the Gertboard and are powered by an external power supply. 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 below).

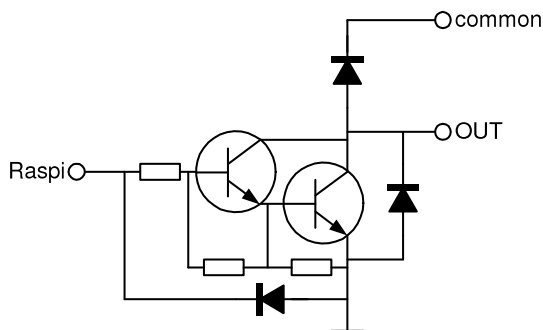


Fig. 15: Circuit diagram of each open collector driver.

The ‘common’ pin is, as the name states, common for all open collector drivers. It is not connected to any other point 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.

On the Gertboard building block diagram on page 7, the area containing the components for the open collector drivers are outlined in yellow. The pins corresponding to ‘Raspi’ in the diagram above 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 program `oc01` (for open collector) allows the functional testing of the open collector drivers. A simple mechanism was required to switch the driver on and off, so we created a little circuit (see diagram below) consisting of two large LEDs and a resistor in series. 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.

Since this small test circuit will not be used again, it can simply be hand soldered together off-board. Remember that LEDs are diodes, and have to be connected the right way round. The small ‘flat’ in the LED moulding denotes the ‘cathode’ or negative pin. If you think of the LED symbol in the circuit diagram below as an arrow, it is pointing in the direction of the current flow, from + to -, or from anode to cathode.

To turn the circuit off and on using the open collector driver (say you want to use driver 1), first check that it works with the power supply described above. 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 \perp pin on the board. Now, we need a signal to control the driver. For the `oc01` test we are using 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 , with the `oc01` test, 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 ends up powering the circuit: it is just turned off and on by the open collector driver.

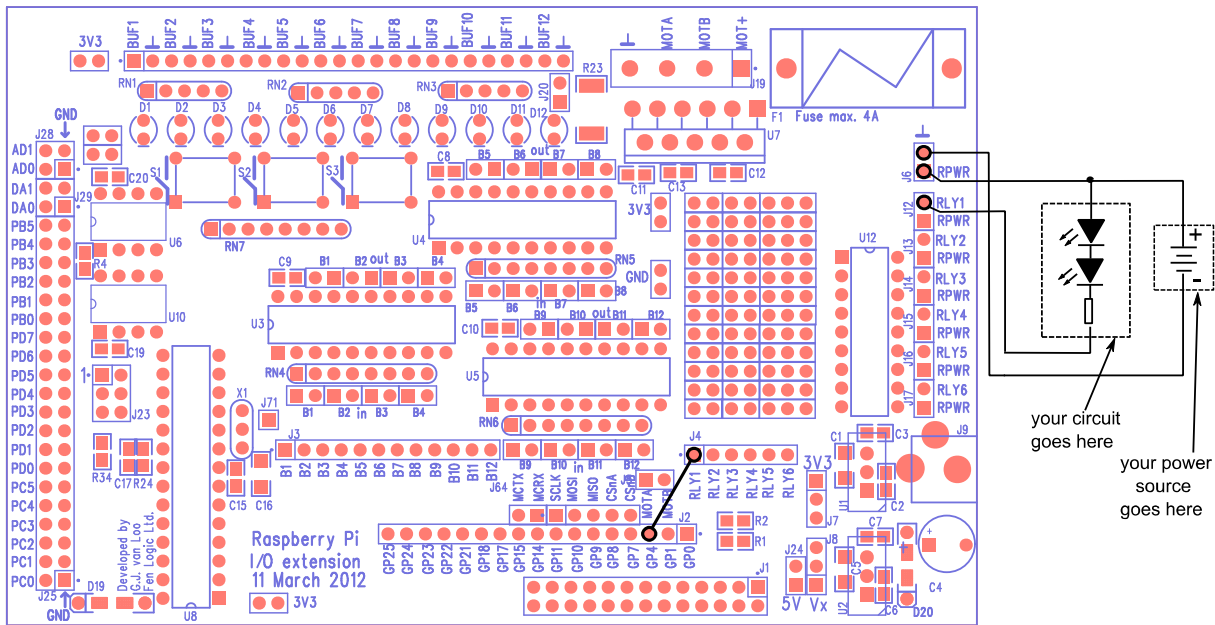


Fig.16: Wiring diagram showing how to connect Gertboard to test the open collector drivers. It also shows the small test power supply made up of two LEDs in series, a 90 Ω resistor and a 9V battery.

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 causes the voltage on RLYn pin 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 RLYn pin to the RPWR. This allows current to flow to the top (positive side) of your circuit, allowing the energy to dissipate, and preventing damage.

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 11). Then in it uses `GPIO_SET0` and `GPIO_CLR0` (described on page 17) 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.

Motor Controller

The Gertboard has a position for a L6203 (Miniwatt package) motor controller. The motor controller is for brushed DC motors.

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 building block diagram on page 7, 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 a small header just above GP4 and GP1 in header J2. The MOTA and MOTB pins just above header J2 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 near J2.

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 per the table 3 above. 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 in the diagram below:

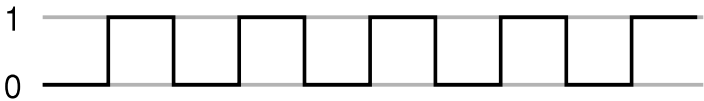


Fig. 17: An example of a PWM output. In this example the output is neither on nor off all the time. In fact, here it is on for 50% of the time, and is therefore said to have a *duty cycle* of 50%.

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. The diagram above shows a 50% duty cycle; the one below is 25%.

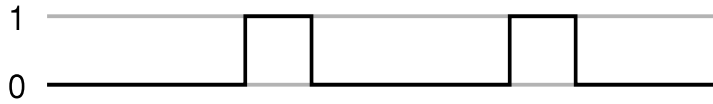


Fig. 18: In this PWM example, the duty cycle is 25%.

There is a PWM in the BCM2835 (the Raspberry Pi processor), and it's 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 set the other motor controller input (MOTB in our test) to a steady high or low, the speed and direction of the motor can be controlled.

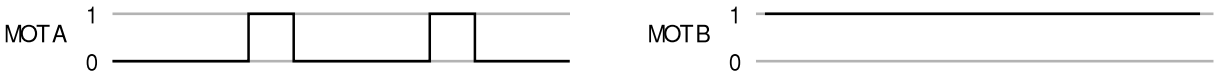


Fig. 19: The motor direction is set by MOTB. Whilst MOTA has a duty cycle of 25%, the motor only receives power when MOTA and MOTB are different, thus it receives power for 75% of the time.

For example, in the diagram 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 $\frac{3}{4}$ speed.

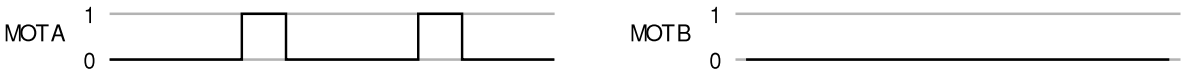


Fig. 20: In this example, the truth table predicts that the motor will run in the opposite direction at around 25% speed.

If on the other hand you set MOTB low, as in the diagram 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 $\frac{1}{4}$ speed.

Testing the motor controller

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 test program for the motor controller is called `motor`. To set up Gertboard for this, connect GP17 in J2 to the MOTB pin (the MOTB pin in the 2-pin header above GP1 and GP4, not the one at the top of the board), and GP18 to MOTA in that little header. 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 \perp screw terminals. This is shown below.

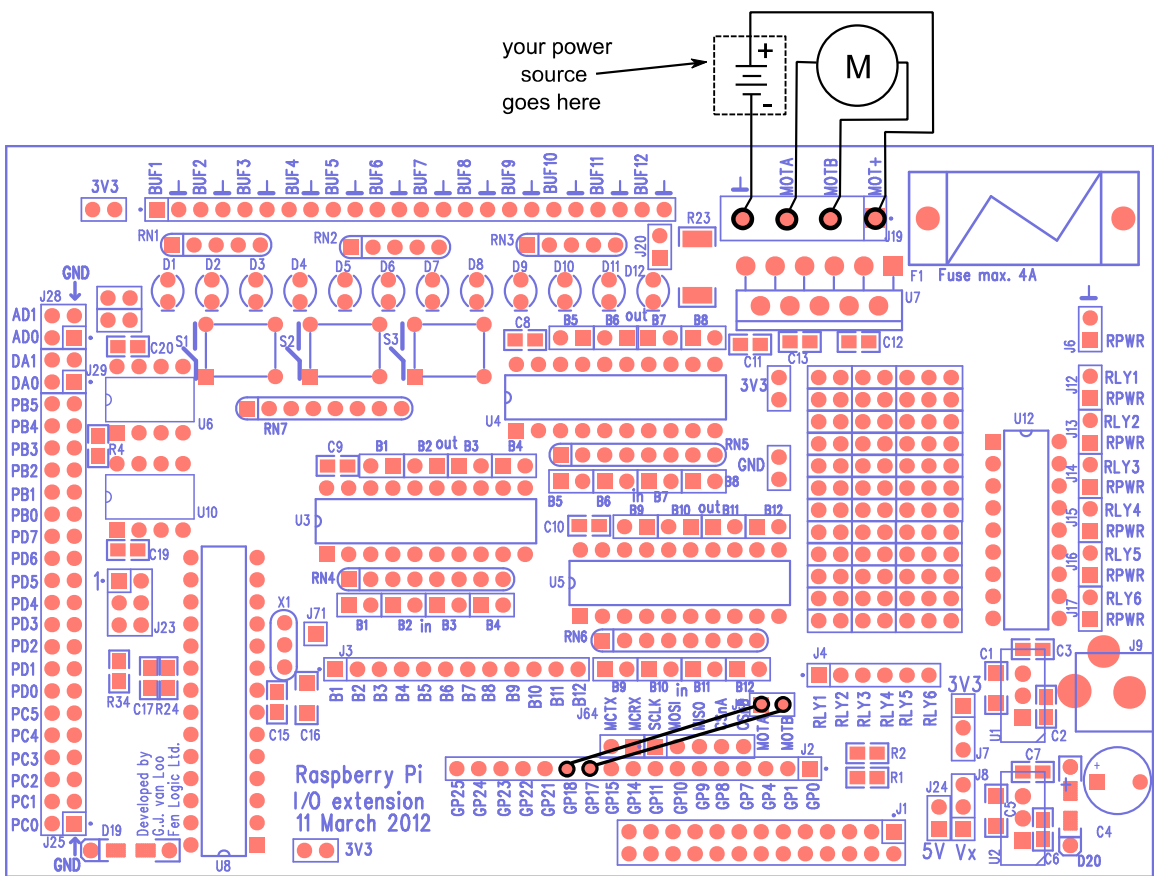


Fig. 20: The wiring diagram for the test program motor .

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 11 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 for the GPIO pins. A summary of the alternate function of GPIO pins used on the Gertboard, see the table on page 9.)

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 22.

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 22), then slow it down again. Finally the PWM is switched off, and the GPIO interface is closed down.

Digital to Analogue and Analogue to Digital Converters

In the Gertboard building blocks diagram on page 7, the components implementing the converters are outlined in orange. Both the analogue converter (D/A) and analogue to digital converter (A/D) are 8-pin chips from Microchip. The D/A is U6 (above) and the A/D is U10 (below). 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 building 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 chips 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 page A-6.

Digital to analogue converter

The Gertboard uses a MCP48xx digital to analogue converter (D/A) from Microchip. The device comes in three different types: 8, 10 or 12 bits. It is likely that MCP4802, the 8 bit version, will be used, but if higher resolutions are needed, it can be replaced with the MCP4812 (10 bits) or MCP4822 (12 bits). 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 "Testing the D/A and A/D" section). 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. 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

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 (which is probably the one you are using) ignores the last four. Since you could use any of those chips on the Gertboard, `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`.

The value on the output pin, V_{out} , is given by the following formula (assuming the 8-bit MCP4802):

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

To test the D/A, a multimeter is required. The test program for this is `dtoa`. 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 multimeter as follows: the black lead needs to be connected to ground. You can use any of the pins marked with \perp 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.

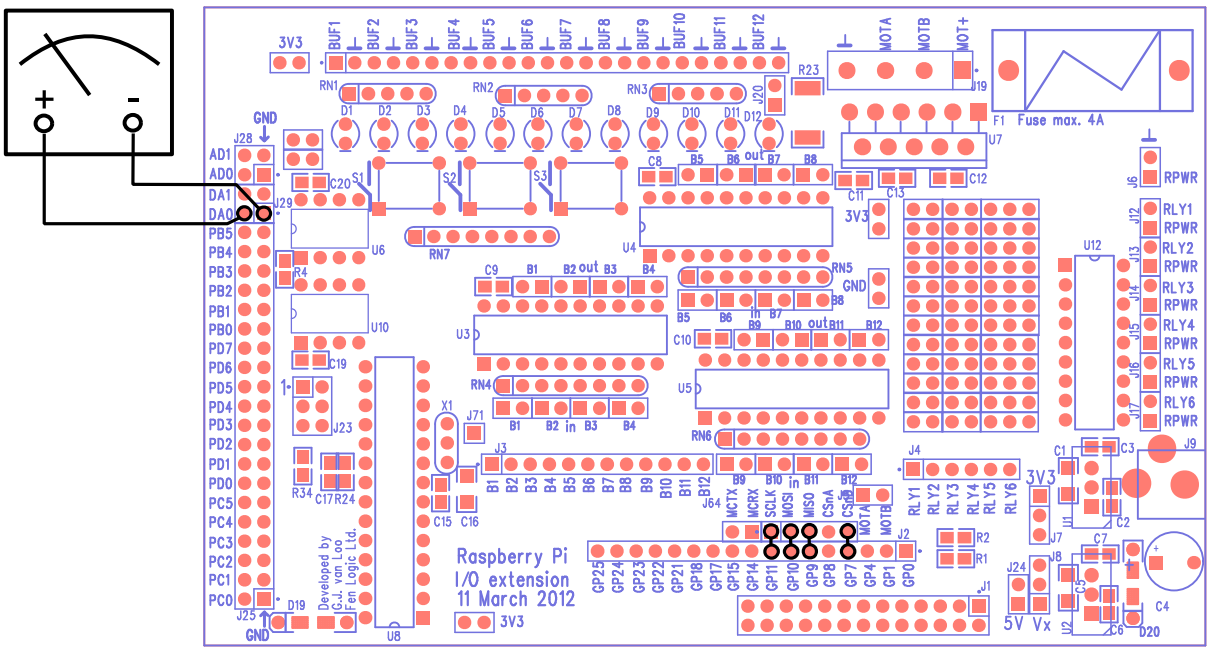


Fig. 21: The wiring diagram required to measure the output from the D to A converter fitted to the Gertboard whilst running the test program `dtoa`.

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.

The test program for the A/D is called `atod`. 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 \perp), 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.

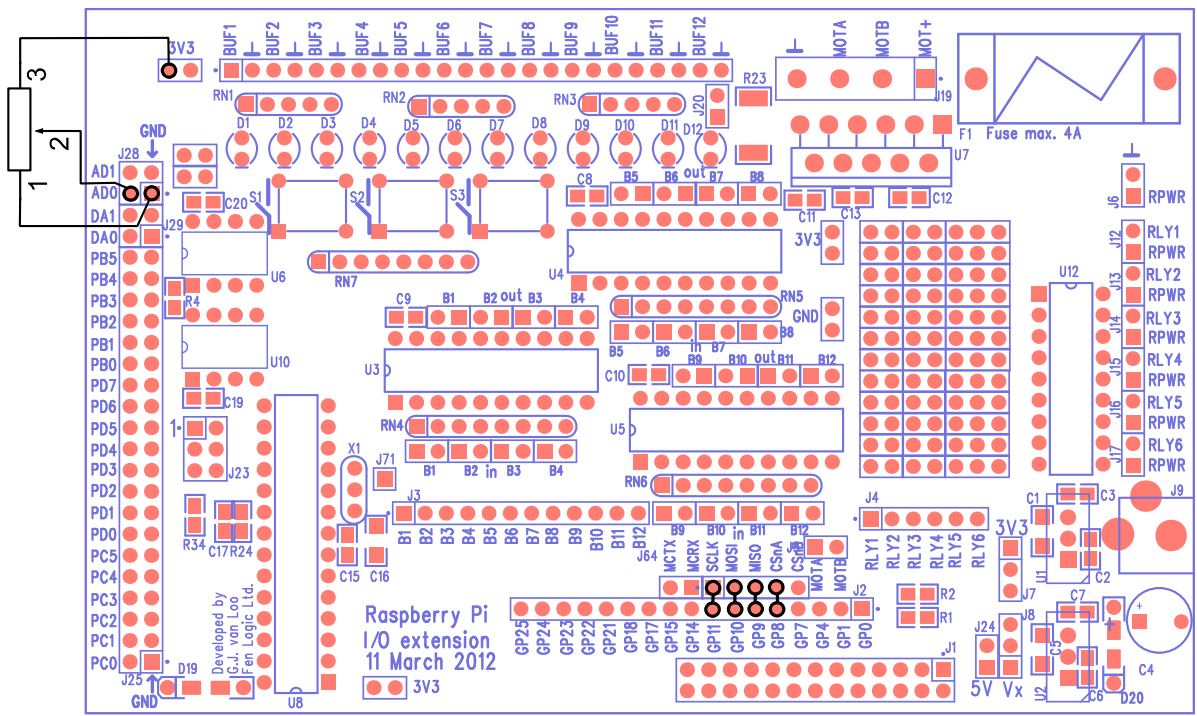


Fig. 22: Wiring diagram showing how the Gertboard is connected to verify that the A/D converter is working properly, using the test program atod.

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.

Even without a multimeter 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.

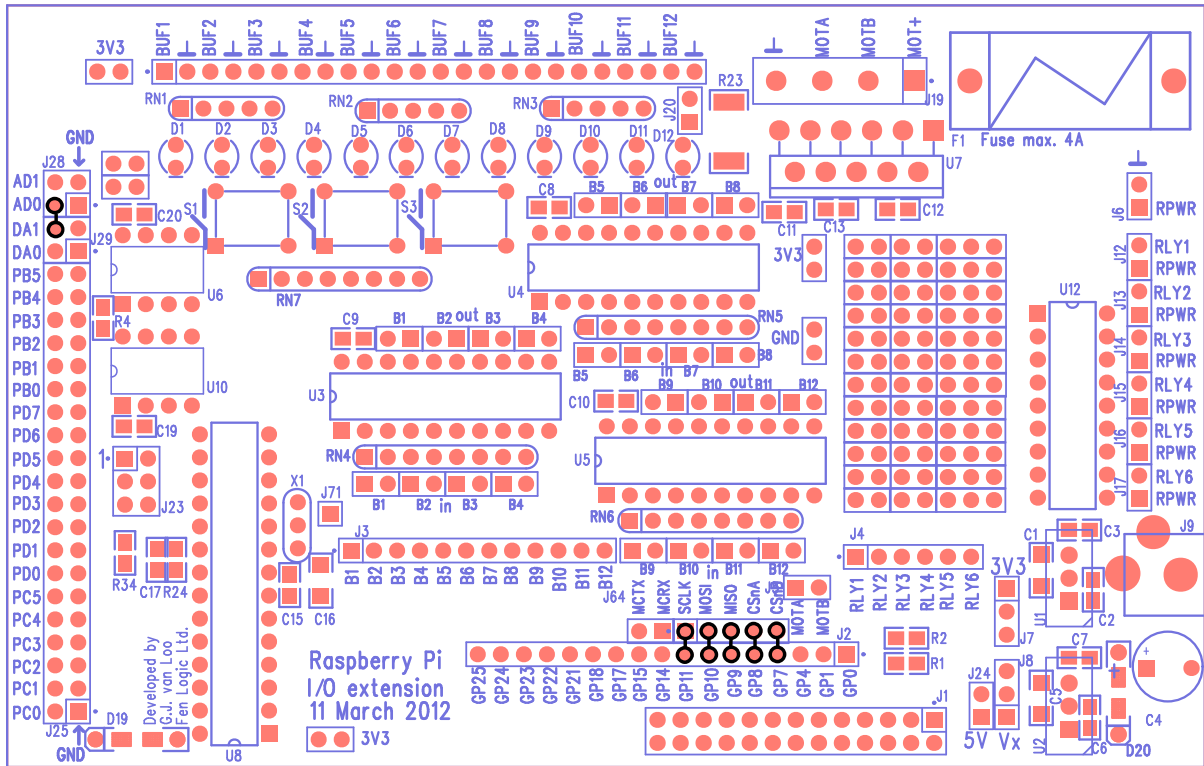


Fig. 23: The wiring diagram for an alternative method of testing the A/D and D/A converters together, without the aid of a multimeter and potentiometer.

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.

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. 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. It is also 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 is 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 immensely 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. The Atmel chips most commonly used on the Gertboard are the ATmega168 and ATmega328, so Gordon assumes you have one of these.

To use Gordon's system, first you need to install the Arduino IDE. Then you download a custom version of `avrdude`, which allows you to program the AVR microcontroller using the SPI bus. (GPIO pins GPIO7 through GPIO11 can be used as a SPI bus.) Then you have to edit various configuration files to fully integrate the Gertboard into the Arduino IDE. Finally, you have to program the 'fuses' on the ATmega chip. Happily, Gordon has written some scripts to do all this for you. Full instructions, scripts, and the modified `avrdude` are available at:

<https://projects.drogon.net/raspberry-pi/gertboard/> We assume now that you have downloaded and successfully installed and configured the Arduino IDE, as described above, 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 version number is given in the title bar. The Debian squeeze package is version 0018, while the wheezy package is 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 (ATmega168 or ATmega328). For IDE version 1.0.1, you will also have to 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 pins 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
digital 0	PD0	digital 7	PD7	analogue 0, A0	PC0
digital 1	PD1	digital 8	PB0	analogue 1, A1	PC1
digital 2	PD2	digital 9	PB1	analogue 2, A2	PC2
digital 3	PD3	digital 10	PB2	analogue 3, A3	PC3
digital 4	PD4	digital 11	PB3	analogue 4, A4	PC4
digital 5	PD5	digital 12	PB4	analogue 5, A5	PC5
digital 6	PD6	digital 13	PB5		

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

In both versions of the Arduino IDE, digital pins are referred to in the code 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 handled slightly differently. In version 0018, analogue pins are referred to simply by number, so whether 0 refers to PD0 (a digital pin) or PC0 (an analogue pin) depends on the context. The command

```
value = digitalRead(0);
```

will cause a read from digital 0 (PD0), and `value` will be assigned LOW or HIGH, while the command

```
value = analogRead(0);
```

will cause a read from analogue 0 (PC0), and `value` will be assigned a number between 0 and 1023, as the A/D converters in the ATmega chip return 10 bit values.

In version 1.0.1, however, although numbers 0 through 5 still work to specify analogue pins, they are referred to in the examples as A0 to A5, and this seems to be the preferred style now. So to read from analogue pin 0 you would use the command

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

A few sketches to get you going

A good first sketch to try is `Blink`, which makes an LED turn on and off. With version 0018 of the IDE it's in the File > Examples > Digital menu; in 1.0.1 it's in 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.

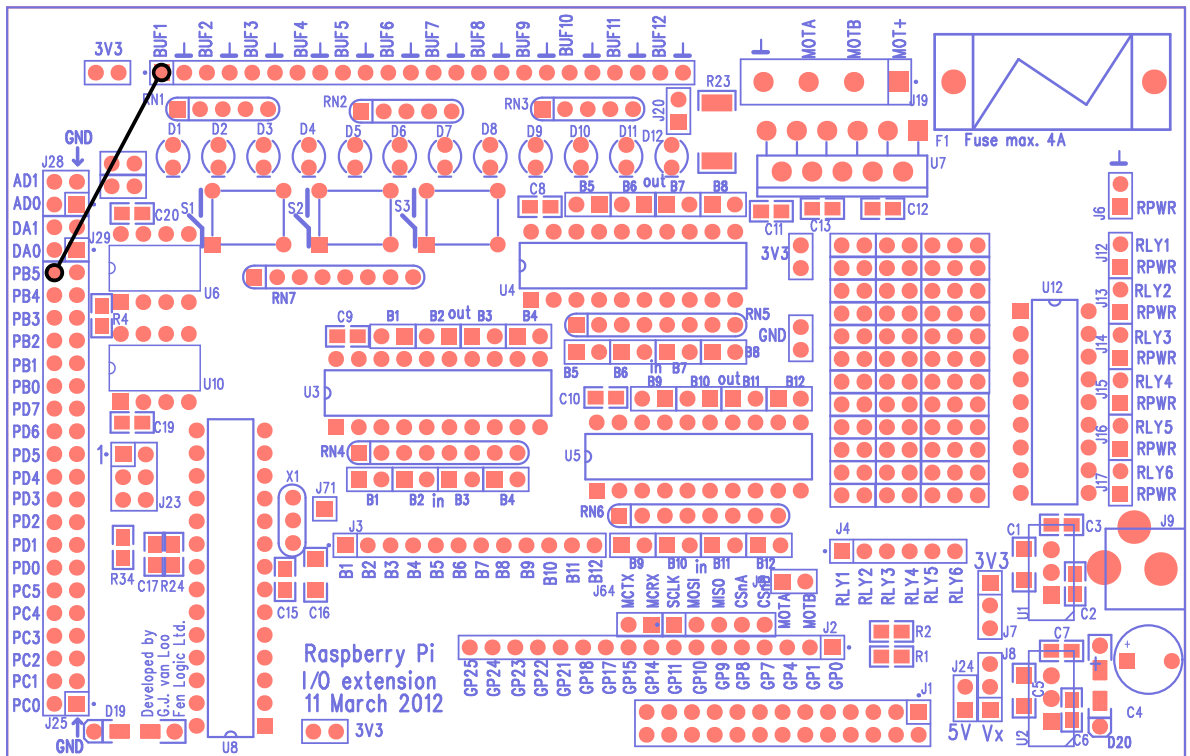


Fig. 24: Wiring diagram for the sketch `Blink`.

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.

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.

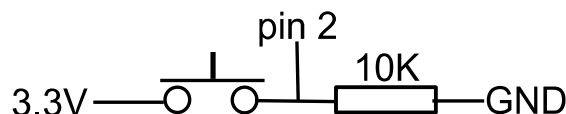


Fig. 25: Suggested switch circuit for use with `Button` sketch.

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

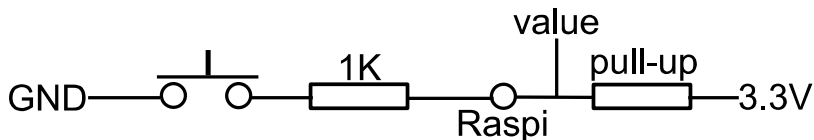


Fig. 26: Circuit actually in use on the Gertboard, showing an additional 1k resistor to protect the input to BCM2835.

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 push button 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 16). 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 digital 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.

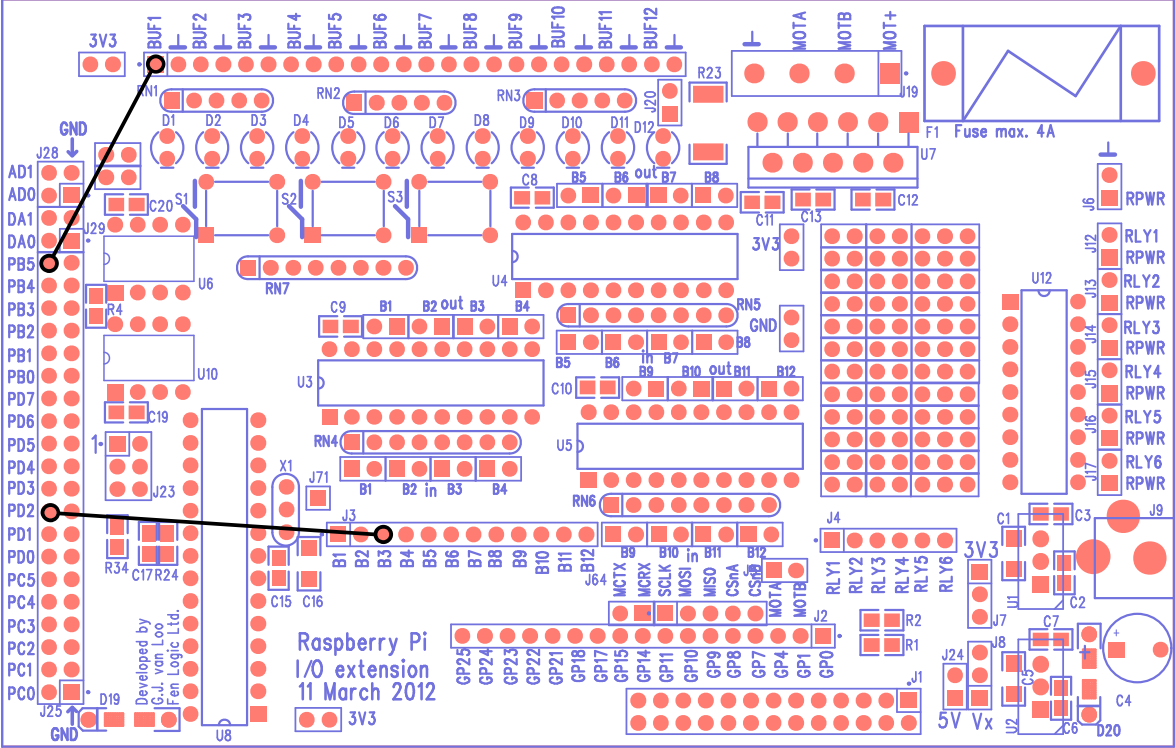


Fig. 27: Wiring diagram showing the additional strap necessary for button operation for the sketch Button .

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.

Now let’s try using an analogue pin. Find the `AnalogInput` sketch under `File > Examples > Analog` (in both versions 0018 and 1.0.1). 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. As above, 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.

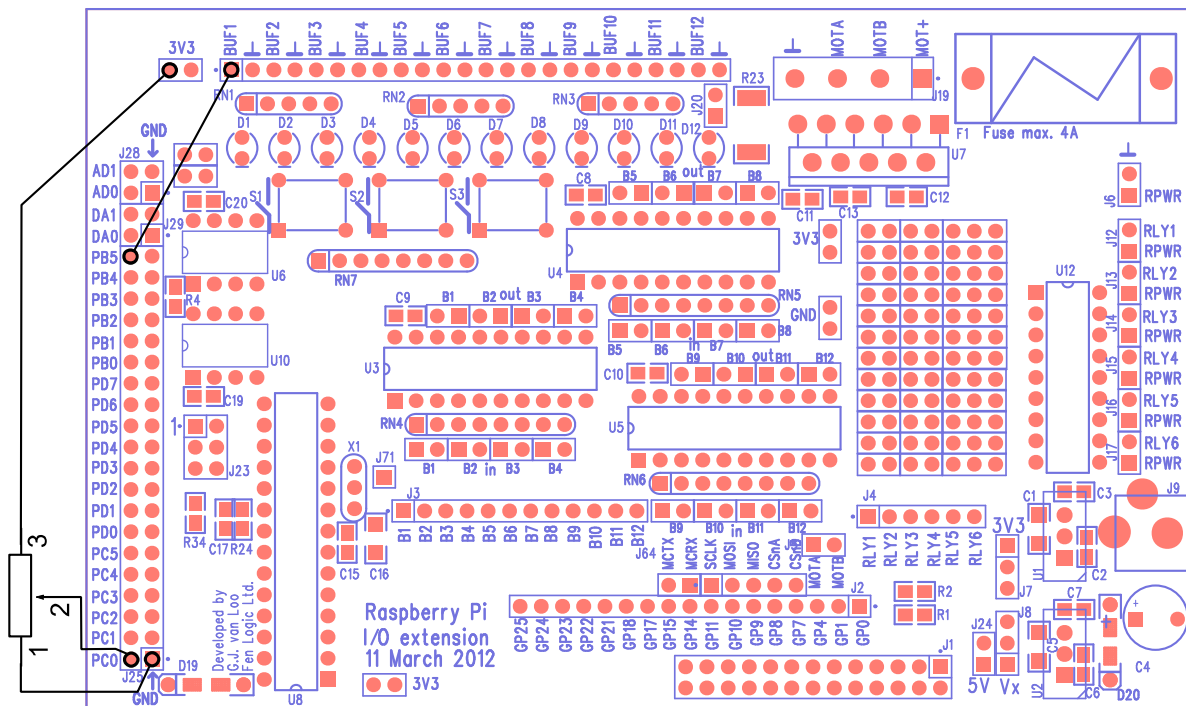


Fig. 28: Wiring diagram for the AnalogInput sketch.

Minicom

Some of the Arduino sketches involve reading or writing data via the serial port, or UART. An example is `AnalogInSerial` under `File > Examples > Analog` for version 0018. In version 1.0.1, this same example has been renamed `AnalogReadSerial` and is under `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 described above. 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.

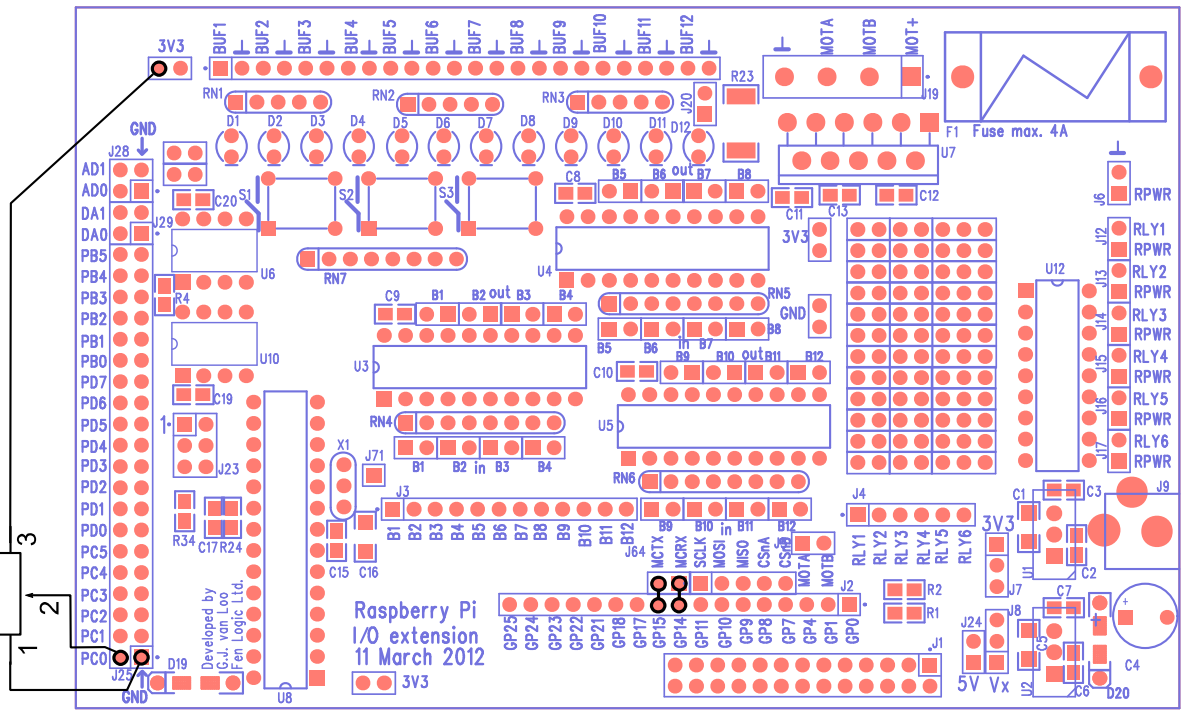


Fig. 29: Wiring diagram for the sketch `AnalogInSerial/AnalogReadSerial`.

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 on page 9, 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.

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.

Combined Tests

This section shows some examples of using more than one building 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, and 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. 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).

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.

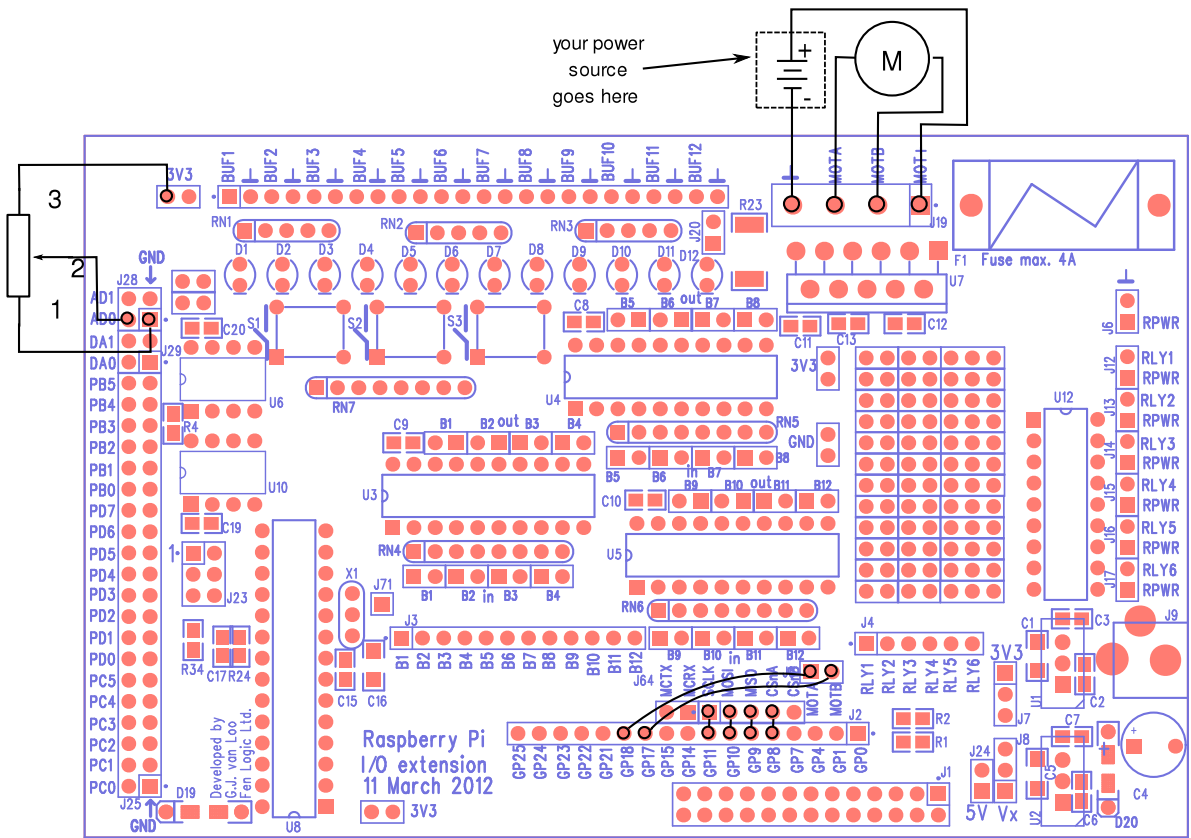


Fig. 30: Wiring diagram for the combined potmot test.

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 A/D and D/A converters (page 27). GPIO17 is set up as an output (using `INP_GPIO` and `OUT_GPIO`), and GPIO18 is set up as a PWM using `INP_GPIO` and `SET_GPIO_ALT` as described in the section on the motor controller (page 24). 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 on page 22. 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.

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, GP1 to 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).

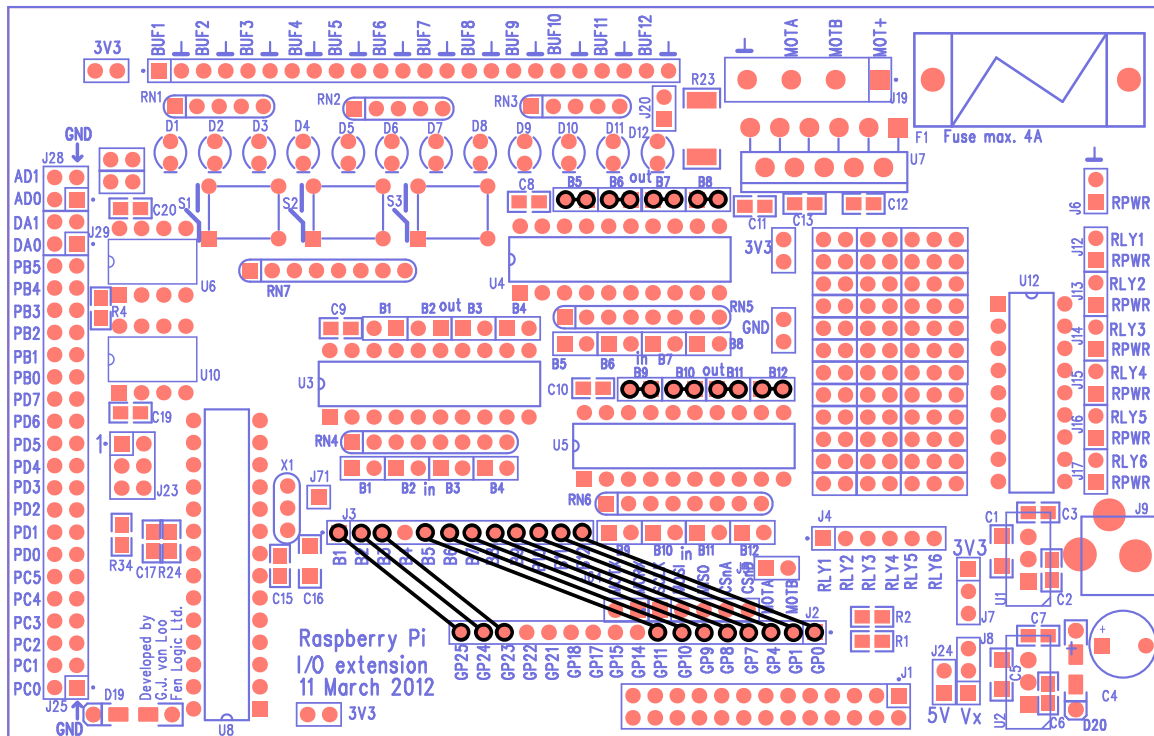


Fig. 31: Wiring diagram for the decoder test.

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 16) and to set GPIO11 to GP7, GPIO4, GPIO1, and GPIO0 up as outputs (as described on page 11). 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 17.

For More Information

For further information, 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.