Hardware design with RP2040

Using RP2040 microcontrollers to build boards and products

Colophon

build-date: 2021-01-21
build-version: fcd04ef-clean

Legal Disclaimer Notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI (TRADING) LTD ("RPTL) "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPTL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPTL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPTL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPTL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPTL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPTL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPTL's Standard Terms. RPTL's provision of the RESOURCES does not expand or otherwise modify RPTL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of Contents

Chapter 1. About the RP2040

RP2040 is a low-cost, high-performance microcontroller device with flexible digital interfaces. Key features:

- Dual Cortex M0+ processors, up to 133 MHz
- 264 kB of embedded SRAM in 6 banks
- 30 multifunction GPIO
- 6 dedicated IO for SPI Flash (supporting XIP)
- Dedicated hardware for commonly used peripherals
- Programmable IO for extended peripheral support
- 4 channel ADC with internal temperature sensor, 0.5 MSa/s, 12-bit conversion
- USB 1.1 Host/Device

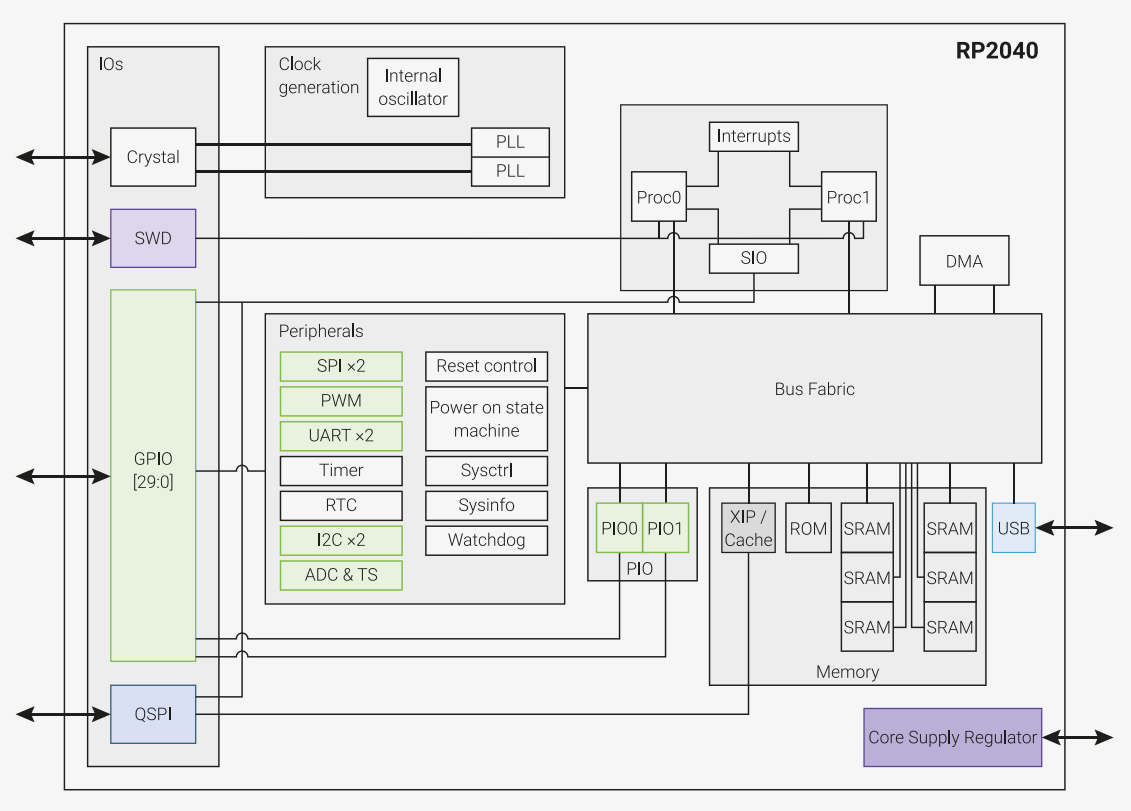


Figure 1. A system overview of the RP2040 chip

Code may be executed directly from external memory, through a dedicated SPI, DSPI or QSPI interface. A small cache improves performance for typical applications.

Debug is available via the SWD interface.

Internal SRAM is arranged in banks which can contain code or data and is accessed via dedicated AHB bus fabric connections, allowing bus masters to access separate bus slaves without being stalled.

DMA bus masters are available to offload repetitive data transfer tasks from the processors.

GPIO pins can be driven directly, or from a variety of dedicated logic functions.

Dedicated peripheral IP provides fixed functions such as SPI, I2C, UART.

Flexible configurable PIO controllers can be used to provide a wide variety of IO functions.

A simple USB controller with embedded PHY can be used to provide FS/LS Host or Device connectivity under software

control.

4 GPIOs also share package pins with ADC inputs.

2 PLLs are available to provide a USB or ADC fixed 48MHz clock, and a flexible system clock up to 133MHz

An internal Voltage Regulator will supply the core voltage so the end product only needs supply the IO voltage.

Chapter 2. Minimal Design Example

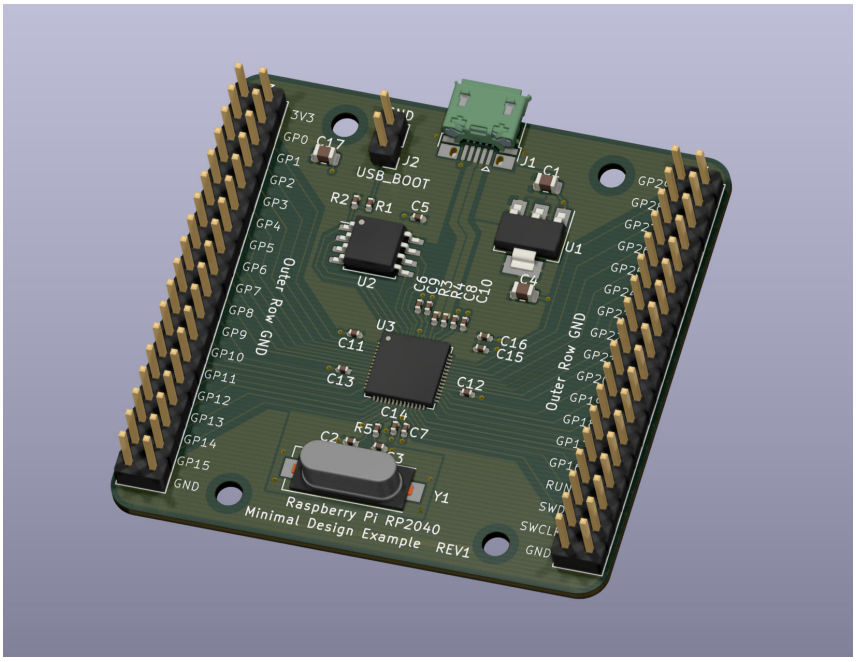


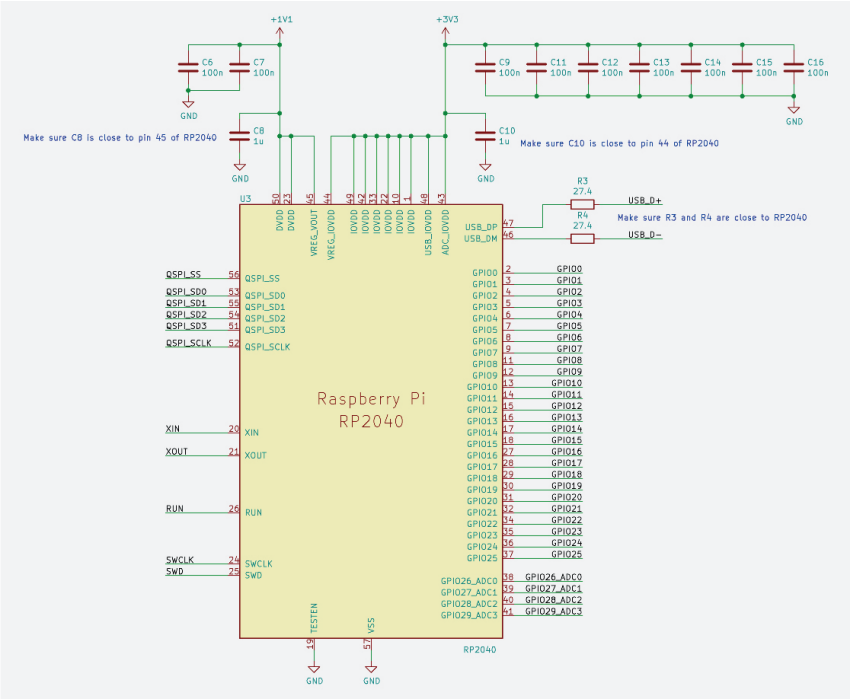
Figure 2. KiCad 3D rendering of the Minimal Design Example

This minimal design example is intended to demonstrate how you can get started with your own RP2040 based PCB designs. It consists of very nearly the minimum amount of circuitry required to make a functional design that can run your code. Schematics and layout files are available for KiCad at https://datasheets.raspberrypi.org/rp2040/Minimal-KiCAD.zip. KiCad is a free, open source suite of tools for designing PCBs and can be found at https://kicad.org/.

This example PCB has two copper layers, and has components on the top side only (this makes it cheaper and easier to assemble). It also uses small SMD (surface mount devices) components. The relatively large minimum track width, clearances and hole sizes should make this design easily and cheaply manufacturable from a range of PCB suppliers. The board is nominally 1mm thick, but it could be manufactured with a thicker PCB, for example 1.6mm is very common, but you might run into difficulties with the USB characteristic impedance (discussed below).

Whilst it might be seen as beneficial to use large, easily hand-solderable components for such an example design, the reality is that RP2040 is a 56 pin, 7x7mm QFN (Quad Flat No-leads) package with a small pitch (0.4mm pin-to-pin spacing). This requires a considerable amount of skill and experience to hand solder successfully. We therefore consider it best to have the PCBs machine assembled, however, if you are able to wield a soldering iron deftly enough to solder a QFN package successfully, then the use of other small SMD components (such as 0402 capacitors) should present few problems.

Figure 3. Schematic section RP2040 connections



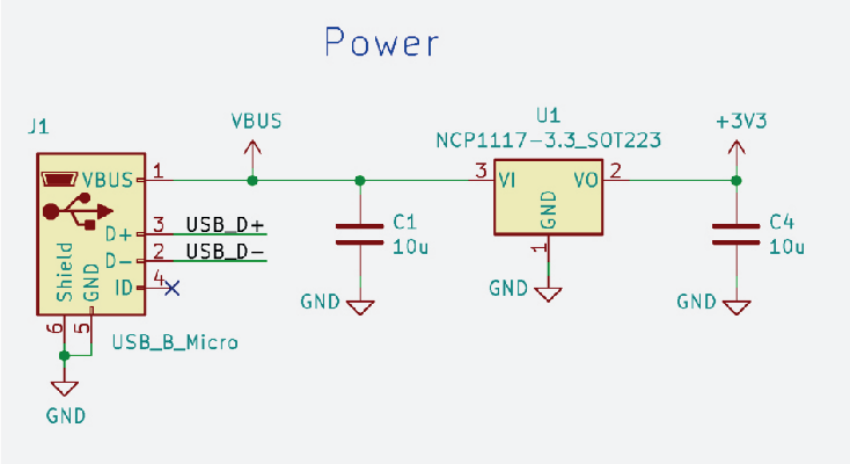
This design consists of four main elements: **power, flash storage, crystal oscillator, and IOs (Input/Outputs)**, and we'll consider each in turn below.

2.1. Power

At its simplest, RP2040 requires two different voltage supplies, **3.3V** (for the IO) and **1.1V** (for the chip's digital core). Fortunately, there is an *internal Low Dropout Voltage Regulator* (LDO) built into the device, which converts 3.3V to 1.1V for us, so we don't have to worry too much about the 1.1V supply.

2.1.1. Input Supply

Figure 4. Schematic section showing the Power input



The input power connection for this design is via the 5V VBUS pin of a Micro-USB connector (labelled **J1** in Figure 4). This is a common method of powering electronic devices, and it makes sense here, as RP2040 has USB functionality, which we will be wiring to the data pins of this connector. As we need only 3.3V for this design, we need to lower the incoming 5V USB supply, in this case, using a second, *external* LDO voltage regulator. The *NCP1117* (**U1**) chosen here has a fixed output of 3.3V, is widely available, and can provide up to 1A of current, which will be plenty for most designs. A look at the datasheet for the NCP1117 tells us that this device requires a **10µF** capacitor on the input, and another on the output (**C1**

and **C4**).

2.1.2. Decoupling Capacitors

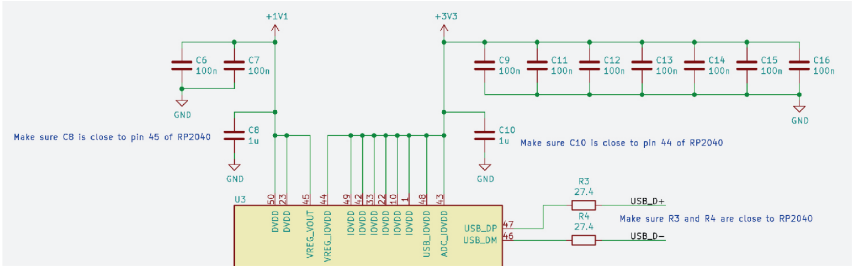


Figure 5. Schematic section showing the RP2040 power supply inputs, Voltage Regulator and decoupling capacitors

Another aspect of the power supply design are the decoupling capacitors required for RP2040. These provide two basic functions. Firstly, they filter out power supply noise, and secondly, provide a local supply of charge that the circuits inside RP2040 can use at short notice. This prevents the voltage level in the immediate vicinity from dropping too much when the current demand suddenly increases. Because, of this, it is **important to place decoupling close to the power pins**. Ordinarily, we recommend the use of a **100nF capacitor per power pin**, however, we deviate from this rule in a couple of instances.

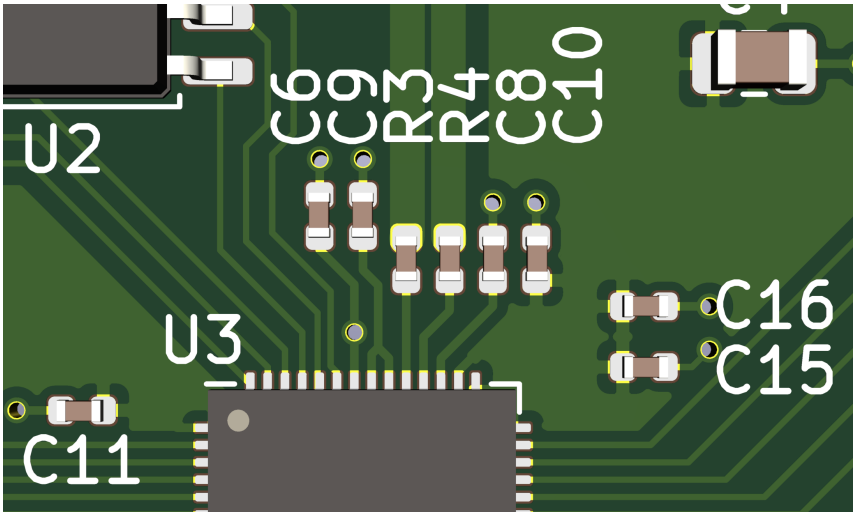


Figure 6. Section of layout showing RP2040 routing and decoupling

Firstly, in order to be able to have enough space for all of the chip pins to be able to be routed out, away from the device, we have to compromise with the amount of decoupling capacitors we can use. In this design, pins 48 and 49 of RP2040 share a single capacitor (**C9** in Figure 6 and Figure 5), as there is not a lot of room on that side of the device. This could be overcome if we used more complex/expensive technology, such as smaller components, or a four layer PCB with components on both the top and bottom sides. *This is a design trade-off*; we have decreased the complexity and cost, at the expense of having less decoupling capacitance, and capacitors which are slightly further away from the chip than is optimal (this increases the inductance). This could have the effect of limiting the maximum speed the design could operate at, as the voltage supply could get too noisy and drop below the minimum allowed voltage; but for most applications, this trade-off should be acceptable.

Secondly, the internal Voltage Regulator has its own special requirements, as you can see below.

2.1.3. Internal Voltage Regulator

The internal Voltage Regulator produces a 1.1V supply from an input of 3.3V. We simply connect the VREG_OUT pin to the DVDD pins. The regulator does have some special requirements when it comes to decoupling capacitors. **We must place 1µF capacitors close to both the input (VREG_IN) and the output (VREG_OUT)**, in order to provide a stable 1.1V supply. The Voltage Regulator also has restrictions on the amount of ESR (Equivalent Series Resistance) of these capacitors, but in practice, by using physically small ceramic chip capacitors, these requirements will almost certainly be met. In this

design, capacitors **C8** and **C10** (Figure 5) are ceramic capacitors of 0402 size.

For more details on the on-chip voltage regulator see On-Chip Voltage Regulator

2.2. Flash storage

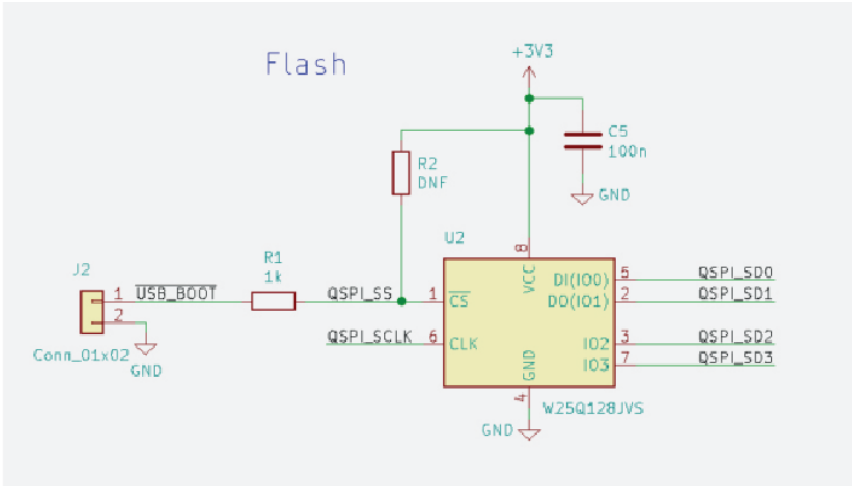


Figure 7. Schematic section showing the flash memory and USB_BOOT circuitry

In order to be able to store program code which RP2040 can boot and run from, we need to use a flash memory, specifically, a quad SPI flash memory. The device chosen here is an *W25Q128JVS* device (**U2** in the Figure 7), which is a 128Mbit chip (16Mbyte). This is the largest memory size that RP2040 can support. If your particular application doesn't need as much storage, then a smaller, cheaper memory could be used instead.

For more details on selecting a flash device see SSI

As this databus can be quite high frequency and is regularly in use, the **QSPI pins of RP2040 should be wired directly to the flash, using short connections** to maintain the signal integrity, and to also reduce crosstalk in surrounding circuits. Crosstalk is where signals on one circuit net can induce unwanted voltages on a neighbouring circuit, potentially causing errors to occur.

The **QSPI_SS signal is a special case**. It is connected to the flash directly, but it also has two resistors connected to it. The first (**R2**) is a pull-up to the 3.3V supply. The flash memory requires the chip-select input to be at the same voltage as its own 3.3V supply pin as the device is powered up, otherwise, it does not function correctly. When the RP2040 is powered up, its QSPI_SS pin will automatically default to a pull-up, but there is a short period of time during switch-on where the state of the QSPI_SS pin cannot be guaranteed. The addition of a pull-up resistor ensures that this requirement will always be satisfied. **R2** is marked as *DNF* (Do Not Fit) on the schematic, as we have found that with this particular flash device, the external pull-up is unnecessary. However, if a different flash is used, it may become important to be able to insert a **10kΩ** resistor here, so it has been included just in case. The second resistor (**R1**) is a **1kΩ** resistor, connected to a header (**J2**) labelled *'USB_BOOT'*. This is because the QSPI_SS pin is used as a *'boot strap'*; RP2040 checks the value of this IO during the boot sequence, and if it is found to be a logic 0, then RP2040 reverts to the BOOTSEL mode, where RP2040 presents itself as a USB mass storage device, and code can be copied directly to it. If we simply *place a jumper wire between the pins of J2*, we pull QSPI_SS pin to ground, and if the device is then subsequently reset (e.g. by toggling the RUN pin), RP2040 will restart in BOOTSEL mode instead of attempting to run the contents of the flash.

Both **R1** and **R2** should be placed close to the flash chip, so we avoid additional lengths of copper tracks which could affect the signal.

2.3. Crystal oscillator

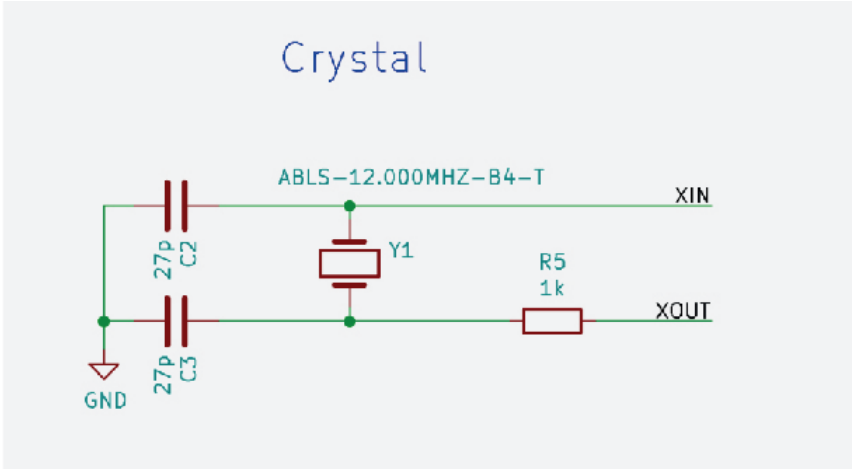


Figure 8. Schematic section showing the crystal oscillator and load capacitors

Strictly speaking, RP2040 does not actually require an external clock source, as it has its own internal oscillator. However, as the frequency of this internal oscillator is not well defined or controlled, varying from chip to chip, as well as with different supply voltages and temperatures, it is recommended to use a stable external frequency source. Applications which rely on exact frequencies are not possible without an external frequency source, USB being a prime example.

Providing an external frequency source can be done in one of two ways: either by providing a **clock source with a CMOS output** (3.3V square wave) into the XIN pin, or by using a **12MHz crystal** connected between XIN and XOUT. Using a crystal is the preferred option here, as they are both relatively cheap and very accurate.

The chosen crystal for this design is an *ABLS-12.000MHZ-B4-T* (**Y1** in Figure 8). This is a commonly available 12MHz crystal with a 30ppm tolerance, which should be good enough for most applications. This device also has a maximum ESR of **50Ω**, and a load capacitance of **18pF**, both of which have a bearing on the choice of accompanying components.

For a crystal to oscillate at the desired frequency, the manufacturer specifies the *load capacitance* that it needs for it to do so, and in this case, it is **18pF**. This load capacitance is achieved by placing two capacitors of equal value, one on each side of the crystal to ground (**C2** and **C3**). From the crystal's point of view, these capacitors are connected in *series* between its two terminals. Basic circuit theory tells us that they combine to give a capacitance of **(C2*C3)/(C2+C3)**, and as **C2=C3**, then it is simply **C2/2**. In this example, we've used **27pF** capacitors, so the series combination is **13.5pF**. In addition to this intentional load capacitance, we must also add a value for the unintentional extra capacitance, or parasitic capacitance, that we get from the PCB tracks and the XIN and XOUT pins of RP2040. We'll assume a value of **5pF** for this, and as this capacitance is in *parallel* to **C2** and **C3**, we simple add this to give us a *total load capacitance* of **18.5pF**, which is close enough to the target of **18pF**.

The second consideration is the *maximum ESR* (Equivalent Series Resistance) of the crystal. We've opted for a device with a maximum of **50Ω**, as we've found that this, along with a **1kΩ** series resistor (**R5**), is a good value to prevent the crystal being over-driven and being damaged. *You may have to adjust these values if you choose to use a different crystal*.

2.4. IOs

2.4.1. USB

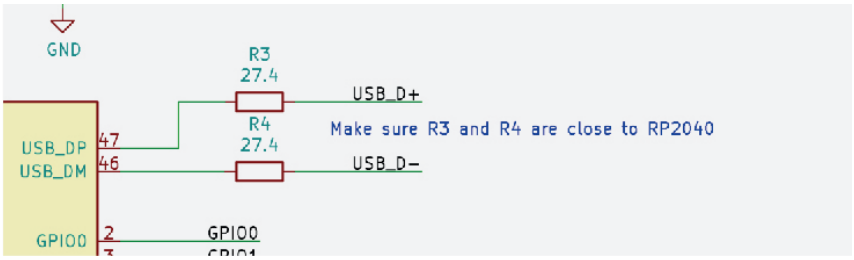


Figure 9. Schematic section showing the USB pins of RP2040 and series termination

The RP2040 provides two pins to be used for *Full Speed* (FS) or *Low Speed* (LS) USB, either as a *Host or Device*, depending

on the software used. As we've already discussed, RP2040 can also boot as a USB mass storage device, so wiring up these pins to the USB connector (**J1** in Figure 4) makes sense. The USB_DP and USB_DM pins on RP2040 do not require any additional pull-ups or pull-downs (required to indicate speed, FS or LS, or whether it is a Host or Device), as these are built in to the IOs. However, these IOs do **require 27Ω series termination resistors (R3 and R4 in Figure 9), placed close to the chip**, in order to meet the USB impedance specification.

Even though RP2040 is limited to Full Speed data rate (12Mbits-per-second), we should try and makes sure that the *characteristic impedance* of the transmission lines (the copper tracks connecting the chip to the connector) are close to the USB specification of **90Ω** (measured differentially). On a **1mm** thick board such as this, if we use **0.8mm** wide tracks on USB_DP and USB_DM, with a gap of **0.15mm** between them, we should get a differential characteristic impedance of around **90Ω**. This is to ensure that the signals can travel along these transmission lines as cleanly as possible, minimising voltage reflections which can reduce the integrity of the signal. In order for these transmission lines to work properly, we *need to make sure that directly below these lines is a ground*. A solid, uninterrupted area of ground copper, stretching the entire length of the track. On this design, almost the entirety of the bottom copper layer is devoted to ground, and particular care was taken to ensure that the USB tracks pass over nothing but ground. If a PCB *thicker than 1mm* is chosen for your build, then we have two options. We could re-engineer the USB transmission lines to compensate for the greater distance between the track and ground underneath (which could be a physical impossibility), or we could ignore it, and hope for the best. USB FS can be quite forgiving, but your mileage may vary. It is likely to work in many applications, but it's probably not going to be compliant to the USB standard.

2.4.2. IO headers

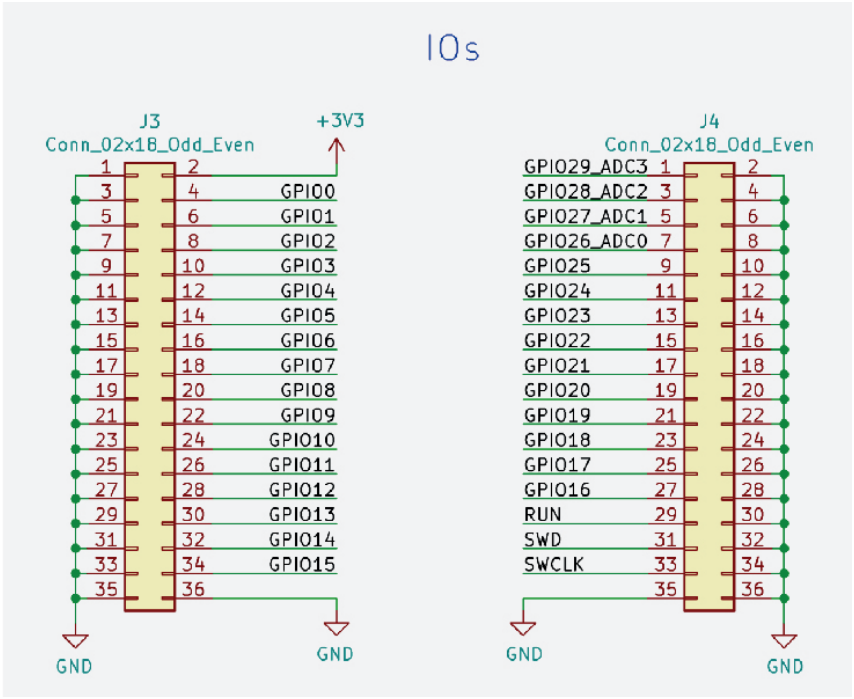


Figure 10. Schematic section showing the 2.54mm IO headers

In addition to the USB connector already mentioned, there are a pair of *2x18-way 2.54mm headers* (**J3** and **J4** in Figure 10), one on each side of the board, to which the rest of the IO have been connected. As this is a general purpose design, with no particular application in mind, the IO have been made available to be connected as the user wishes. The inner row of pins on each header are the IOs, and the outer row are all connected to ground. It is good practice to include many grounds on IO connectors. This helps to maintain a low impedance ground, and also to provide plenty of potential return paths for currents travelling to and from the IO connections. This is important to minimise electro-magnetic interference which can be caused by the return currents of quickly switching signals taking long, looping paths to complete the circuit.

Both headers are on the same 2.54mm grid, which makes connecting this board to other things, such as breadboards, easier. You might want to consider fitting only a single row 18-way header instead of the 2x18-way, dispensing with the outer row of ground connections, to make it more convenient to fit to a breadboard.

2.5. Schematic

The complete schematic is shown below. As previously mentioned, the design files are available in KiCad format.

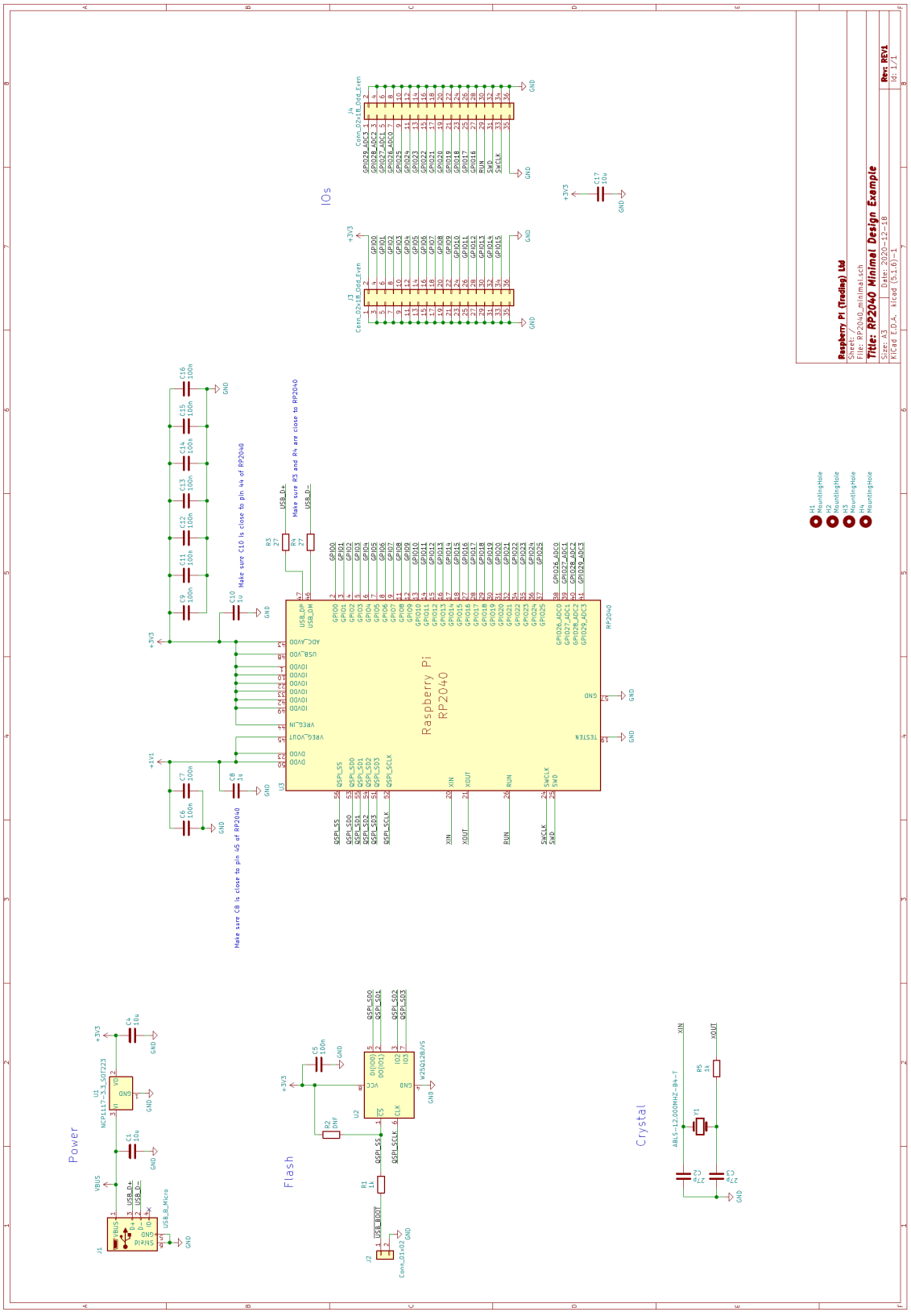


Figure 11. Complete schematic of the Minimal board

2.6. Supported flash chips

The initial flash probe sequence, used by the bootrom to extract the second stage from flash, uses an `03h` serial read command, with 24-bit addressing, and a serial clock of approximately 1 MHz. It repeatedly cycles through the four combinations of clock polarity and clock phase, looking for a valid second stage CRC32 checksum.

As the second stage is then free to configure execute-in-place using the same `03h` serial read command, RP2040 can perform cached flash execute-in-place with **any** chip supporting `03h` serial read with 24-bit addressing, which includes most 25-series flash devices. The Pico SDK provides an example second stage for CPOL=0 CPHA=0, at https://github.com/raspberrypi/pico-sdk/tree/master/src/rp2_common/boot_stage2/boot2_generic_03h.S. To support flash programming using the routines in the bootrom, the device must also respond to the following commands:

- `02h` 256-byte page program
- `05h` status register read
- `06h` set write enable latch
- `20h` 4kB sector erase

RP2040 also supports a wide variety of dual-SPI and QSPI access modes. For example, https://github.com/raspberrypi/pico-sdk/tree/master/src/rp2_common/boot_stage2/boot2_w25q080.S configures a Winbond W25Q-series device for quad-IO continuous read mode, where RP2040 sends quad-IO addresses (without a command prefix) and the flash responds with quad-IO data.

Some caution is needed with flash XIP modes where the flash device stops responding to standard serial commands, like the Winbond continuous read mode mentioned above. This can cause issues when RP2040 is reset, but the flash device is not power-cycled, because the flash will then not respond to the bootrom's flash probe sequence. Before issuing the `03h` serial read, the bootrom always issues the following fixed sequence, which is a best-effort sequence for discontinuing XIP on a range of flash devices:

- `CSn=1`, `IO[3:0]=4'b0000` (via pull downs to avoid contention), issue x32 clocks
- `CSn=0`, `IO[3:0]=4'b1111` (via pull ups to avoid contention), issue x32 clocks
- `CSn=1`
- `CSn=0`, `MOSI=1'b1` (driven low-Z, all other IOs Hi-Z), issue x16 clocks

If your chosen device does not respond to this sequence when in its continuous read mode, then it must be kept in a state where each transfer is prefixed by a serial command, otherwise RP2040 will not be able to recover following an internal reset.

2.7. Making a PCB

The minimal design example, see Chapter 2, was deliberately designed with two copper layers, and with components on the top side only. The design rules are relaxed, to allow low cost PCB fabrication. This particular design has been verified to work with Eurocircuits (https://www.eurocircuits.com/) standard PCB pool, though there should be few problems having it manufactured by other PCB prototyping manufacturers.

Chapter 3. The VGA, SD Card & Audio Demo board for Raspberry Pi Pico

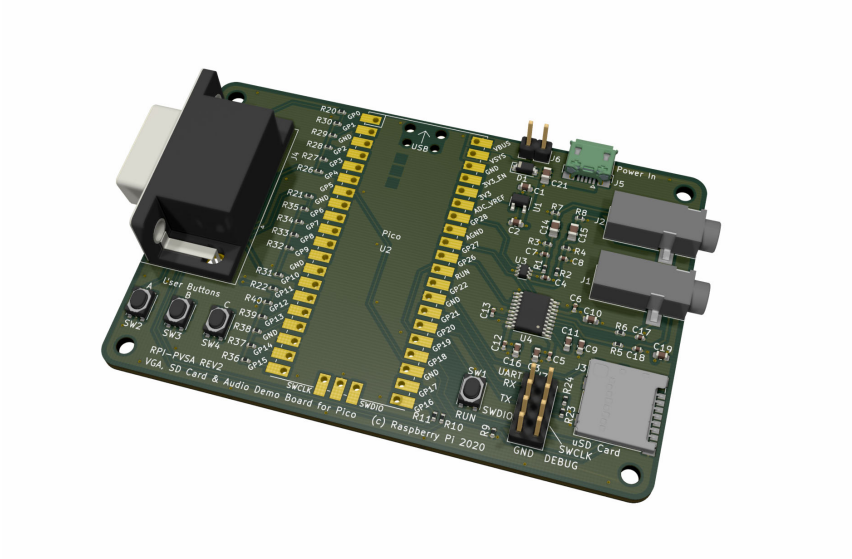


Figure 12. KiCad 3D rendering of the VGA SD Card & Audio Design Example for Raspberry Pi Pico

This example design is intended to serve two distinct purposes. Firstly, we show how we can design a PCB that incorporates *Raspberry Pi Pico as a module*, used simply as a component on a larger design. Secondly, some of the more complex RP2040 applications require specific additional hardware in order to function correctly. This design provides some example designs for four of these applications, **VGA** video, **SD Card** storage, and two flavours of audio output; **analogue PWM**, and **digital I2S**. Experimental software using these features can be found at https://github.com/raspberrypi/pico-playground.

This design is based around using Raspberry Pi Pico, but as Raspberry Pi Pico provides direct access to the pins of RP2040, *much of the circuitry shown here would be equally applicable to designs based around RP2040 itself.*

One of the key differences between designing with the Raspberry Pi Pico and RP2040 is that not all of the IOs of RP2040 are available to be used on Raspberry Pi Pico. This is because a couple of the IOs are used for internal house-keeping (such as power supply control and monitoring, and an LED), and are not exposed to the outside world. This brings some challenges with it, particularly as our choice of application examples want more pins than are available on Raspberry Pi Pico. We think we've thought of some cunning solutions to this, especially when you consider we've also added three user buttons and a UART connection to the mix. We'll go through these constraints, and their solutions, in detail later.

Schematic, PCB layout and Raspberry Pi Pico footprint files are provided in **KiCad** format, with similar design rules as the previous Minimal Design Example in Chapter 2. This time around, whereas the Minimal Design Example has two layers, with a 1mm thick PCB, we've opted for a four layer, 1.6mm thick PCB. This is primarily because adding extra layers to our PCB means that we can now devote entire layers to power and ground. This is important in a number of ways. Firstly, it improves power supply decoupling. With the additional of these two layers, we now have two large, parallel rectangles of copper; one connected to the power supply, the other to ground. These are then separated by a thin dielectric material (an insulating PCB layer sandwiched between the copper layers), which makes this a simple parallel plate decoupling capacitor. Secondly, and perhaps most importantly for this design, we now have a variety of low impedance paths back to RP2040 where the quickly changing IO return currents can flow back quickly and unhindered, without creating current loops which can cause Electro-Magnetic emissions. Another benefit of moving to four layers is that as there is now less of a gap between signal tracks on the top layer, and the ground plane directly beneath, it is now much easier to create tracks of different characteristic impedances that may be required in your designs. In this case, we will want 75Ω tracks for the VGA colour signals as VGA is a 75Ω system, using 75Ω cables and 75Ω load termination in the monitor.

This design can be sub-divided into five sections: **Power, VGA, SD Card, Audio, and Raspberry Pi Pico itself.**

3.1. Power

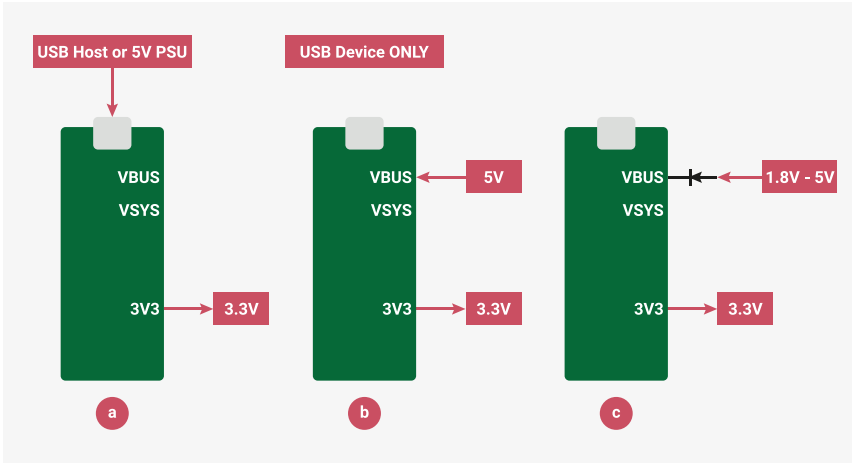


Figure 13. Recommended ways of powering Raspberry Pi Pico

There are three main ways we can safely power Raspberry Pi Pico, and the choice is entirely dependent on your application. We can either power Raspberry Pi Pico from the **micro USB connector** on the device itself (option (a) in Figure 13), or we can provide power to either the **VBUS pin** (option b), or the **VSYS pin** (option c).

NOTE

The **3V3** pin is an output from Raspberry Pi Pico and should not be connected to an external power source. It is intended to be used as an output to provide power to external circuits.

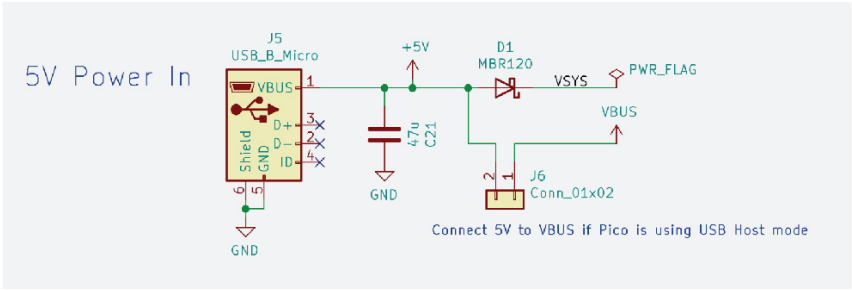


Figure 14. Section of schematic showing power input

The **VSYS pin** is the main system power supply on Raspberry Pi Pico. From this supply, Raspberry Pi Pico generates its own 3.3V supply which is used to power RP2040, and also the 3V3 output pin of Raspberry Pi Pico which we can use to power circuits on our design.

The **VBUS pin** is connected to the VBUS of the micro USB connector of Raspberry Pi Pico. There is an onboard diode connecting VBUS to VSYS, which means that VBUS can be used to power VSYS, but not the other way around.

This design provides different options for providing the power, and the *choice of which one to use depends very much on your application*. The first thing to consider is if the USB functionality of Raspberry Pi Pico will be used.

Not using USB

If we are not using USB, then we must provide power for Raspberry Pi Pico. One way of doing this is to *provide power to Raspberry Pi Pico from our board*, through Raspberry Pi Pico's pins, see Figure 15 for an illustration of this. The preferred way of implementing this is to provide a voltage to the **VSYS pin** via a Schottky diode (Figure 14). The one-way nature of the diode ensures we don't get any problems if we also supply power to the VBUS pin of Raspberry Pi Pico (accidentally or deliberately). Raspberry Pi Pico can take a voltage of between 1.8 and 5.5V, as it has an internal buck-boost regulator (one which can regulate the output to a higher or lower voltage than its input), but due to the fact we have an additional voltage regulator in our design (**U1**, more on this later), we need to make sure that VSYS is greater than 3.5V so that **U1** will operate correctly.

Alternatively, we could *provide power to the VBUS pin* of Raspberry Pi Pico (not to be confused with the VBUS connection on Raspberry Pi Pico's USB connector), rather than the VSYS pin. This would internally power VSYS via Raspberry Pi Pico's

onboard diode, but we must be sure that we *do not connect another power supply to Raspberry Pi Pico's USB connector*.

On this design we use a micro USB connector (**J5** in Figure 14) to provide a 5V power input. This is then connected to VSYS via **D1**, which is an MBR120 Schottky diode that can carry up to 1A. There is also an optional jumper (**J6**) we could use if we need to power the VBUS pin, but as we not using USB, this is unnecessary.

As a third alternative, we could attach a 5V supply to Raspberry Pi Pico's USB connector, rather than our board's USB connector, similar to 'Device Mode' discussed below and in Figure 16.

Using USB

If we are going to be using USB, then we need to know whether it will be in **Host** or **Device** mode.

Device Mode

If we are using Raspberry Pi Pico in Device mode, then the host it is attached to will provide 5V on the *VBUS pin of the USB connector*, which in turn will internally provide VSYS with power (5V minus the drop across the onboard diode). This is everything we need, voltage-wise, we do not need to do anything extra on our design, but this power is only available when the USB host is attached. See Figure 16. If we need to be self-powering, i.e., not reliant on the incoming 5V from the USB host, then we need to provide our own power from the carrier board. Again, we can connect a 5V supply to the micro USB connector **J5**, so that we provide around 5V to the **VSYS pin** of Raspberry Pi Pico. *Make sure jumper **J6** is open circuit*, as this could result in directly connecting two 5V supplies together. See Figure 15 for an illustration.

Host mode

If we are using USB Host mode, then this time, *Raspberry Pi Pico needs to provide 5V to the VBUS pin of **its** micro USB connector* (not J5). This means that our carrier board design must supply the **VBUS pin** of Raspberry Pi Pico with 5V, as well as powering Raspberry Pi Pico. We can do this on our design by simply connecting the micro USB connector (**J5** on the schematic) to a 5V supply, and also by fitting a jumper on **J6**, so that this 5V supply gets connected directly to the VBUS pin of Raspberry Pi Pico. VSYS is supplied by a combination of the onboard diode on Raspberry Pi Pico, as well as diode D1 on our design, which is perfectly safe.

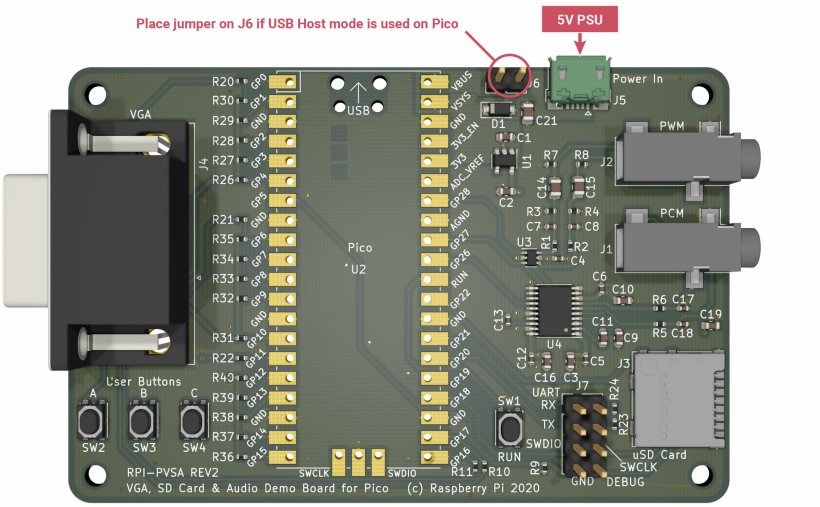


Figure 15. Powering the system using the USB power connector on the VGA, SD Card & Audio board

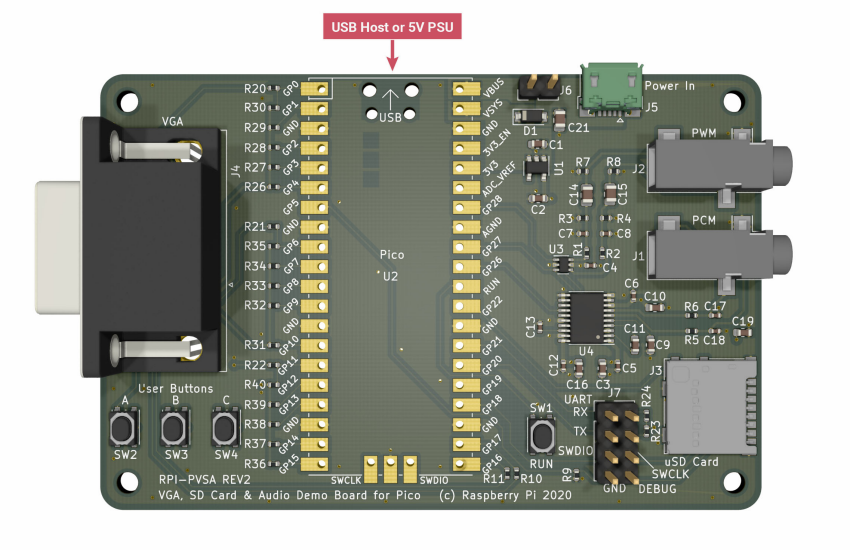


Figure 16. Powering the system using the USB connector on Raspberry Pi Pico

3.1.1. Audio Power supply

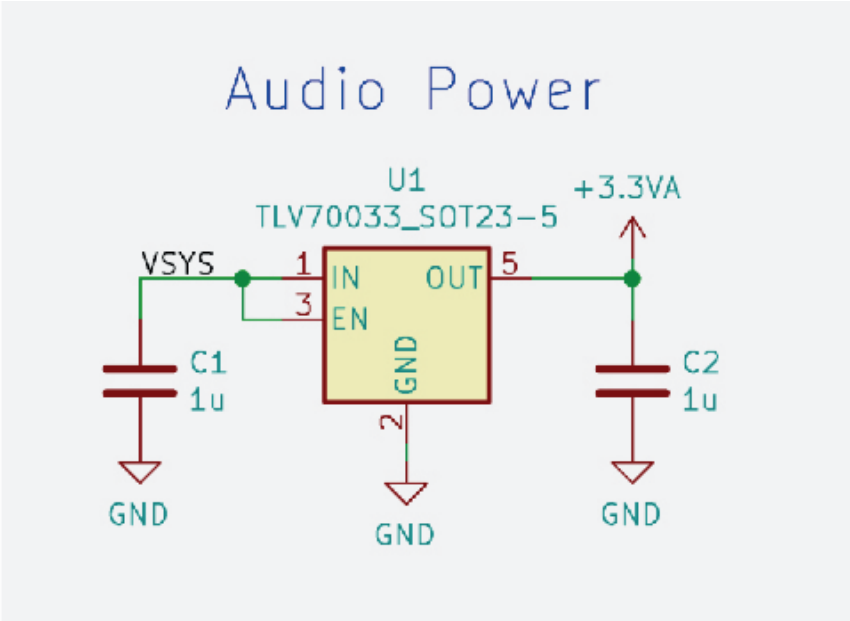


Figure 17. Schematic section showing an additional LDO used for powering the audio

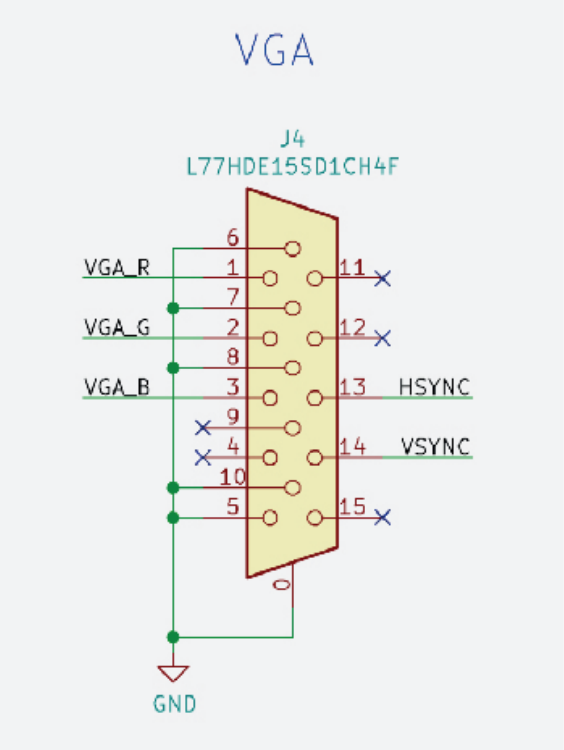
In addition to providing power for Raspberry Pi Pico, we have some circuits on this design which need suitable power supplies. Fortunately, they are all 3.3V, so we can simply use the 3V3 supply from Raspberry Pi Pico itself. However, as we have some audio circuitry on this design, it's good to have a nice, clean power source, without all the digital switching noise, for the sensitive audio output sections. To this end, we've included a 3.3V linear voltage regulator (**U1** in Figure 17), specifically for the audio output, which is supplied by VSYS (which is always present, unlike VBUS). This device is a TLV70033, which is a Low Drop-Out (LDO) regulator, with a fixed 3.3V output. This can supply 200mA, which is more than enough for the audio circuits used here. The datasheet for the TLV70033 tells us that we need 1µF capacitors on the input and output pins. We've used 0603 sized ones here (**C1** and **C2**).

NOTE

The switching regulator used on Raspberry Pi Pico has two operating modes, depending on the amount of current running through it. In low current mode (less than a few tens of mA), in order to increase its efficiency, it starts to run in 'Power Saving Mode' which uses PFM (Pulse Frequency Modulation). Ordinarily, this is a good thing, as it increases efficiency, reducing the power Raspberry Pi Pico consumes at low loads. However, this comes at a price, and that is a little more voltage ripple on the 3V3 supply. Most of the time this isn't a problem, but for noise sensitive circuits, you might want to switch this power saving feature off, and return to the less efficient, but less noisy PWM (Pulse Width Modulation) mode. Raspberry Pi Pico allows us to do this by forcing the regulator to always use PWM mode, and we do this by setting GPIO23 on RP2040 high. In the VGA demo below, the effects of this noise can be seen if we look carefully at the VGA monitor; we can see slight variations of colour in the horizontal lines, as this supply noise is transferred directly to the DAC outputs. If we disable the PFM mode of the regulator, this magically goes away.

3.2. VGA Video

Figure 18. Schematic section showing the VGA video connector



The first application of RP2040 we're demonstrating is VGA analogue video output. This particular example uses the PIO (Programmable IO) on RP2040 to output a commonly used 16-bit RGB data format (RGB-565) and these digital outputs then need to be converted to three analogue output signals; one for each colour. RGB-565 uses 5-bits each for the Red and Blue channels, and 6-bits for the Green. In addition to these 16 data bits, VGA monitors also require HSYNC and VSYNC signals for horizontal and vertical blanking timing. That brings us to a total of 18 pins that are needed. As we've mentioned before, pins are at a premium, and we want to use as few as possible so that we can cram more functionality into this design. To this end, we can free up a pin by limiting the green channel to 5-bits, making all the channels the same resolution, by removing the green LSB (Lowest Significant Bit). It is still desirable for RP2040 to process RGB-565 format data, so PIO will still output 6-bits of green data to the GPIOs, but we can choose not to use the green LSB in the function select register of that particular GPIO, instead letting RP2040 use it for something else (in this case, the clock for the SD Card). Another design constraint we have is that the VGA PIO software requires that all 16-bits of data need to be on contiguous (in unbroken, consecutive numerical order) GPIOs, with the sequence being Red first, then Green, then Blue, with the LSB first in each case. Raspberry Pi Pico has two contiguous bunches of GPIOs available for our use. GPIOs 0 to 22, and 26 to 28. We therefore must place VGA data somewhere in 0 to 22, and it makes sense to start at one end or the other in order to make sure there are as many contiguous pins remaining for other functions as possible. We've chosen to

use GPIO 0 to 15, which means that the Green LSB is GPIO 5, and this is going to be used as SD_CLK. HSYNC and VSYNC can go on any GPIO, as long as they are next to each other. We've picked 16 and 17.

3.2.1. Resistor DAC

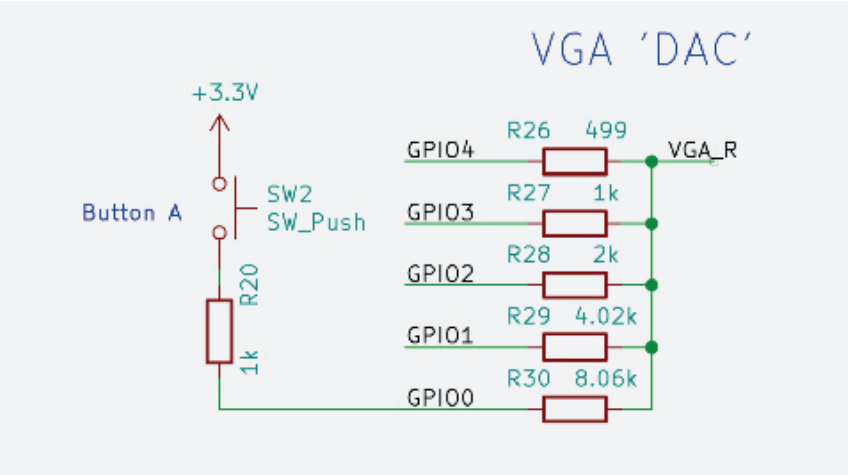


Figure 19. Schematic section showing the Red channel of the VGA resistor DAC

The three colour channels on a VGA connector need to be analogue signals, varying from 0 to 0.7V. We therefore need to convert the digital, 3.3V outputs of RP2040 to an analogue signal. You can use dedicated video DACs (Digital to Analogue Convertors) to do this, but a cheaper and simpler method is shown here. You can create a simple DAC using a bunch of resistors connected directly to the digital outputs. The values of the resistors are weighted to give different amounts of significance to each bit, in the ratio 1:2:4:8:16. It's not going to be as good as using a dedicated video DAC, one of the major drawbacks is that any voltage variation on the IOVDD supply of RP2040 is going to be present on the DAC output, but it's cheaper, considerably less complex, and a lot more fun. If we look at the Red channel, net VGA_R on the schematic, we can see the Red LSB (GPIO0) is connected to it through a 8.06kΩ resistor. The next bit (GPIO1) has (roughly) half this (4.02kΩ), the next has half again, and so on for the rest of the bits. Ideally, for the most linear DAC performance, we want exactly double the previous resistor value, but these are the nearest commonly available 1% values. This means each GPIO output bit can contribute twice as much current through its resistor than the previous bit, and all these individual current contributions are summed together at the output. The result of this is that if all the bits are high (3.3V), corresponding with the maximum digital value, we have all five resistors in parallel to 3.3V. Basic circuit theory tells us that this is 1/Rparallel = 1/499 + 1/1000 + 1/2000 + 1/4020 + 1/8060 = 0.00388, so Rparallel is 258Ω. If we have a monitor connected to this signal, then we will have a 75Ω resistor to ground inside the monitor (this isn't shown on this schematic). This creates a potential divider, with 3.3V connected to 258Ω, which in turn is then connected to 75Ω to ground in the monitor. This means we have a full-scale voltage of 3.3 * 75 / (258 + 75) = 0.74V, which is close enough to the target of 0.7V.

3.2.2. User buttons

The user buttons are not strictly part of the VGA, but because we've decided to add them (**SW2**, **SW3** and **SW4**, see Figure 19), connecting them to the LSBs of the Red, Green and Blue channels, it's important to talk about them, and on their use in the software. We thought it was important to add a few buttons to this design, especially as VGA, SD Card and audio gives us a lot of potential applications that could use a button or two (e.g. video or music controls, etc). As we've already said, pins are at a premium, and we couldn't afford to dedicate a pin or two to something as frivolous as buttons, so we've come up with the idea of making the VGA LSBs multi-purpose, with a simple hack.

The basic idea is that the GPIO in question is used for VGA as usual during the active periods of video data, but during the video blanking periods, when the DAC levels are not as critical, we can flip the GPIO direction to an input, and then poll it, before flipping it back to an output for the next active video period. If button **SW2** is pressed, then GPIO0 will be connected to potential divider of a 1kΩ resistor (**R20**) to 3.3V, and 8.06kΩ (**R30**) to (worst-case) 0V. This means that GPIO0 will see at least 2.93V, and will therefore register as logic 1. If the button is not depressed, then GPIO4 will be 0.7V or lower, which will result in a logic 0. Of course, this relies on a monitor's 75Ω load resistor for the pull-down. If there is no monitor present, then the user could activate the GPIO's internal pull-down instead.

Obviously, if the button is pressed during active video transmission, then we might expect this to have an effect on the DAC signal level. However, as we are only interfering with the LSB, any effect would be minimal, but the introduction of the 1kΩ resistor (**R20**) in series with the button means that RP2040 will have little problem over-driving this, so the effect on the DAC signal will be minimal. The final point to note regarding the DAC is that, as we've previously mentioned, the outputs should have 75Ω characteristic impedance. On this PCB, we've used a 1.6mm, four layer board stack-up, with a gap of 0.36mm between the outer and inner layers. This means that track widths of 0.3mm gives us roughly 75Ω.

3.3. SD Card

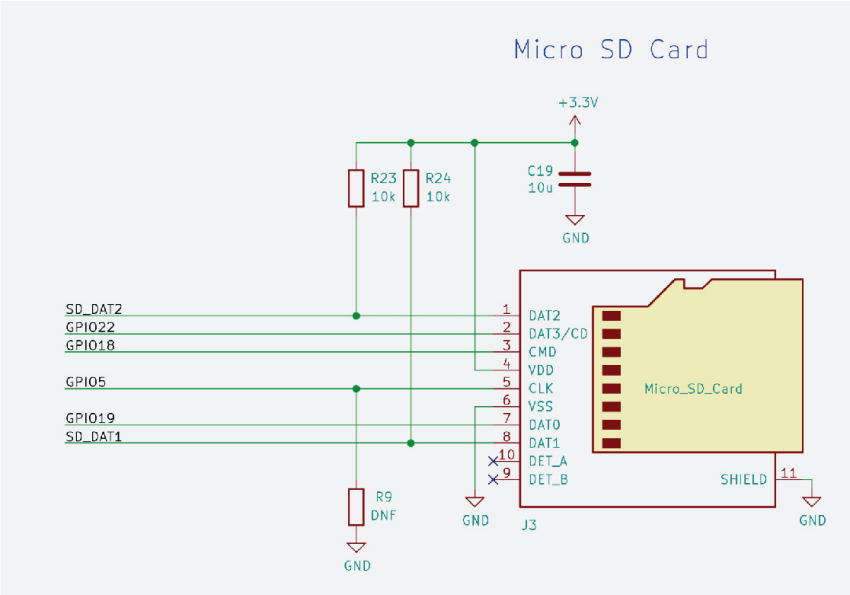
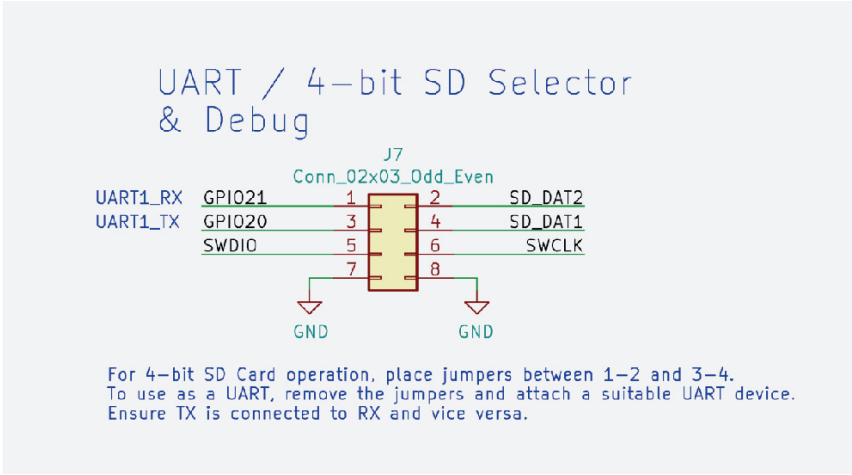


Figure 20. Schematic section showing the micro SD Card connector

The second application we are demonstrating is using an SD Card. This design has a micro SD Card (**J3**), which has a 4-bit data bus, as well as a Clock (CLK) and Command (CMD) signal. We can access the SD Card in either 4-bit mode, SPI, or 1-bit mode. The constraints our SD PIO software puts upon us is that the 4 data signals must be connected to contiguous GPIOs. The CLK and CMD signals can go anywhere. As the VGA signals have used up GPIOs 0 to 17, we are left with contiguous block of 5 GPIOs, 18 to 22, so we are going to use GPIO19 to 22 for the data bus, and GPIO18 for the CMD signal. For the CLK signal, we going to be using GPIO5, which is in the middle of the VGA signals. If you remember, GPIO5 was earmarked for the 6th, unused bit of the Green VGA output. This GPIO can be repurposed by selecting a different function on the GPIO mux, so we are free to assign it to be SD CLK. Often, SD interfaces include pull up and down resistors on the PCB. This is to ensure that safe values are present at all times, especially when the IOs are in an undefined state, but also because some of the SD signals are used as mode select pins (e.g. SPI mode, 1-bit mode, etc). In this design, we are relying on RP2040 to set the GPIO pulls. We have added the option for a pull down for the CLK signal (**R9**) should we find that in a particular application it is needed, as it is important the CLK input is in a valid state at all times. Having said all this, we have actually included pull-ups on bits 1 and 2 of the SD data bus (**R23 & R24**). This is because we haven't wired these SD card IOs directly to Raspberry Pi Pico, and have instead connected them via jumper headers (pins 1 to 2 and 3 to 4 of **J7**), which means it's possible to remove the jumpers and still have valid levels on these IOs. Obviously, if we want 4-bit operation, we must connect the jumpers first. The reason we've done this is, as has been already mentioned, the SD Card can also be accessed using SPI or 1-bit mode, which means that if either of these methods are used, we can potentially repurpose data bits 1 and 2 for other uses. Which brings us to …

3.3.1. UART

Figure 21. Schematic section showing the optional UART and SWD debug header



As alluded to above, if we use the SD Card in 1-bit mode, or even not use SD Card at all, we are then free to use these pins for a UART, which is always a useful thing to have. To this end, we can simply connect a 3.3V compatible UART to pins 1 and 3 of **J7**, rather than the jumpers needed for 4-bit SD Card operation. Nominally, GPIO21 is UART1_RX, and GPIO20 is UART1_TX if the dedicated hardware UART controllers are used, but if a PIO UART is implemented, then the TX and RX selection would be configurable.

3.3.2. Debug – SWD

J7 is also home to the SWDIO and SWCLK pins on this design. If Raspberry Pi Pico has been attached to this PCB in such as way as to connect the debug pins, then they are made available on this header to connect a debugger to. Of course, a debugger could also be connected directly to the Raspberry Pi Pico itself if this is more suitable.

3.4. Audio

This design demonstrates two different audio options that RP2040 can use, analogue PWM and digital PCM/I2S. However, as you might expect, we cannot afford to dedicate separate pins for each solution, so these two options use the same pins, and the choice of which is to be used is made in software. The circuitry for both audio options have been populated, as the option not being used shouldn't suffer any problems when driven in the wrong mode.

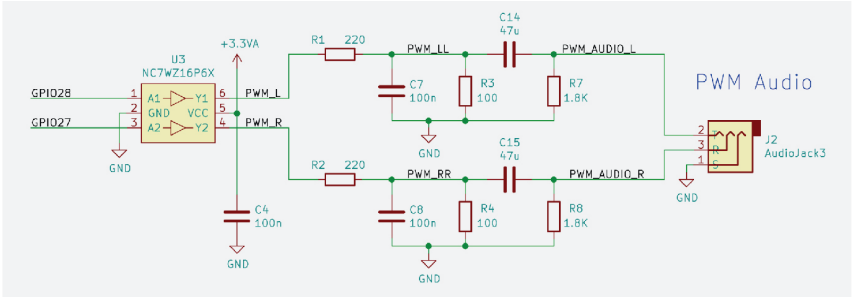
NOTE

We need to remember to connect the audio output device to the correct jack: **J1** for PCM and **J2** for PWM.

These outputs are intended to be used as a line level driver, and connected to an amplifiers line-in input, but they should also work for headphones with higher impedances. The remaining GPIOs we have available are 26 to 28, and happily, this is all that is needed; two for PWM, and three for PCM.

3.4.1. PWM Audio

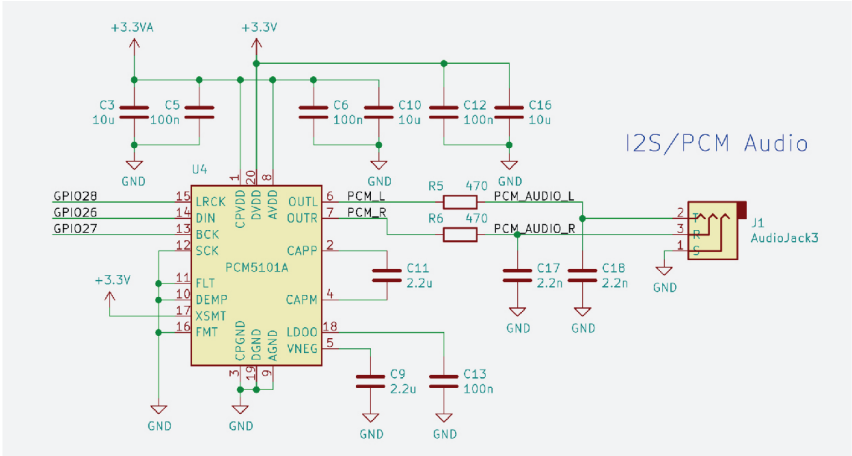
Figure 22. Schematic section showing the Analogue PWM audio circuit



The first method we are going to consider is the analogue PWM. This method is the same as is used on the Raspberry Pi 4 audio output jack, and we've borrowed the circuit from it. This works by taking the digital audio, and outputting it from two GPIO pins as digital PWM (Pulse Width Modulation) signals, one for each of the stereo pair. These digital signals are then fed into a small logic buffer (**U3**). This is so that we can use our nice, clean, audio 3.3V supply we discussed earlier, so hopefully we won't get the digital noise from the main 3.3V supply on our audio signal. This buffered output, which is still a 3.3V digital signal, is then fed into an analogue filter, and the result is that we get an a.c. coupled analogue signal in the audible frequency range, which we can then connected to an amp or headphones.

3.4.2. PCM/I2S Audio

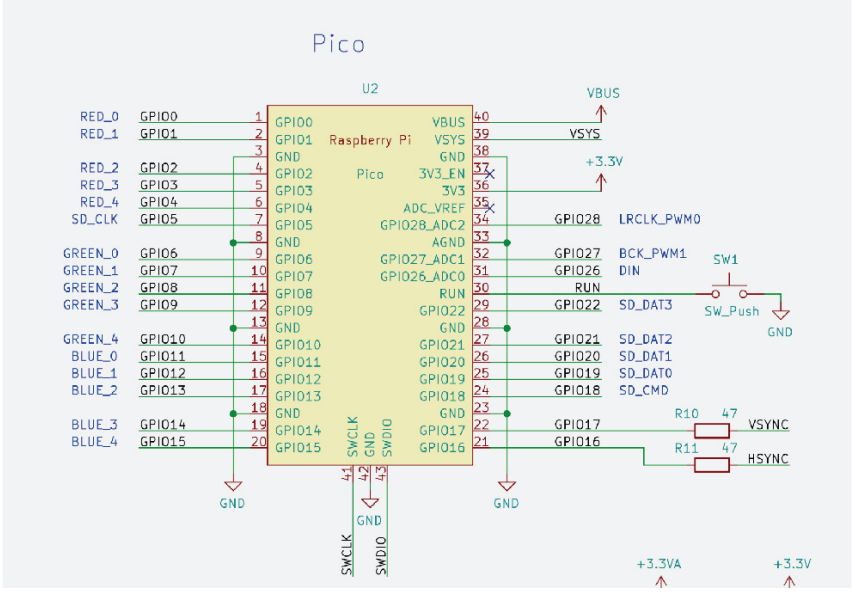
Figure 23. Schematic section showing the Digital I2S PCM audio circuit



The second audio option used here is digital PCM using I2S. This method takes digital audio in PCM (Pulse Code Modulation) format, and sends it using the I2S protocol to an audio DAC, which in turn is connected to an audio output jack. In this design, we've chosen to use the PCM5101A audio DAC. GPIO27 is connected to the BCK input (bit clock), GPIO26 to DIN (serial data), and GPIO28 to LRCK (left or right clock). The rest of the circuitry surrounding the DAC is as per the typical application circuit in the PCM5101A datasheet.

3.5. Raspberry Pi Pico

Figure 24. Schematic section showing the connections to Raspberry Pi Pico

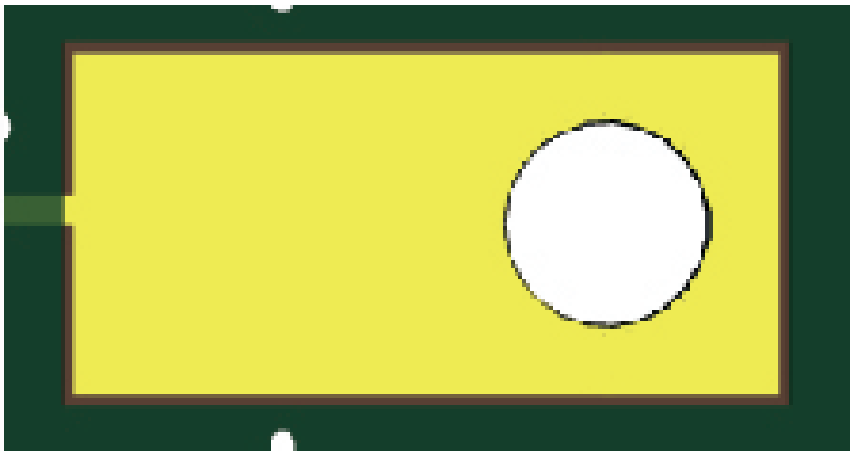


The final piece of the design is Raspberry Pi Pico itself. We have already covered the vast majority of the Raspberry Pi Pico connections in the preceding sections, however, there are a few pins we haven't covered. We've already discussed

how the power pins are to be connected (VBUS and VSYS inputs and 3V3 output), but as yet we haven't really mentioned **GND**. **All** of the GND pins should be connected to ground net on our board, and ideally to a low impedance ground plane to minimize noise and EMC emissions. There is an **AGND** pin on Raspberry Pi Pico which is intended to be used as a low noise ADC return path, but as we are not using the ADC in this application, we simply connect this to regular GND. There is also an **ADC_VREF** pin, which can be optionally used to supply a clean and stable reference as an alternative to the Raspberry Pi Pico onboard 3.3V supply, but again, as we are not using the ADC, we can safely leave this pin floating. The **RUN** pin on Raspberry Pi Pico is the reset_n (active low) for RP2040. It is pulled high (i.e. the RP2040 is running) on Raspberry Pi Pico, but with the addition of a push button (**SW1**) we can pull this pin low, causing RP2040 to reset. The final pin on Raspberry Pi Pico is **3V3_EN**, which controls the 3.3V supply on Raspberry Pi Pico. As we have no need to disable this supply on this board, we can leave this pin floating as there is a pull-up on Raspberry Pi Pico itself.

And finally, to perhaps the most important part of this design, how do we attached the Raspberry Pi Pico itself? There are two choices, and this design lets us pick either.

Figure 25. Close-up of Surface Mount and Through Hole pad



Each pin on Raspberry Pi Pico has two soldering options. You can either solder 0.1" headers using the through-holes, or alternatively, as Raspberry Pi Pico has castellated edges, where the pin extends to the edge of the board, and then down the edge of the PCB itself, it can be soldered down directly to a PCB.

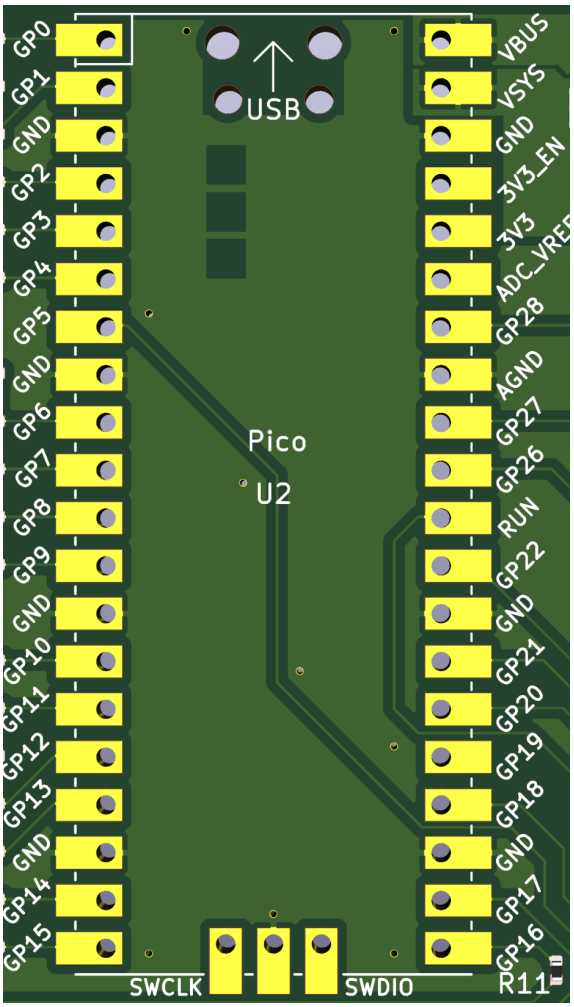


Figure 26. Surface Mount and Through Hole Footprint for attaching to Raspberry Pi Pico

The CAD footprint provided with this design provides both options, so Raspberry Pi Pico can either be soldered direct to this design, or 0.1" headers may be used (or indeed, a combination of 0.1" headers and sockets) to connect the two PCBs together.

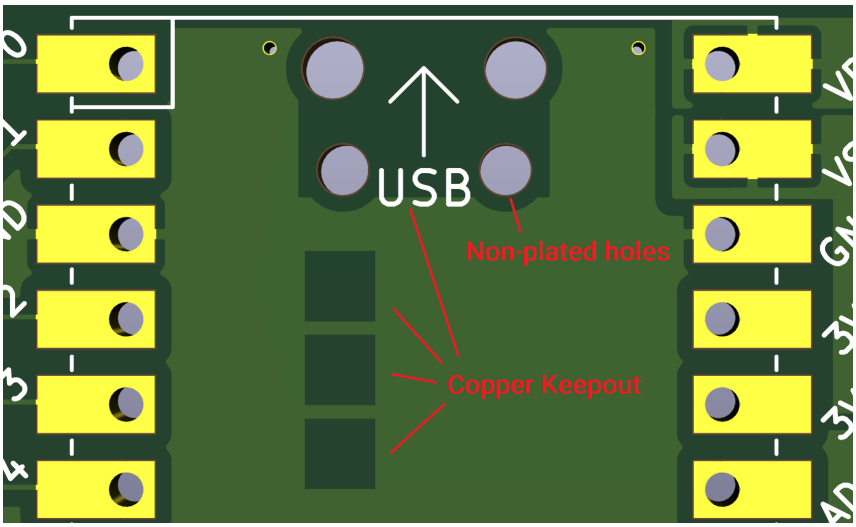


Figure 27. Holes and copper keepout areas under the USB connector and testpoints on Raspberry Pi Pico

Another feature of this footprint are the four drill holes you can see towards the top of the board, directly underneath Raspberry Pi Pico's micro USB connector (see Figure 27). These are here to help the Raspberry Pi Pico sit flat against our carrier PCB, as the metal through-hole lugs which anchor the USB connector to the Raspberry Pi Pico can sometimes protrude slightly from the board. These holes allow any excess metal, or solder, to safely poke through. As well as these holes, you can see some areas of keepout on the top copper layer. This is because the underside of Raspberry Pi Pico has

some testpoints, which are exposed areas of copper that get used during production testing, and these keepouts align with the testpoints. This is not strictly necessary, as there is still solder resist (the green insulating material on the surfaces of the PCB) on our PCB, but we consider it good practice to do so as it makes the chances of shorting these testpoints through accidental damage, or poor PCB fabrication, almost zero. Of course, this only applies if Raspberry Pi Pico is soldered directly to our board. If you want to use headers to attach Raspberry Pi Pico, then these copper keepouts, and also the USB holes, are unnecessary and can be removed.

NOTE

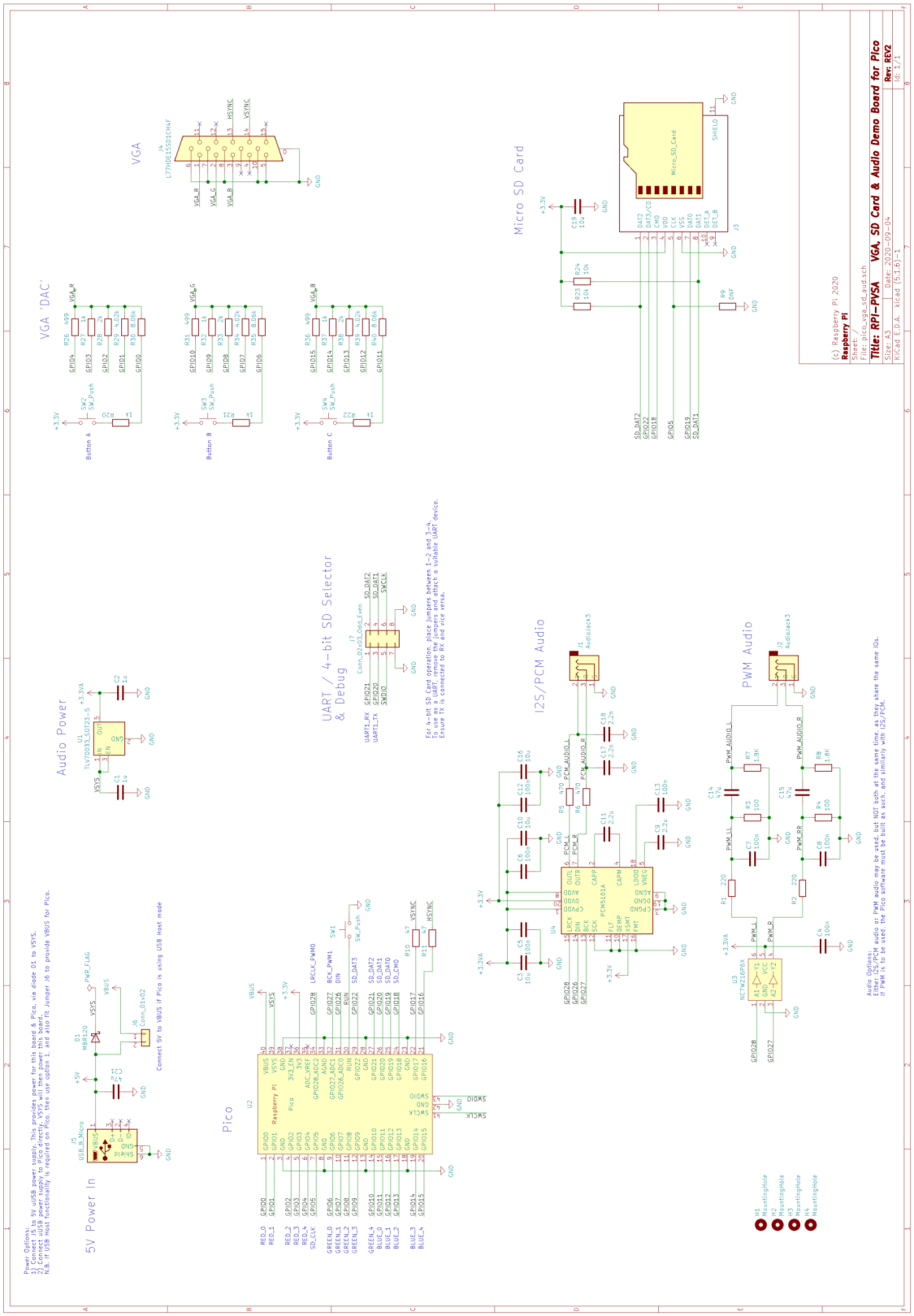
KiCad currently doesn't have a keepout layer in its footprints. The recommended approach, and the one we've used here, is to show the keepout zones on the **dwgs.user** layer, and the user must then manually remove the copper on the PCB layout itself.

This brings us to the topic of component keepouts. Obviously, if you will be directly soldering Raspberry Pi Pico to our board, then the entire footprint will have to have a component keepout underneath. If you are only ever going to attach Raspberry Pi Pico with sockets and/or headers, then you are free to place components beneath it (provided you keep them a sensible distance from the header/sockets themselves). You must make sure that the height of any components added underneath are less than the gap needed by the socket or header used.

3.6. Schematic

The final part of this guide is the schematic itself. As mentioned in the introduction to this guide, the actual KiCad schematic files are available, and you are encouraged to go and check them out, particularly as the schematic drawing shown below could very well now be outdated.

Figure 28. The complete schematic



Appendix A: Using the Rescue Debug Port

A.1. Overview

The Rescue Debug Port (DP) on RP2040 can be used to reset the chip into a known state if the user has programmed some bad code into the flash. For example some code that turned off the system clock would stop the processor debug ports being accessed, but the rescue DP would still work because it is clocked from the `SWCLK` of the SWD interface.

On the Raspberry Pi Pico, the BOOTSEL button can be used to force the chip into BOOTSEL mode instead of executing the code in flash. The rescue DP is intended for use in designs that use an RP2040 but don't have a BOOTSEL button.

NOTE

For further information on how to configure SWD see the **Getting started with Raspberry Pi Pico** book.

A.2. Activating the Rescue DP from OpenOCD

The RP2040 port of OpenOCD provides two targets:

- `rp2040.cfg`
- `rp2040-rescue.cfg`

`rp2040-rescue.cfg` connects to the rescue debug port with id `0xf`.

To use the rescue DP, start OpenOCD with the `rp2040-rescue` configuration.

```
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040-rescue.cfg
...
Warn : gdb services need one or more targets defined
Now attach a debugger to your RP2040 and load some code
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections

Ctrl + C
```

Now start OpenOCD with the normal `rp2040` configuration.

```
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

To verify the rescue DP restarted the chip, you can check the `VREG_AND_POR.CHIP_RESET` register: `0x40064008`. Bit 20 of this register is the `HAD_PSM_RESTART` bit.

In another terminal connect to the OpenOCD telnet port and use `mdw` (memory display word) to read the `CHIP_RESET` register. If the rescue DP restarted the chip, then the value will be `0x00100000`, aka bit 20 will be set.

```
$ telnet 127.0.0.1 4444
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> mdw  0x40064008
0x40064008: 00100000
```

You can now load code as described in Use GDB and OpenOCD to debug Hello World in **Getting started with Raspberry Pi Pico** book.

Raspberry Pi Trading Ltd