ESP32s – Not all Pins are Equal!

Peter Wentworth, September 2018. cspwcspw@gmail.com

The easiest way to experiment with an ESP32 is using a development kit. A devkit will have an ESP32 module, some flash memory, a USB connection, some switches (EN and BOOT) and a chip to provide USB to serial connectivity. This makes it easy to plug your devkit into your USB port on your computer and talk to it. The devkit also "breaks out", or connects, some of the connections on the ESP32 to external pins on devkit board. In the top-left of the diagram below, GPIOs 34, 35, 36, and 39 are broken out. GPIO 36 and GPIO 37 are not. The devkit board is also breadboard-friendly (i.e. pin spacing and size is standardized and will slot into your breadboard).

But the pin usage on an ESP32 is sometimes complicated, and on a devkit board it may be even more complicated. The internet is loaded with complaints about "I could not get this pin to do that, etc." So here is what I have found so far. Anyone with additional comments please email or correct any of my mistakes or misconceptions.

The document discusses some key ideas and then has a bigger worked example at the end: I connected an OV7670 camera (needing 8 parallel data bits and 6 control lines) to my ESP32, and I also connected a parallel LCD Arduino shield (needing another 8 parallel data lines and two control lines). So I needed 24 GPIO pins. And I wanted to reserve the two pins for serial communication with my computer.

Below is the typical devkit pin-out diagram from Neil Kolban's book: The red annotations are mine.

I occasionally reference the book as [Kolban, pg xyz]. These page numbers refer to the September 2018 version of the book. But since you can search the PDF, my reference should get you close.

If you count the GPIOs, you'll see that my project uses every single one – I do not even have a single pin left free. So understanding the constraints for each pin was critical.

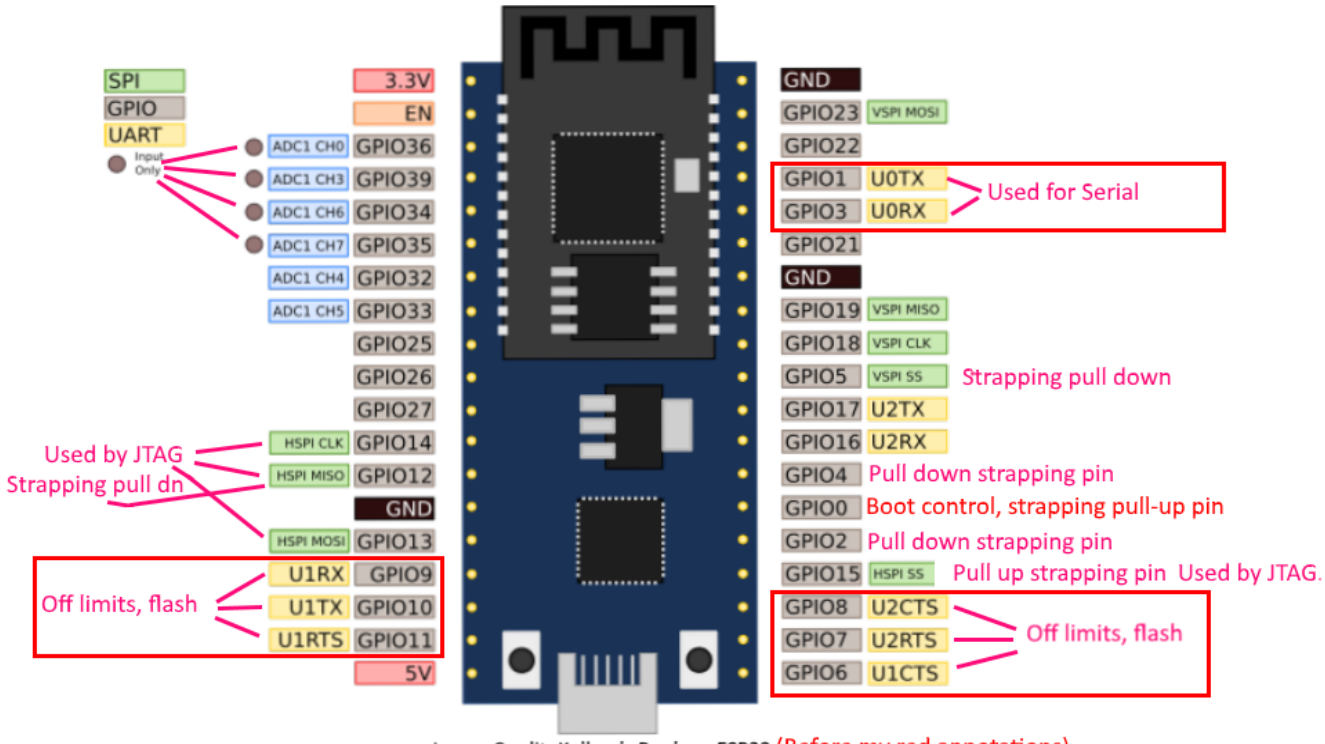


Image Credit: Kolban's Book on ESP32 (Before my red annotations)

Flash Memory needs pins

Let us start with the six off-limit pins in the boxes at the bottom. They are already in use for the flash memory that comes on the devkit. Using them for anything else will almost certainly give trouble. If anyone knows any good reason why they are exposed on the Devkit boards, please enlighten me.

Serial communication needs pins

The next two pins are also in a box in the diagram above GPIO1 and GPIO3. They are used for Serial transmit and receive. So if you want to be able to get your computer to talk to your board, don't attach other devices to these pins. If you are going to deploy your ESP32 in the field and need some extra pins to talk to a sensor, etc., you can use these. But then you lose your normal Serial communication capability, and you have to find more sophisticated ways of downloading your code onto the ESP32.

Some pins are only for input, without pull-up or pull-down capability

On the ESP chip, pins 34-39 are input only, with no pullups or pulldown logic. On most devkits, pins 37 and 38 are not broken out. So the top four pins on the left of the diagram above are "special case".

JTAG debugging https://en.wikipedia.org/wiki/JTAG

At boot time, four pins are internally routed to the JTAG debugging circuitry on the ESP32 chip. These are pins 12, 13, 14, 15. Any attempt to use these pins for any purpose other than via a JTAG debugger can give weird results – the JTAG engine can reboot your ESP32, it can write things in memory, and

generally causes chaos. So if you want to use the pins for non-JTAG purposes, you need to first steal back the pins for general GPIO usage … https://www.esp32.com/viewtopic.php?t=2687

```
#include "driver/gpio.h"
void reclaim_JTAG_pins()   // Call this during setup
{
   PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[12], PIN_FUNC_GPIO);
   PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[13], PIN_FUNC_GPIO);
   PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[14], PIN_FUNC_GPIO);
   PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[15], PIN_FUNC_GPIO);
}
```

Strapping Pins – six of them

GPIOs 0, 2, 4, 5, 12 and 15 are strapping pins. That means that they are permanently strapped to a default logic value – either high/1 or low/0 (with weak resistors, so we can override them). GPIOs 0, 5, 15 are strapped to pull up, GPIO 2, 4, 12 are strapped to pull to ground. [https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf pg 11]

At boot time the system reads the high/low values of the strapping pins and makes configuration decisions: whether to start a bootloader or whether to start the user's program (GPIO 0), whether to output boot-time diagnostic information (GPIO 15), what external voltage levels to use for I/O devices, etc.

We can use jumpers or switches (like BOOT) to override the default logic value on the strapping pins.

What does this mean for us? If we need to use strapping pins for general GPIO, we have to arrange that our devices do not pull the pin the "wrong way", or we won't be able to boot properly.

My LCD screen and my camera have some "input-only" control pins (inputs to the LCD, output from the ESP32). Input-only pins on your external device will usually have high impedance. So they are generally safe to attach to strapping pins. They won't drag the pin to an unwanted logic level, and interfere with the boot process. Conversely, if you connect an output from an external device like a camera to a strapping pin, you have to be very, very careful. If you can guarantee that when the reset or power-on occurs your external device will produce logic 0 on an output pin, you could connect that output to a pull-down strapping pin and all should be fine. Similarly, if you know that a bus floats or is pulled high, you could connect that to a pull-up strapping pin without getting into boot-time trouble.

GPIO pins are driven through ports

GPIOs 0-31 attach to one port called ***out*** within the ESP32, while GPIOs 32-39 attach to a different port called ***out1***. So there are two groups of GPIO pins. ***Direct Port Manipulation*** allows us to set to 1 any selection of the pins attached to a port by writing a 32-bit mask to a port register called ***w1ts*** - "write 1 to set". A companion port register called ***w1tc*** (write 1 to clear) will clear specific GPIO pins on a port according to the bit mask. [Kolban, ***Register based GPIO***, pp 549]

In the example at the end of this document, we used direct port manipulation to output byte-at-a-time to the parallel LCD screen. If all 8 wires are on same port (i.e. all GPIOs are selected from 0-31), we can

clear all 8 data pins using one port write, and then set the bits we need to be 1 bits in another port write. Note that the second port for pins GPIO pins 32-39 only has two output pins anyway (as explained earlier - GPIOs 34-39 are input only).

So when you design your project, if you are going to be using multiple wires in a parallel fashion, and you want the speed that direct port manipulation can bring, you should group all your input or output lines (perhaps even some control lines) onto the same port.

Power Domains

Caveat (beware) – I don't really understand the full implications of this, so any comments or clarifications (or if you'd like to contribute a section to my document) are welcome.

There are four different power domains / power supplies on the ESP32. Power and voltage levels for attached I/O devices may not be the same as voltages used to run the CPU, or the clocks. There is also an analog power domain. A "domain" really means "the collection of devices, clocks, pins, processors, components, logic units" that depend on this power source.

So, for example, when the chip goes into deep-sleep mode to save power, power to some domains is switched off, while the VRTC power supply (and the components in its domain) are still active. Some pins can wake up the processor from a deep sleep, but only if they're powered on the VRTC power domain. The hairy details are at
http://wiki.ai-thinker.com/_media/esp32/docs/esp32_chip_pin_list_en.pdf

WiFi

Perhaps the most frequent clash of functionality in the ESP32 is that WiFi circuitry shares some of the circuitry and pins used by the Analog to Digital Converter 2 (ADC2).

There is a lot of discussion, and a lot of confusion (in my mind, at least) about exactly what this means. If WiFi is active, A to D conversions that use ADC2 do not work. The ADC2 channel gets input from 10 pins: GPIO 0,2,4,12,13,14,15,25,26,27.

It seems there are plans to fix the clash: https://github.com/espressif/esp-idf/issues/461
https://github.com/espressif/arduino-esp32/issues/102
Here are the main points of my confusion – any clarification will be welcome …

- Are all these pins off limits when Wifi is active?
- Can the pins be used for general I/O – is it just A/D conversion that fails?
- Someone suggested that it is just WiFi startup that is problematic – after that the pins work fine. While WiFi is active, exactly what are we not allowed to do on what pins?
- Does the same restriction / problem apply to Bluetooth?

Direct I/O MUX

Some high-speed peripheral modules have signals that, by default, are directly connected to pre-assigned GPIO pads.

The ESP32 also has a very general GPIO multiplexer that allows reconfiguration of which signals connect where. But, routing through the multiplexer is slower than using the direct connection.

Looking at the diagram of the devkit board again, the outermost labels (coloured blue, green, yellow) represent the default direct I/O connections. So let us assume we have an SPI device and want to drive it through the VSPI hardware circuitry. That module is already directly connected to pins 5, 18, 19, and 23, and this will allow clock speeds of 80MHz.

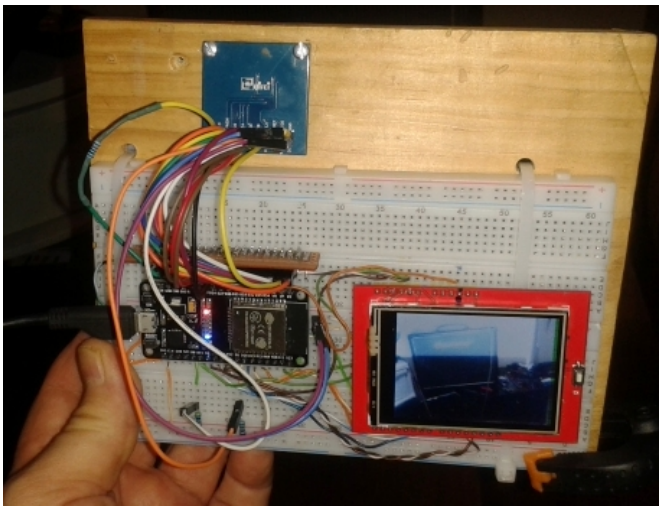
However, we can redirect the VSPI component through the GPIO multiplexor to use different GPIO pins, but the extra switching delays through the multiplexor will now limit your maximum clock speed to 40MHz.

You'd need a very good reason, and will pay a price, if you decide not to stick with using the pins that are already directly connected.

Many more constraints

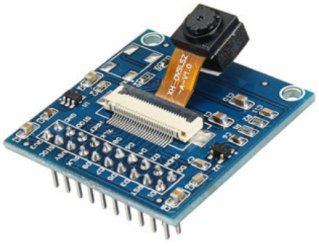
There are two Digital to Analog pins, ten touch pins, a sensitive differential amplifier using pins VN and VP, and so on. If you project uses features like this, you will need more even care!

My Example Project



I use an ESP32 WROOM devkit to input data from an OV7670 camera and then display the frames on an ILI9341 LCD Shield. On a breadboard I can get 320x240 RGB565 pixels at 25 frames per second. I can also switch into 160x120 or 80x60 camera modes.

Both devices are parallel data devices. That means they can transfer data byte-at-a-time, but it also means that we need 8 data lines for the inbound camera data, and 8 more data lines for output to the LCD. On top of that, we need control lines for both devices. So a real constraint in this project is the number of pins we have available, and I needed some tricks to make it all fit.



The OV7670 camera module exposes 20 pins in the header, with 10 of those

for data lines. In the RGB656 colour mode, the two least significant data bits d0 and d1 are left unconnected (18 to go). Two are for power (16 to go). The reset line can be directly wired to the ESP32 EN pin(15 to go) and the PWDN line can be tied via a resistor to ground (14 to go). But we need 14 GPIOs, no less.

The two wire I2C interface carries the SCCB (Serial Camera Control Bus) commands. Note that commands and setup of the camera registers is a two-wire serial affair, and it won't matter it if is slow. But pixel data arrives really fast, so this is the parallel part of things.

Multiple devices can be on the same I2C bus, each with their own built-in address so that they know which commands are intended for it. This camera has a hard-wired I2C address 0x21. Each address is shifted left by one bit, and the least significant bit denotes a "write" or a "read" command on the bus. So you'll see other documentation that says the address is 0x42. That is the "write" address of device with base address 0x21. I used the **Wire** library to talk to the device, and **Wire** wants the address specified as base 0x21.

I2C buses are "open-drain" - that means devices actively pull the line low, but the line must pull itself up – they must not drive the line high. That prevents multiple devices burning each other out – one device driving the line high with a current source, one sinking the current as it tries to drive the line low. If the bus protocol is "drive low, float high" and everyone obeys the protocol, you should have no trouble even if devices try to use the bus simultaneously. The pull-up resistors I used on these two lines were 4.7K ohm. The camera pins for control information are called SIOD and SIOC (data and clock).

My choice was to use ESP32 pins GPIO 05 for SIOD, and GPIO 00 for SIOC. Both are strapping pins, with default high pullups. So adding my 4.7K ohm bus pull-ups didn't change boot-up behaviour. The internal ESP32 pullups are not sufficiently strong to pull the bus high, so my pull-ups are needed.

The camera does not generate its own clock. So an ESP32 pin is configured using some LED dimming PWM logic to generate a fast clock: this is fed into XCLK on the camera. XCLK and SIOC are the only high-impedance inputs to the camera, All other not-yet-connected camera pins are outputs from the camera. I chose GPIO 26 for XCLK. Only after the XCLK starts running is the I2C control logic in the camera responsive. So at this point with only the power, RESET, PWDN, SIOD, SIOC and XCLK pins connected, we can send commands, receive responses, debug the parameters we need for the Wire library, etc. I wrote a small test suite to read camera registers and validate content against expected values, and also to write and read back a register. This validates the SCCB communication.

All the other pins on the camera are for flow of data and timing signals from the camera to the ESP32:

```
int cam_databus[] = {  13, 35, 12, 32, 14, 33, 27, 25}; // lsb on the right
int VP = 36;  // Some call these "friendly names" for pins. It depends whether
int VN = 39;  // your dev board is silkscreened with these names, or with GPIO numbers.
//                     SIOD, SIOC, VSYNC, HREF, XCLK, PCLK ...
theCam = new pw_OV7670(05,    00,     VN,   34,   26,   VP, cam_databus,…
```

I chose these pins (and XCLK) because they were all on the same side of the devkit, and I could keep all the higher-speed jumper wires shorter. Notice that pins 12, 13, and 14 are used – normally defaulted

for JTAG debugging, so one has to steal aback those pins for general IO before initializing the camera, as discussed earlier. I also “used up” those ESP GPIO pins that can only do input. I need pins that can do output for the LCD side of things.

I was ready to hook up the LCD but not many pins left! I have to have 8 data pins. There are 5 control lines on this LCD shield, but only two more pins free on the ESP32. (Plus the two GPIO 01 and GPIO 03 are still free – the primary serial communication port, which I don’t want to sacrifice.) Some experimentation assured me that I could get away with only two control pins. Here is how:

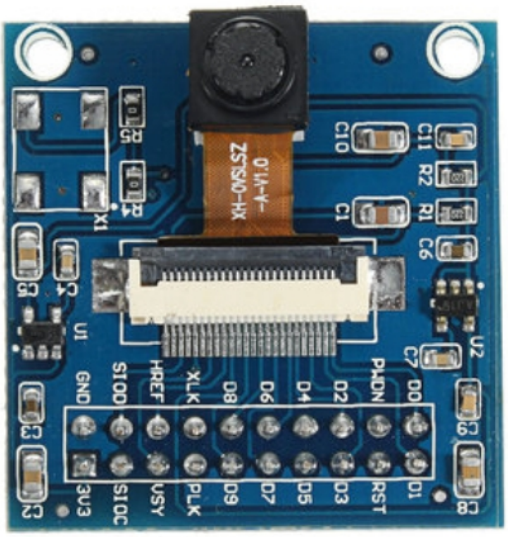
```
int lcdDataBus[] = {18, 4, 19, 21, 22, 23, TX2, RX2}; // D7, D6, D5, D4, D3, D2, D1, D0
#define LCD_RESET -1
// I connect this to EN on my ESP32, since I have no need to reset the LCD
// independently of resetting the system.  I use -1 to mean "no pin".

#define LCD_RD -1 // In principle, one can read ID info, etc.from the LCD controller.
                  // In this project, I don't. So I just pull the RD pin high.
#define LCD_WR  2 // When it goes high, the LCD latches bits from the output pins. Vital.
#define LCD_DC 15 // HIGH means data is pixel data, LOW means data is a command. Vital.
#define LCD_CS -1 // CS low means the LCD pays attention. For this project, strap it low.
```

Both pins GPIO 02 and GPIO 15 are strapping pins, but I’m safe because I’ve made sure that I’ve attached them to high-impedance pins that are input lines (WR and DC) on the LCD. Also, all these pins are on the same port, because my LCD driver uses direct port manipulation.

I’ll publish code and further details / perhaps a video about my project soon. I hope this document helps to consolidate many of the factors to keep in mind as you plan your pin mapping for your project.

Summary of connections for this project:

Camera



OV7670 **DEVKIT**

GND ------ GND

3v3 ------ 3v3

SIOD <----> GPIO 05 with 4.7K pullup to 3.3v

SIOC <----> GPIO 00 with 4.7K pullup to 3.3v

HREF ----> GPIO 34

VSY ----> VN

XLK <---- GPIO 26

PLK ----> VP

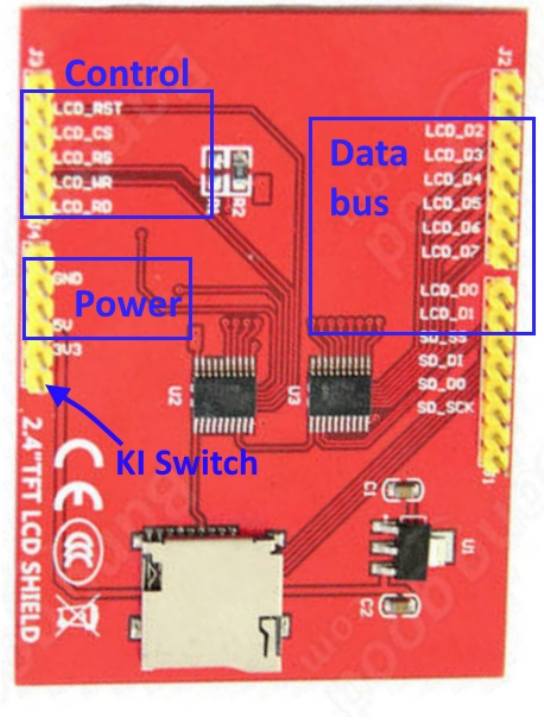
D2..D9 ----> GPIOs 25,27,33,14,32,12,35,13

PWDN tied via 10K resistor to ground.

RST <---- EN

D0 and D1 are left unconnected.

LCD SHIELD



DEVKIT **LCD Shield**

VIN (5V) ------ 5V (3.3v did not work for me.)

GND ------ GND

GPIO 02 -----> LCD_WR

GPIO 15 -----> LCD_RS (Called LCD_DC in my code)

EN -----> LCD_RST (RESET)

GND -----> LCD_CS (tied low)

3.3v ----> LCD_RD tied high via 10K resistor

GPIOs 18, 4, 19, 21, 22, 23, TX2, RX2 ---->
LCD_D7, LCD_D6 … LCD D0

The K1 Switch pin can also be connected to EN or to GPIO 00. The EN connection means the switch will work like the EN switch on the devkit – it will reboot the system. A GPIO 00 connection means it will work like the BOOT switch on the devkit – ready the bootloader for new firmware.

Last revised: 9 Sept 2018