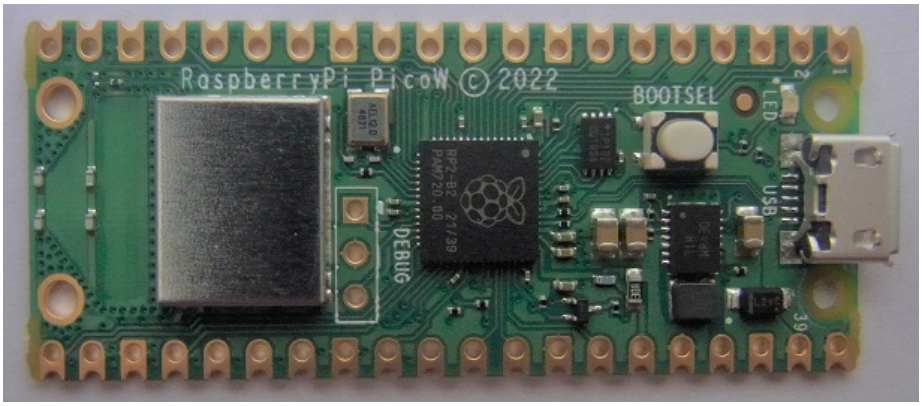
Lean2

Embedded systems without the bloat

PicoWi: standalone WiFi driver for the Pi Pico W

Introduction



Raspberry Pi Pico W

The aim of this project is to provide a fast WiFi driver for the CYW43439 chip on the Pi Pico W module, with C code running on the RP2040 processor; it can also be used with similar Broadcom / Cypress / Infineon chips that have an SPI interface, such as the CYW4343W.

It is based on my Zerowi project which performs a similar function on the Pi Zero device CYW43438, which has an SDIO interface. However, due to myriad difficulties getting the code running, the code has been restructured and simplified to emphasise the various stages in setting up the chip, and to provide copious run-time diagnostics.

The structured approach of the WiFi drivers is mirrored in the example programs, and the individual parts of this blog; they range from a simple LED-flash program, to one that provides some TCP/IP functionality.

A major problem with debugging Pico-W code is the difficulty attaching any hardware diagnostic tools, such as an oscilloscope or logic analyser; this has been addressed by supporting an add-on board with a Murata 1DX module and CYW4343W chip; full details are given in part 1 of this project.

Development environment

For simplicity, I use a Raspberry Pi 4 to build the code and program the Pico, with two I/O lines connected to the Pico SWD interface. This is really easy to set up, using a single script that installs the SDK and all the necessary software tools on the Pi 4:

```
wget https://raw.githubusercontent.com/raspberrypi/pico-setup/mast
chmod +x pico_setup.sh
./pico_setup.sh
```

For SWD programming, the Pico must be connected to the I/O pins on the Pi as follows:

```
Pico SWCLK   Pi pin 22 (GPIO 25)
Pico GND     Pi pin 20
Pico SWDIO   Pi pin 18 (GPIO 24)
```

The serial interface is used extensively for displaying diagnostic information; pin 1 of the Pico is the serial output, and pin 3 is ground. I use a 3.3 volt FTDI USB-serial adaptor to display the serial output on a PC, but the Pi 4 serial input could be used instead.

The PicoWi source code is on github:

```
cd ~
git clone http://github.com/jbentham/picowi
cd ~/picowi/build
chmod +x prog
chmod +x reset
```

I've included the necessary CMakeLists.txt, so the project can be built using the following commands:

```
cd ~/picowi/build
cmake ..      # create the makefiles
make picowi   # make the picowi library
make blink    # make the 'blink' application
./prog blink  # program the RP2040, and run the application
```

When building the current pico-SDK, there are some 'parameter passing' warnings when the pio assembler is compiled; these can be ignored.

Compilation is reasonably fast on the Pi 4; once the SDK libraries have been built, you can do a complete re-build of the PicoWi library and application within 10 seconds, and reprogram the RP2040 in under 3 seconds.

The 'reset' command is useful when you just want to restart the Pico, without loading any new code.

If OpenOCD reports 'read incorrect DLIPDR' then there is a problem with the wiring. I've set the SWD speed to 2 MHz, which should work error-free, providing the wires are sufficiently short (e.g. under 6 inches or 150 mm) and there are good power & ground connections between the Pi & Pico. I use a short USB cable to power the Pico from the Pi, and this is generally problem-free, though sometimes the Pi won't boot with the Pico connected; this appears to be a USB communication problem.

Compile-time settings

There is currently only one setting in the CMakeLists.txt file, to choose between the on-board CYW43439 device, or an external CYW4343W module:

```
# Set to 0 for Pico-W CYW43439, 1 for Murata 1DX (CYW4343W)
set (CHIP_4343W 0)
```

The Pico-specific settings are in picowi_pico.h:

```
#define USE_GPIO_REGS   0       // Set non-zero for direct registe
                                // (boosts SPI from 2 to 5.4 MHz r
#define SD_CLK_DELAY    0       // Clock on/off delay time in usec
#define USE_PIO         1       // Set non-zero to use Pico PIO fc
#define PIO_SPI_FREQ    8000000 // SPI frequency if using PIO
```

These affect the way the SPI interface is driven; the default is to use the Pico PIO (programmable I/O) with the given clock frequency; 8 MHz is a conservative value, I have run it at 12MHz, and higher speeds should be possible with some tweaking of the I/O settings.

Setting USE_PIO to zero will enable a 'bit-bashed' (or 'bit-banged') driver; this can run over 7 MHz if using direct register writes, or 2 MHz if using normal function calls.

You'll note that I haven't included a driver for the SPI peripheral inside the RP2040; this would have been easier to use than the PIO peripheral, but the on-board CYW43439 chip isn't connected to suitable I/O pins. The actual pins used are defined in picowi_pico.h:

```
#define SD_ON_PIN       23
#define SD_CMD_PIN      24
#define SD_DIN_PIN      24
#define SD_D0_PIN       24
#define SD_CS_PIN       25
#define SD_CLK_PIN      29
#define SD_IRQ_PIN      24
```

You'll see that pin 24 is performing multiple functions; this hardware configuration is discussed in detail in the next part of this blog. If you are using an external module, the pin definitions can be modified to use any of the RP2040 I/O pins.

Diagnostic settings

My code makes extensive use of a serial console for diagnostic purposes, and I generally use an FTDI USB-serial adaptor connected to pin 1 of the Pico module to monitor this. In view of the large amount of information that can be produced, I modify the serial interface to run at 460800 baud:

```
Edit pico-sdk/src/rp2_common/hardware_uart/include/hardware/uart.h
Change definition to:
  #define PICO_DEFAULT_UART_BAUD_RATE 460800
```

I originally tried running the interface at 921600 baud, but this resulted in occasional characters being lost, which is a major problem when trying to understand what is going wrong with the code.

You can use the Pico USB link instead; it must be enabled in the CMakeLists.txt, using the name of the main file, for example to enable it for the 'ping' example program:

```
pico_enable_stdio_usb(ping 1)
```

Then you can use a terminal program such as minicom on the Pi 4 to view the console:

```
# Run minicom, press ctrl-A X to exit.
minicom -D /dev/ttyACM0 -b 460800
```

A disadvantage of this approach is that when the Pico is reprogrammed, its CPU is reset, which causes a failure of the USB link. After a few seconds, the link is re-established, but there will be a gap in the console display, which can be misleading. Also, there is the possibility that the extra workload of maintaining a (potentially very busy) USB connection might cause timing problems, so if you are making extensive use of the diagnostics, it might be necessary to use a hard-wired serial interface.

You can control the extent to which diagnostic data is reported on the console; this is done by inserting function calls, rather than using compile-time definitions, to give fine-grained control. The display options are in a bitfield, so can be individually enabled or disabled, for example:

```
// Display SPI traffic details
set_display_mode(DISP_INFO|DISP_EVENT|DISP_SDPCM|
                 DISP_REG|DISP_JOIN|DISP_DATA);

// Display nothing
set_display_mode(DISP_NOTHING);

// Display ARP and ICMP data transfers
set_display_mode(DISP_ARP|DISP_IP|DISP_ICMP);
```

WiFi network

For the time being, the code does not support the Access Point functionality within the WiFi chip. It can only join a network that is unencrypted, or with WPA1 or WPA2 encryption, as set in the file picowi_join.h:

```
// Security settings: 0 for none, 1 for WPA_TKIP, 2 for WPA2
#define SECURITY            2
```

The network name (SSID) and password are defined in the 'main' file for each application, e.g. join.c or ping.c, which means they are insecure, as they can be seen by anyone with access to the source code or binary executable:

```
// Insecure password for test purposes only!!!
#define SSID                "testnet"
#define PASSWD              "testpass"
```

Other resources

The data sheets for the CYW43439 and CYW4343W are well worth a read, as they contain a good description of the low-level SPI interface, but contain nothing on the inner workings of these incredibly complicated chips. The Infineon WICED development environment has very comprehensive coverage of the WiFi chips, though it would take some work to port this code to the RP2040. The Pi Pico SDK contains the full source code to drive the CYW43439, with the lwIP (lightweight IP) open-source TCP/IP stack.

I'm using a different approach, with a completely new low-level driver, and a built-in IP stack to maximise throughput, as described in the following parts:

Introduction
Part 1: low-level interface
Part 2: initialisation
Part 3: IOCTLs and events
Part 4: scan and join a network
Part 5: ARP, IP and ICMP
Source code

I'll be releasing updates with more TCP/IP functionality.

Copyright (c) Jeremy P Bentham 2022. Please credit this blog if you use the information or software in it.

December 6, 2022 / Pico, WiFi /

2 thoughts on “PicoWi: standalone WiFi driver for the Pi Pico W”

tony1tf

December 7, 2022 at 9:45 am

Wow! As usual Jeremy, we can but wonder at your ability to get into the real nitty gritty of these devices and offer practical help. I havn’t found out what you want to achieve with this interface, compared with the existing drivers provided by R Pi. Perhaps you are going to show the speed improvements in later episodes. Do you have contacts at Broadcom or R Pi? I’m sure they would be happy to employ you!

★ Like

iosoftcode

December 7, 2022 at 11:28 am

I’m too old to be employed by anyone, but thanks for the compliment. My goal is to provide a very fast Web interface for measurement & streaming applications, but it’ll take a lot more work to achieve that. I could have used the lwip TCP/IP stack with my low-level code, but took the brave/foolish decision not to do that, we’ll see if I’ve made the correct choice.

★ Like

Lean2 / Blog at WordPress.com.