CYW43439 Bluetooth

The CYW4349 wifi and bluetooth chip is used in the Raspberry Pico Pi W. The C SDK includes the required drivers and stacks to run this for bluetooth and WIFI.

This document describes how the chip interface works for blueooth, with the aim to create a Micropython implementation for a HCI interface layer.

Conventions used for numbering

Because this description uses both byte arrays and u32 values, the following is used to clarify which is in use.
All u32 values are used with underscore separators at byte boundaries, so a hex number 0x12345678 would be shown as 12_34_56_78.

When this is stored in a byte array, this may be with different word lengths and little endian or big endian. Byte arrays are not shown using the underscore, so a little endian array for the value above would be shown as 78 56 34 12, and a big endian representation is 12 34 56 78

HCI interface

The HCI layer has the following format. The header for the SPI packet is the length plus HCI event.

Length is number of bytes in the data in the packet, which excludes the HCI event.

The header for the SPI packet is the length plus HCI event.

Packet header		
Length	**HCI event**	**Data**
3 bytes	1 byte	n bytes

Packet format

Packet header		
Length	**HCI event**	**Data**
06 00 00	04	0e 04 01 03 0c 00

Example packet

CYW4349 interface

The CYW4349 interface uses half-duplex SPI to communicate to registers and memory in the chip.

With SPI every data transfer is 32 bits, so all data is 32 bit aligned and the length rounded up to the next 32 bit value.

Each command to the CYW43439 has a 32 bit command field and then following data values.

Command	Data
32 bits	n x 32 bits

SPI command format

Write	Incr	Function	Address	Size
1 bit	1 bit	2 bits	17 bits	11 bits

SPI command field layout

With this bit layout.

0	0	00	00	00	00	00	00	00	00	00	0	0	00	00	00	00	00
wr	in	fn	address									size					
1	1	0-3	0 00 00 to 1 ff ff									0 00 to 3 ff					

Bit layout of command field

w	i	ff	aaaa	aaaa	aaaa	aaaa	a sss	ssss	ssss

32-bit word layout of command field

Field	Explanation
Write / read	1 Write to CYW43439 0 Read from CYW43439
Increment	
Function	0 BUS 1 BACKPLANE 2 WLAN
Address	Address in memory or of register
Size	Size of data (not rounded to 32 bits)

Explanation of fields

The address depends on which Function is being accessed.

Memory map

Function	Description	Address range
SPI	gSPI registers	00 – 1f
Backplane		0001_0000
Backplane	SDIO_FUNCTION2_WATERMARK	0001_0008
Backplane	BACKPLANE_REG	0001_000a
Backplane	SDIO_CHIP_CLOCK_CSR	0001_000e
Backplane	SDIO_PULL_UP	0001_000f
Backplane	WIFI firmware	0000_0000
Backplane	WIFI base	0006_861c
Backplane	H2BT_BUFFER	0006_861c (0x0000)
Backplane	BT2H_BUFFER	0006_961c (0x1000)
Backplane	SEND_HEAD	0006_a61c (0x2000)
Backplane	SEND_TAIL	0006_a620 (0x2004)
Backplane	RECEIVE_HEAD	0006_a624 (0x2008)
Backplane	RECEIVE_TAIL	0006_a628 (0x200c)
Backplane	BT2WLAN_PWRUP_ADDR	0064_0894
Backplane	CHIPCOMMON_BASE	1800_0000
Backplane	CHIP NUMBER	1800_0000
Backplane	WLAN_BASE_ADDR	1800_0d68
Backplane	HOST_CONTROL	1800_0d6c
Backplane	BT_CONTROL	1800_0c7c
Backplane	SDIO_BASE	1800_2000
Backplane	SDIO_INT_HOST_MASK	1800_2024
Backplane	WLAN_ARMCM3_BASE	1800_3000
Backplane	SCOSRAM_BASE	1800_4000
Backplane	SOCSRAM_BANKX_INDEX	1800_4010
Backplane	SOCSRAM_BANKX_PDA	1800_4044
Backplane	BTFW_MEM_OFFSET	1900_0000

Backplane addresses

Backplane memory and registers are accessed via a 15 bit address range, from 0x0 to 0x7fff. This creates a window into the full memory range and the base address for this window can be set.

As the address field is 17 bits, there are two special bits.

Bit 16 is set to show a 32 bit (4 byte) data transfer, so is always set for SPI transactions (0x8000)

Bit 17 is unknown but is set for the backplane registers (0x1 000a throught 0x1 000c), so may indicate a fixed address regardless of the current backplane settings.

Key register addresses

Function	Address	Field	Explanation
BACKPLANE	18_00_0c_7c	BT CTRL	Controls bluetooth
BACKPLANE	18_00_0d_6c	HOST CTL	Controls host
BACKPLANE	18_00_0d_68	BASE ADDRESS	Base address for WIFI (and buffers)

Key registers for bluetooth

Register address	Field	Mask	Explanation
HOST CTRL	DATA VALID	00_00_00_02	Tell chip data is now valid (toggle value)
HOST CTRL	WAKE BT	00_02_00_00	Wake up the BT chip
HOST CTRL	SW READY	01_00_00_00	Set host as ready
BT CTRL	BT AWAKE	00_00_01_00	Is the BT chip awake
BT CTRL	FW READY	01_00_00_00	Is the BT chip ready

Bit fields for key registers

Setting the backplane base address

Addresses are 32 bit. The chipset base address for the backplane window is 0x18 00 00 00.

The backplane window is set as the top three bytes of the full address.

Any memory access is therefore made up of the command address and the backplane address.

	Byte 3	Byte 2	Byte 1	Byte 0
Backplane address	ff	ff	ff	
Field address		01	ff	ff

Mapping of addresses via backplane and field address

Function	Address	Field	Explanation
BACKPLANE	1_000a	WINDOW LOW	Byte 1 of window address

BACKPLANE	1_000b	WINDOW MID	Byte 2 of window address
BACKPLANE	1_000c	WINDOW HIGH	Byte 3 of window address

Backplane address mask registers

Transmit and receive buffers

To send HCI data to the chip and receive data from the chip, data is stored in transmit and receive circular buffers in the chip memory.

Each buffer is 0xfff bytes long (so is larger than the backplane window)

These have a specific address range, offset from the WIFI base address.

The buffer for host to chip data is offset at 0x000 and the chip to host buffer is at 0x1000

There are also pointers for the head and tail of each buffer stored in registers.

The key registers and memory locations in use are:

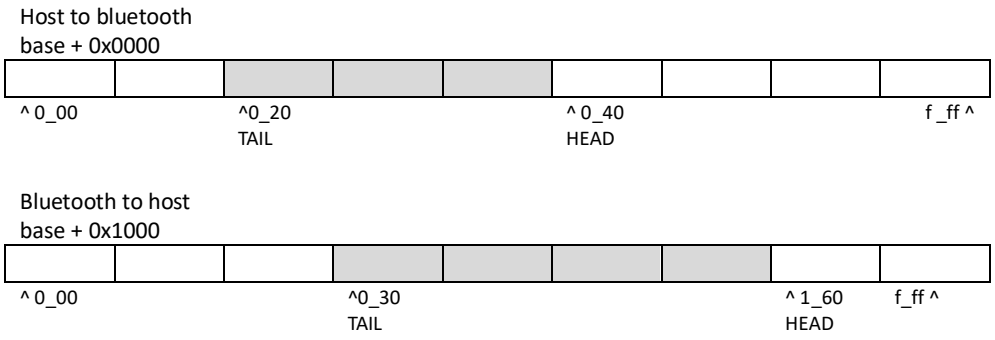
Address		**Memory location** (offest from WIFI base)
Host to bluetooth buffer		00_00
Bluetooth to host buffer		10_00
Host to bluetooth	Head	20_00
	Tail	20_04
Bluetooth to host	Head	20_08
	Tail	20_0c

Buffer registers

The example shows the two buffers in use, with the head and tail locations marked. WIFI base is 0x6_861c. To reach this the backplane window is set to 0x00_06_80_00 and offset of 0x06_1c is used (or 0x16_1c or 0x26_1c). When the 0x80 00 SPI mask is added, this becomes 0x86_1c (or 0x96_1c or 0xa6_1c).

			Example	
Address		**Memory location**	**Address**	**Values**
Host to bluetooth buffer		00_00	6_86_1c	
Bluetooth to host buffer		10_00	6_96_1c	
Host to bluetooth	Head	20_00	6_a6_1c	0_40
	Tail	20_04	6_a6_20	0_20
Bluetooth to host	Head	20_08	6_a6_24	1_60
	Tail	20_0c	6_a6_28	0_30

Buffer example



Buffer diagram

SPI interface

The SPI interface is half-duplex, meaning it only has a single data line and can therefore only transmit or receive at one time.

The GPIO pins used are shown below.

Pin	Name	Explanation
23	REG ON	Power on
24	DATA OUT	Data out
24	DATA IN	Data in
24	IRQ	Interrupt line
25	CS	Chip select
29	CLOCK	SPI clock

GPIO pins for SPI interface

Transfers are in units of 32 bits. A 32-bit word is written to the BT chip, then a 32 bit word is read from the chip.

In a simple transfer, a register write, two 32-bits words are written and the read data is discarded.

For a backplane read, a buffer is inserted between the command and the received data. This allows for the chip to process the command and generate the response.

The length of the buffer is stored in register 0x01c (Response delay register F1). By default this is 4 bytes. For the Pico Pi CYW43439 driver this is set at 16 bytes (or 4 32-bit words).

Direction	Command	Value
Write	Write to 00_06_a6_1c	00_00_00_08
Read	00_00_00_00	00_00_00_00

Simple write to register

Direction	Command	Value				
Write	Read from 8d_6c	00_00_00_00	00_00_00_00	00_00_00_00	00_00_00_00	00_00_00_00
Read	00_00_00_00	00_00_00_00	00_00_00_00	00_00_00_00	00_00_00_00	00_01_00_01

Simple read from register

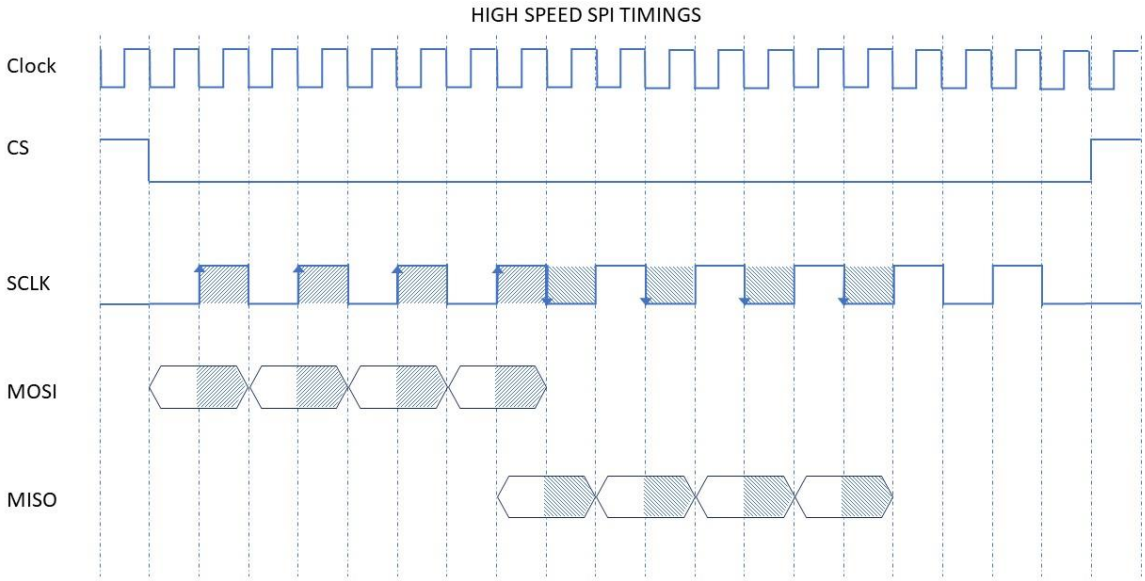
SPI physical interface

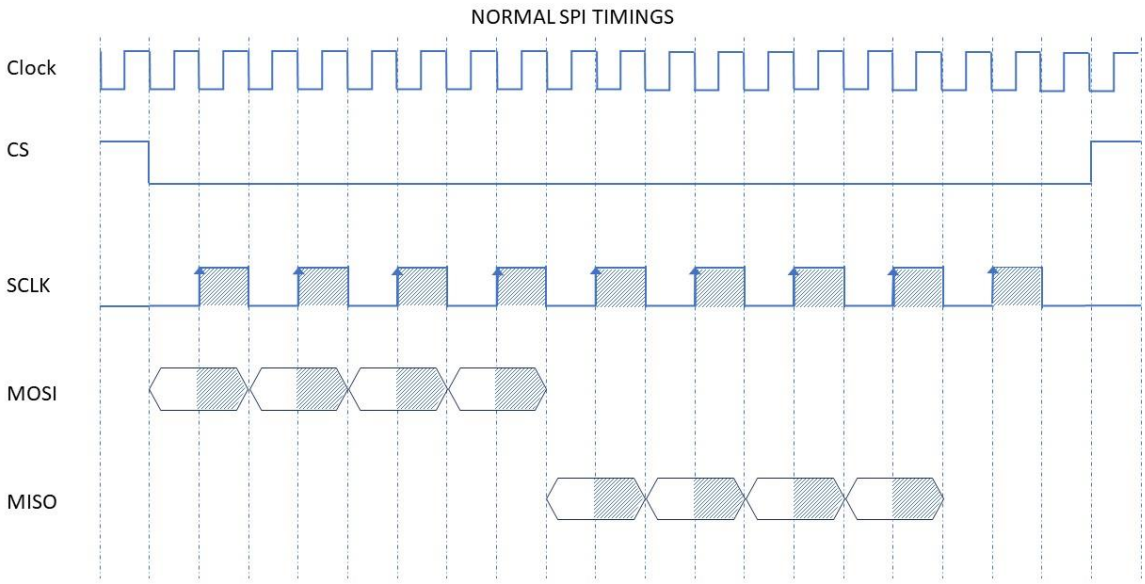
SPI from the Pico to the CYW43439 is half-duplex, which is implemented by sharing that data line between MISO, MOSI and the interrupt pin (GPIO 24).

It is also slightly complex because the write is performed before the rising edge of the clock, and the read after the falling edge of the clock. The gap between write and read seems to vary depending on the clock speed, so a slow clock allows the data to be presented to GPIO24 before the falling edge, but a fast clock then requires another cycle before the data is ready.

This is referred to as High Speed mode in the CYW specification (but not reflected in the SPI timing diagrams in that document).

Data transfers are 32 bit words and big endian, which is selected with an option on register 0x00. Before this is set the CYW defaults to 16 bit little-endian, so the command and returned data must be swapped until these options are set.




NORMAL SPI TIMINGS
Clock
CS
SCLK
MOSI
MISO

Endian settings

The CW43439 can be in one of two word length and two endian settings. The chips boots in 16 bit words, little endian.

The SPI transfers 32 bits words, normally sent most signicant bit first.

It seems for the CYW that this word is interpreted as comprising 4 bytes. The two settings above will create various translations of the data byte ordering.

The datasheet (page 18) uses terminology of C0 to C3 without explaining the significance of each label.

Assume that a buffer of four bytes is used and is ordered C0 C1 C2 C3. The settings allow you interpret the bytes in different ways. (Don't confuse buffer order and MSB / LSB – this is just an ordering of the bytes).

If big endian (32 bits or 16 bits) is used then this is transmitted as C0 C1 C2 C3, so in the order of the four bytes, and no translation is made.

For 32 bit little-endian, the transmission ordering is byte reversed to C3 C2 C1 C0.

For 16 bit little-endian, the transmission ordering is byte reversed in each 16 bit word, to C1 C0 C3 C2.

It seems the chip natually uses little endian memory access for storing 32 bit words – the test register FE_ED_BE_AD is stored as below.

Memory address	14 15 16 17
Contents	AD BE ED FE
Buffer order	C0 C1 C2 C3

Using the test register example, the transmission orders will be as below.

	Mapping	**Transmitted**
32 / 16 bit big-endian	C0 C1 C2 C3	AD BE ED FE
32 bit little-endian	C3 C2 C1 C0	FE ED BE AD
16 bit little-endian	C1 C0 C3 C2	BE AD FE ED

The command to read the test register is as below.

wr	in	fn	address	Size
0	1	00	0 00 14	0 04

wr	i	ff	aaaa	aaaa	aaaa	aaaa	a sss	ssss	ssss
0	1	00	0000	0000	0000	1010	0 000	0000	0004

The command value (u32) is 40_00_A0_04.

Endian	**Word size**	**Register x00 (endian / word size)**	**How this maps to bytes**	**Command example bytes 40_00_A0_04**	**FEEDBEAD example bytes FE_ED_BE_AD**
Big endian	32 bit	11	C0 C1 C2 C3	04 A0 00 40	AD BE ED FE
Big endian	16 bit	10	C0 C1 C2 C3	04 A0 00 40	AD BE ED FE
Little endian	32 bit	01	C3 C2 C1 C0	40 00 A0 04	FE ED BE AD
Little endian	16 bit	00 (default)	C1 C0 C3 C2	A0 04 40 00	BE AD FE ED

Simple read from register

Note that 32 bit big endian is actually the same as 16 bit big endian.

In default mode the command sent is A0 04 40 00 and the data received is BE AD FE ED. This is word swapped to AD BE ED FE, which is little endian for FE_ED_BE_AD.

This table show the bit level transmission for different settings

Setting	Bits	Bytes
11: 32 BIT	0000 0100 1010 0000 0000 0000 0100 0000	04 A0 00 40
BIG ENDIAN	1010 1101 1011 1110 1110 1101 1111 1110	AD BE ED FE
10: 16 BIT	0000 0100 1010 0000 0000 0000 0100 0000	04 A0 00 40
BIG ENDIAN	1010 1101 1011 1110 1110 1101 1111 1110	AD BE ED FE
01: 32 BIT	0100 0000 0000 0000 1010 0000 0000 0100	40 00 A0 04
LITTLE ENDIAN	1111 1110 1110 1101 1011 1110 1010 1101	FE ED BE AD
00: 16 BIT	1010 0000 0000 0100 0100 0000 0000 0000	A0 04 40 00
LITTLE ENDIAN	1011 1110 1010 1101 1111 1110 1110 1101	BE AD FE ED

Initialisation of the CYW43439

This is long and complex.

First, there are configuration parameters for the chip and the SPI protocol.

Then the WLAN firmware must be loaded, then NVRAM data, then the Bluetooth firmware.

Only at that point is the CYW ready for HCI communication.

In these examples HIGH_SPEED SPI is not used, which makes the examples easier to understand.

Chip configuration

Firsly, the CYW43439 needs to be powered up – via GPIO pin 23.

The clock and data output pins are brought to 0, then power taken to 0, a 20ms wait, then taken to 1 and a further 250ms wait.

```
cs  = Pin(25, Pin.OUT)
clk = Pin(29, Pin.OUT)
pwr = Pin(23, Pin.OUT)

def power_on():
    clk.value(0)
    data_pin=Pin(24, Pin.OUT)
    data_pin.value(0)
    pwr.value(0)
    sleep_ms(20)
    pwr.value(1)
    sleep_ms(250)
```

When the chip is first powered up, the first read has 4 bits of 0 added to the start of the retrieved word. This can be avoided by a dummy write, which clears these 4 bits.

The code just sends 1 byte of 0 with no read, but this works to clear the buffer.

```
read = spi_transfer(b'\x00', 1, 0)
```

The configuration register (x0000) then needs to be set correctly.

In most other examples of chip configuration, a test is now performed to read a test register.

It is better to avoid any reads until the right HIGH_SPEED setting set as this will affect how the SPI transfer fuction is coded. HIGH_SPEED is the default setting.

In this case HIGH_SPEED is not used so this needs to be set before any reads.

```
SPI_FUNC                = 0
BACK_FUNC               = 1

WORD_LENGTH_32          = 0x0000_0001
BIG_ENDIAN              = 0x0000_0002
HIGH_SPEED              = 0x0000_0010
INT_POLARITY_HIGH       = 0x0000_0020
WAKE_UP                 = 0x0000_0080
STATUS_ENABLE           = 0x0001_0000
INTR_WITH_STATUS        = 0x0002_0000
```

```
CONFIG_REG                = 0x00

config = WORD_LENGTH_32 | BIG_ENDIAN | INT_POLARITY_HIGH | WAKE_UP |
INTR_WITH_STATUS

cyw_write_reg_u32_swap(SPI_FUNC, CONFIG_REG, config)
sleep_ms(500)
```

The function `cyw_write_reg_u32_swap` is used to swap the command bytes around because the defauly start-up condition requires this.

The configuration sets a 32 bit word length, big endian (but see later) and also not to use HIGH_SPEED SPI.

According to the specification, page 22, the default is 16 bit words, little endian and HIGH_SPEED mode SPI.

Now the test register (x0014) can be read and compared with the predicted value of FE ED BE AD.

```
FEEDBEAD_REG              = 0x14

read = cyw_read_reg_u32(SPI_FUNC, FEEDBEAD_REG)
```

Each read from the backplane can have a string of 00 bytes added before the actual data, to allow for the timing delay of processing the request. A value of 4 bytes is used, but many implementations including the PICO SDK use 16 bytes.

```
BACKPLANE_PAD_VALUE       = 0x04
BACKPLANE_PAD_REG         = 0x1d

cyw_write_reg_u8(SPI_FUNC, BACKPLANE_PAD_REG, BACKPLANE_PAD_VALUE)
```

The interrupt bits are cleared

```
SPI_INT_REG               = 0x04

DATA_UNAVAILABLE          = 0x0001
COMMAND_ERROR             = 0x0008
DATA_ERROR                = 0x0010
F1_OVERFLOW               = 0x0080

config = DATA_UNAVAILABLE | COMMAND_ERROR | DATA_ERROR | F1_OVERFLOW
cyw_write_reg_u16(SPI_FUNC, SPI_INT_REG, config)
```

Then the required interrupts are set

```
F2_F3_FIFO_RD_UNDERFLOW   = 0x0002
F2_F3_FIFO_WR_OVERFLOW    = 0x0004
COMMAND_ERROR             = 0x0008
DATA_ERROR                = 0x0010
F2_PACKET_AVAILABLE       = 0x0020
F1_OVERFLOW               = 0x0080
F1_INTR                   = 0x2000
```

```
config = F2_F3_FIFO_RD_UNDERFLOW | F2_F3_FIFO_WR_OVERFLOW | COMMAND_ERROR |
DATA_ERROR | F2_PACKET_AVAILABLE | F1_OVERFLOW | F1_INTR

cyw_write_reg_u16(SPI_FUNC, SPI_INT_REG, config)
```

The the ALP clock is enabled. This is the first of the backplane registers to be set, although the 0x1_0000 range addresses don't require the backplane window to be set so the regular cyw_write_reg_uX functions can be used.

```
SDIO_CHIP_CLOCK_CSR      = 0x1_000e
SBSDIO_ALP_AVAIL_REQ     = 0x08

cyw_write_reg_u8(BACK_FUNC, SDIO_CHIP_CLOCK_CSR, SBSDIO_ALP_AVAIL_REQ)
```

Set the bluetooth watermark to 0x10

```
SDIO_FUNCTION2_WATERMARK = 0x1_0008

cyw_write_reg_u8(BACK_FUNC, SDIO_FUNCTION2_WATERMARK, 0x10)
read = cyw_read_reg_u8(BACK_FUNC, SDIO_FUNCTION2_WATERMARK)
if (read != 0x10):
    print("**** Set bluetooth watermark failed")
```

Check the ALP clock is started

```
SDIO_CHIP_CLOCK_CSR      = 0x1_000e
SBSDIO_ALP_AVAIL         = 0x40

read = cyw_read_reg_u8(BACK_FUNC, SDIO_CHIP_CLOCK_CSR)
if (read & SBSDIO_ALP_AVAIL == 0):
    print("**** Check ALP available failed")
```

Clear the request for the ALP clock

```
cyw_write_reg_u8(BACK_FUNC, SDIO_CHIP_CLOCK_CSR, 0)
```

Check both device cores are started

```
CORE_WLAN    = 1
CORE_SOCSRAM = 2

check_core(CORE_WLAN)
check_core(CORE_SOCSRAM)
```

The reset the SOCSRAM core

```
reset_core(CORE_SOCSRAM)
```

Disable memory remap for SRAM_3

```
SOCSRAM_BASE_ADDRESS     = 0x1800_4000

SOCSRAM_BANKX_INDEX      = SOCSRAM_BASE_ADDRESS + 0x10
```

```
SOCSRAM_BANKX_PDA        = SOCSRAM_BASE_ADDRESS + 0x44

cyw_write_backplane_reg_u32(SOCSRAM_BANKX_INDEX, 0x3)
cyw_write_backplane_reg_u32(SOCSRAM_BANKX_PDA, 0)
```

Read the chip number (should be 43439)

```
CHIPCOMMON_BASE_ADDRESS  = 0x1800_0000

read = cyw_read_backplane_reg_u16(CHIPCOMMON_BASE_ADDRESS)
print("---- Chip id:", read)
```

Write the WLAN firmware

```
write_firmware()
```

Write the NVRAM data

```
write_nvram()
```

Reset the WLAN core and then check both cores are running

```
reset_core(CORE_WLAN)

check_core_up(CORE_WLAN)
check_core_up(CORE_SOCSRAM)
```

Once the WLAN core is programmed and running, the HT clock can start

```
SDIO_CHIP_CLOCK_CSR      = 0x1_000e
SBSDIO_HT_AVAIL          = 0x80

read = cyw_read_reg_u8(BACK_FUNC, SDIO_CHIP_CLOCK_CSR)
while (read & SBSDIO_HT_AVAIL) == 0:
    print(".... Failed HT clock")
    sleep_ms(200)
    read = cyw_read_reg_u8(BACK_FUNC, SDIO_CHIP_CLOCK_CSR)
```

And set the interrupt masks

```
SDIO_BASE_ADDRESS        = 0x1800_2000
SDIO_INT_HOST_MASK       = SDIO_BASE_ADDRESS + 0x24

I_HMB_SW_MASK            = 0x0000_00f0
I_HMB_FC_CHANGE          = 0x20

cyw_write_backplane_reg_u32(SDIO_INT_HOST_MASK, I_HMB_SW_MASK)
cyw_write_backplane_reg_u32(SDIO_INT_HOST_MASK, I_HMB_FC_CHANGE)
```

Set the bluetooth watermark to a new value

```
SDIO_FUNCTION2_WATERMARK = 0x1_0008
```

```
SPI_F2_WATERMARK          = 0x20

cyw_write_reg_u8(BACK_FUNC, SDIO_FUNCTION2_WATERMARK, SPI_F2_WATERMARK)
```

Then wait for F2 to be ready

```
SPI_STATUS_REG            = 0x08
STATUS_F2_RX_READY        = 0x0000_0020

read = cyw_read_reg_u8(SPI_FUNC, SPI_STATUS_REG)
while (read & STATUS_F2_RX_READY) == 0:
    print(".... Failed F2 ready")
    sleep_ms(200)
        read = cyw_read_reg_u8(SPI_FUNC, SPI_STATUS_REG)
```

Change the pull-up settings

```
SDIO_PULL_UP              = 0x1_000f

cyw_write_reg_u8(BACK_FUNC, SDIO_PULL_UP, 0)
read = cyw_read_reg_u8(BACK_FUNC, SDIO_PULL_UP)
```

Clear the data unavailable flag

```
SPI_INT_REG               = 0x04
DATA_UNAVAILABLE          = 0x0001

status = cyw_read_reg_u16(SPI_FUNC, SPI_INT_REG)
if status & DATA_UNAVAILABLE:
    cyw_write_reg_u16(SPI_FUNC, SPI_INT_REG, status)
```

Sent bluetooth power up wake, then load the bluetooth firmware

```
BTFW_MEM_OFFSET           = 0x1900_0000
BT2WLAN_PWRUP_ADDR        = 0x0064_0894
BT2WLAN_PWRUP_WAKE        = 0x03

cyw_write_backplane_reg_u32(BTFW_MEM_OFFSET + BT2WLAN_PWRUP_ADDR,
BT2WLAN_PWRUP_WAKE);
```

Write BT firmware

```
write_bt_firmware()
```

Set the host ready

```
HOST_CONTROL_REG          = 0x1800_0d6c
SW_READY                  = 0x0100_0000

def host_ready():
    val = cyw_read_backplane_reg_u32(HOST_CONTROL_REG)
    val |= SW_READY
    cyw_write_backplane_reg_u32(HOST_CONTROL_REG, val)
```

Wake up BT

```
HOST_CONTROL_REG          = 0x1800_0d6c
WAKE_BT                   = 0x0002_0000

def wake_bt():
    val = cyw_read_backplane_reg_u32(HOST_CONTROL_REG)
    new_val = val | WAKE_BT
    if new_val != val:
        cyw_write_backplane_reg_u32(HOST_CONTROL_REG, new_val)
```

Wait for BT to be ready

```
BT_CONTROL_REG            = 0x1800_0c7c
FW_READY                  = 0x0100_0000

def is_bt_ready():
    return cyw_read_backplane_reg_u32(BT_CONTROL_REG) & FW_READY

def wait_bt_ready():
    while not is_bt_ready():
        print("BT not ready yet")
        sleep_ms(500)
```