**CYW43439 Bluetooth Driver**

1. **Introduction**

The CYW4349 wifi and bluetooth chip is used in the Raspberry Pico Pi W. The C SDK includes the required drivers and protocol 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. This has been reverse engineered from various sources listed at the en

For clarity, in this document, hex numbers are shown with underscores (12\_34\_56\_78) and byte arrays without (12 34 56 78 in a big-endian array, 78 56 34 12 in a little-endian array)

1. **Overview**

All HCI information is written to and read from circular memory buffers in the CYW chip (one for write, a separate one for read).

The HCI packet has a three-byte length field added at the beginning, which makes it compatible with the SDIO SPI standard.

The SPI protocol has a 32 bit command field which specifies a write or read, the address to use and the length of the data.

For a write the command if followed by a set of 32 bit words to write to the memory (via MOSI).

For a read, the CYW will respond with a set of 32 bit words read from memory (via MISO).

|  |  |  |  |
| --- | --- | --- | --- |
| **Command word** | **Length** | **HCI event** | **HCI Data** |
| 40 00 A0 04 | 06 00 00 | 04 | 0e 04 01 03 0c 00 |

*Example SPI command to write HCI data*

When writing to the circular buffer for an HCI send, the buffer data is written, then the new buffer head address is written, then another address has its value toggled to indicate a write

When reading, a check is made to see if data is present, then the buffer pointers read, then the actual data.

1. **HCI interface**

The HCI layer data comprises the normal HCI information with a three-byte length field added at the start.

The length used is the number of bytes in the data in the packet, which excludes the 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*

1. **SPI interface**

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

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 is 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 Function two bits select which types of address are in use. For bluetooth only SPI BUS addresses and Backplane addresses are needed. The SPI BUS addresses are used to access the configuration registers, all other writes and reads are to the Backplane.

* 1. **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.

|  |  |  |  |
| --- | --- | --- | --- |
| **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*

* 1. **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*

1. **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 are in the table below.

|  |  |  |
| --- | --- | --- |
| **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*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Host to bluetooth  base + 0x0000 | | | | | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | ^ 0\_00 |  | ^0\_20  TAIL |  |  | ^ 0\_40  HEAD |  |  | f \_ff ^ |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | Bluetooth to host  base + 0x1000 | | | | | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | ^ 0\_00 |  |  | ^0\_30  TAIL |  |  |  | ^ 1\_60  HEAD | f\_ff ^ |  |
|  |  |  |  |  |  |  |  |  |  |  |

*Buffer diagram*

1. **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*

1. **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.

A diagram of a diagram

Description automatically generated

*Normal speed SPI timings*

A graph with lines and numbers

Description automatically generated with medium confidence

*High speed SPI timings*

1. **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 |

*Example memory order*

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 |

*Test register with different endian / word-size setting*

The command to read the test register is as below.

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

*Test register read command in hex*

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

*Test register read command in binary*

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 |

*Command and test register values in different endian and word sizes*

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

*Example of bits on SPI interface*

1. **Processing HCI transactions**
   1. **Sending HCI data**

First the packet must have the data length pre-pended, and then the buffer padded to be word aligned.

The buffer head and tail are read, and then the buffer written at the position of the send head.

Then the new send head is written, and the ‘data send’ address toggled to indicate to the CYW that new data is available.

|  |
| --- |
| base = self.wifi\_base  size = len(dat)  length\_bytes = u32\_to\_le\_bytes(size - 1)[0:3]  size += 3  adjusted\_size = (size + 3) & ~3  padding = bytes(adjusted\_size - size)    buf = length\_bytes + dat + padding  buf\_len = len(buf)    send\_head = cyw\_read\_backplane\_reg\_u32(base + SEND\_HEAD)  send\_tail = cyw\_read\_backplane\_reg\_u32(base + SEND\_TAIL)    cyw\_write\_backplane\_bytes(base + H2BT\_BUFFER + send\_head, buf, buf\_len)  cyw\_write\_backplane\_reg\_u32(base + SEND\_HEAD, send\_tail + buf\_len)  data\_send\_toggle() |

The way to toggle the CYW for new data is shown below, doing an exclusive-or on the HOST\_CONTROL\_REG

|  |
| --- |
| HOST\_CONTROL\_REG = 0x1800\_0d6c  DATA\_VALID = 0x0000\_0002  def data\_send\_toggle():  val = cyw\_read\_backplane\_reg\_u32(HOST\_CONTROL\_REG)  val ^= DATA\_VALID  cyw\_write\_backplane\_reg\_u32(HOST\_CONTROL\_REG, val) |

* 1. **Receiving HCI data**

The presence of new data can be checked by reading the SDIO\_INT\_STATUS and looking for the I\_HMB\_FC\_CHANGE bit.

|  |
| --- |
| SDIO\_BASE\_ADDRESS = 0x1800\_2000  SDIO\_INT\_STATUS = SDIO\_BASE\_ADDRESS + 0x20  I\_HMB\_FC\_CHANGE = 0x20  def readable(self):  read = cyw\_read\_backplane\_reg\_u32(SDIO\_INT\_STATUS);  if read & I\_HMB\_FC\_CHANGE != 0:  cyw\_write\_backplane\_reg\_u32(SDIO\_INT\_STATUS, read & I\_HMB\_FC\_CHANGE)  return read & I\_HMB\_FC\_CHANGE != 0 |

If data is present, then the buffer pointers for the read buffer are retrieved, then data read from the buffer, and finally the tail of the buffer incremented to the head position.

Currently this code assumes only one SPI HCI packet is in the buffer at one time.

|  |
| --- |
| base = self.wifi\_base  receive\_head = cyw\_read\_backplane\_reg\_u32(base + RECEIVE\_HEAD)  receive\_tail = cyw\_read\_backplane\_reg\_u32(base + RECEIVE\_TAIL)  dat = cyw\_read\_backplane\_bytes(base + BT2H\_BUFFER + receive\_tail, receive\_head - receive\_tail)  cyw\_write\_backplane\_reg\_u32(base + RECEIVE\_TAIL, receive\_head)  data\_send\_toggle()    leng = le\_bytes\_to\_u32(dat[0:3])  return dat[3:3 + leng + 1] |

1. **Initialisation of the CYW43439**

To start the initialisation, the SPI protocol and chip configuration parameters are set.

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 SPI protocol easier.

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) |

1. **Loading firmware**

There are two firmwares to load (WLAN and bluetooth) and two configuration files (NVRAM and CLM). It seems CLM is not necessary for Bluetooth so is not loaded in this code.

* 1. **WLAN**

The WLAN firmware is one binary file which is loaded to address 0000\_0000. This is done in 64 byte chunks which is the limit for the SPI interface.

This firmware is 231,077 bytes long and this is rounded to the next word boundary, making a file of 231,080 bytes.

* 1. **NVRAM**

This is loaded near the top of RAM (0008\_0000) and has a ‘magic’ word after it, at 0007\_fffc. The NVRAM is 743 bytes long, rounded to 744, so starts at 0007\_fffb.

It is loaded in chunks of 64 bytes.

* 1. **Bluetooth firmare**

The bluetooth firmware has a defined format and can be loaded to multiple offset addresses. The base address for bluetooth is 1900\_0000 so offsets are relative to that.

The data has a version string, then a number of records. Each record has an address, length and data.

|  |  |
| --- | --- |
| **Length (bytes)** | **Data** |
| Header |  |
| 1 | Number of bytes in the version string (v) |
| v | Version string |
| 1 | Number of records in the file |
| Each record |  |
| 1 | Length of record data (n) |
| 2 | Address (big endian, so high byte then low byte) |
| 1 | Record type |
| n | Data |

There are five types of record. Type 0 is a data record. Type 1 is at the end of all the data. The other three are address records. In the bluetooth firmware only the extended address type 4 is used.

Type 4 sets the highest word of the address, and the lowest word is in each data record.

|  |  |
| --- | --- |
| **Record type** | **Constant** |
| BTFW\_HEX\_LINE\_TYPE\_DATA | 0 |
| BTFW\_HEX\_LINE\_TYPE\_END\_OF\_DATA | 1 |
| BTFW\_HEX\_LINE\_TYPE\_EXTENDED\_SEGMENT\_ADDRESS | 2 |
| BTFW\_HEX\_LINE\_TYPE\_EXTENDED\_ADDRESS | 4 |
| BTFW\_HEX\_LINE\_TYPE\_ABSOLUTE\_32BIT\_ADDRESS | 5 |

Example data below showing the structure and content

|  |  |  |
| --- | --- | --- |
| **Data offset (hex)** | **Data (in hex)** | **Interpretation** |
| 00 | 46 | 70 bytes of data |
| 01 | 43 59 57 34 33 34 33 41 32 5F 30 30 31 2E 30 30 33 2E 30 31 36 2E 30 30 33 31 2E 30 30 30 30 5F 47 65 6E 65 72 69 63 5F 53 44 49 4F 5F 33 37 4D 48 7A 5F 77 6C 62 67 61 5F 42 55 5F 64 6C 5F 73 69 67 6E 65 64 00 | CYW4343A2\_001.003.016.0031.0000\_Generic\_  SDIO\_37MHz\_wlbga\_BU\_dl\_signed\0 |
| 47 | 21 | 33 records in the data |
|  |  |  |
| 48 | 02 | Length of 2 |
| 49 | 00 00 | Address of 0000 |
| 4B | 04 | Sets extended address |
| 4C | 00 21 | 0x0021\_0000 |
|  |  |  |
| 4E | 42 | Length of 66 |
| 4F | E0 00 | Address of e000 |
| 51 | 00 | Record type ‘data’ |
| 52 | 42 52 43 4D 63 66 67 53 00 00 00 00 32 00 00 00 01 01 04 18 92 00 00 00 03 06 AC 1F 12 A2 43 43 00 01 1C 00 F0 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FE 00 00 |  |
|  |  |  |
| 94 | FB | Length of 251 |
| 95 | F0 00 | Address of F000 |
| 97 | 00 | Record type ‘data’ |
| 98 | 42 52 43 4D 63 66 67 44 00 00 00 00 D2 0B 00 00 03 03 18 53 44 49 4F 20 33 37 5F 34 4D 20 77 6C 62 67 61 5F 42 55 20 52 50 49 00 16 03 02 00 00 02 01 90 01 08 01 32 00 01 00 00 00 01 00 00 00 00 00 32 00 FF 0F 00 00 62 08 00 00 70 00 64 00 80 00 00 00 80 00 00 00 AC 00 32 00 FF FF FF 01 00 00 2F 00 64 01 60 00 FF 00 00 00 00 00 00 00 68 01 60 00 FF 00 00 00 06 00 00 00 6C 01 60 00 FF 00 00 00 0C 00 00 00 70 01 60 00 FF 00 00 00 11 00 00 00 74 01 60 00 FF 00 00 00 16 00 00 00 78 01 60 00 FF 00 00 00 1A 00 00 00 7C 01 60 00 FF 00 00 00 1F 00 00 00 84 01 60 00 FF 00 00 00 23 00 00 00 04 03 0C 20 01 20 00 0F 14 1A 66 0A 15 00 00 05 03 15 00 1B 78 50 FF FF 3F 00 05 03 1F 64 B0 BD 0A 05 00 FF FF 07 BC 09 01 04 02 0F 00 00 22 03 02 01 00 F0 01 28 04 |  |
|  |  |  |
| 193 | FB | Lentgh 251 |
| 194 | F0 FB | Address of F0FB |
| 196 | 00 | Record type ‘data’ |
| 197 | 00 00 00 20 15 32 00 FF FF FA FF 01 01 06 00 28 15 32 00 00 00 7F FE 34 10 20 02 2C 09 64 00 0E 00 00 00 0E 00 00 00 F0 01 28 01 00 00 00 20 15 32 00 FF FF FA FF 05 01 06 06 28 15 32 00 00 00 7F FE 34 10 20 02 2C 09 64 00 0E 00 00 00 0E 00 00 00 F0 01 28 02 00 00 00 20 15 32 00 FF FF FA FF 01 01 06 00 28 15 32 00 00 00 7F FE 34 10 20 02 2C 09 64 00 0E 00 00 00 0E 00 00 00 F0 01 A0 01 03 00 00 00 10 15 32 00 00 00 00 00 F0 F0 F0 00 14 15 32 00 00 00 00 00 F0 F0 F0 00 18 15 32 00 00 00 00 00 F0 F0 F0 00 1C 15 32 00 00 00 00 00 4C 4C 00 00 20 15 32 00 FF FF FA FF 05 01 06 06 24 15 32 00 00 00 00 00 00 00 00 00 28 15 32 00 00 00 7F FE 34 10 20 02 34 15 32 00 00 00 00 00 F0 00 00 00 38 15 32 00 00 00 00 00 00 00 00 00 50 15 32 00 00 00 00 00 CA 06 |  |
|  |  |  |
| 292 | FB | Length 251 |
| 293 | F1 F6 | Address of F1 F6 |
| 295 | 00 | Record type ‘data’ |

The records are mostly 251 bytes long, with some shorter ones at the end of a block of addresses.

The code will read each section of data and then send it to the CYW in chunks of 64 bytes. As long as data read is in consecutive addresses, then this continues. Once the address continuity is broken then a final chunk is written for any remaining data, with the actual remaining length.

In this way all writes are word aligned, and only the final write in any address range is not of 64 bytes.

No alignment checking in done – the firmware provided has each address word-aligned in the actual data already.

* 1. **Firmware write addresses**

|  |  |  |  |
| --- | --- | --- | --- |
| **Firmware** | **Start** | **End** | **Length (word aligned)** |
| WLAN | 0000\_0000 | 0003\_86a7 | 231077 (231080) |
|  |  |  |  |
| NVRAM | 0007\_fd14 | 0007\_fffb | 743 (744) |
| NVRAM Magic | 0007\_fffc | 0007\_ffff | 4 |
|  |  |  |  |
| Bluetooth | 1921\_e000 | 1921 e041 |  |
|  | 1921\_f000 | 1921\_fc21 |  |
|  | 190d\_0200 | 190d\_0cd3 |  |
|  | 1920\_38d4 | 1920\_38d7 |  |
|  | 1920\_4154 | 1920\_4167 |  |

1. **Memory map**

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | **Address range** |
| SPI | gSPI registers | 00 – 1f |
| SPI | CONFIGURATION REGISTER | 00 |
| SPI | STATUS ENABLE REGISTER | 02 |
| SPI | INTERRUPT REGISTER | 04 |
| SPI | INTERRUPT REGISTER | 05 |
| SPI | INTERRUPT ENABLE REGISTER | 06 |
| SPI | STATUS REGISTER | 08 |
| SPI | F1 INFO REGISTER | 0c |
| SPI | F2 INFO REGISTER | 0e |
| SPI | TEST READ ONLY REGISTER | 14 |
| SPI | TEST R/W REGISTER | 18 |
| SPI | RESPONSE DELAY REGISTERS | 1c |
|  |  |  |
| Backplane | WIFI firmware | 0000\_0000 |
| Backplane | START\_NVRAM | 0007\_fd14 |
| Backplane | TOP\_OF\_RAM\_MAGIC | 0007\_7ffc |
| Backplane | TOP\_OF\_RAM | 0008\_0000 |
|  |  |  |
| 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\_STATUS | 1800\_2020 |
| Backplane | SDIO\_INT\_HOST\_MASK | 1800\_2024 |
|  |  |  |
| Backplane | WLAN\_ARMCM3\_BASE | 1800\_3000 |
| Backplane | AI\_IOCTRL | 1810\_3408 |
| Backplane | AI\_RESETCTRL | 1810\_3800 |
|  |  |  |
| Backplane | SCOSRAM\_BASE | 1800\_4000 |
| Backplane | SOCSRAM\_BANKX\_INDEX | 1800\_4010 |
| Backplane | SOCSRAM\_BANKX\_PDA | 1800\_4044 |
| Backplane | AI\_IOCTRL | 1810\_4408 |
| Backplane | AI\_RESETCTRL | 1810\_4800 |
|  |  |  |
| Backplane | BTFW\_MEM\_OFFSET | 1900\_0000 |
| Backplane | BT2WLAN\_PWRUP\_ADDR | 1964\_0894 |

*Memory map*

1. **References**

The CYW43439 datasheet is useful but doe not describe most of the programming of the chip. This information has been gleaned from a few other sources and github.

**Datasheet:**

<https://www.infineon.com/dgdl/Infineon-CYW43439-DataSheet-v05_00-EN.pdf?fileId=8ac78c8c8929aa4d01893ee30e391f7a>

**Pico SDK C:**

<https://github.com/raspberrypi/pico-sdk/tree/master/src/rp2_common/pico_cyw43_driver>

**Rust driver:**

<https://github.com/embassy-rs/embassy/tree/main/cyw43>

**Go driver:**

<https://github.com/soypat/cyw43439>

**Document on reverse engineering the SPI protocol:**

<https://iosoft.blog/2022/12/06/picowi/>

**Information about the firmware:**

<https://github.com/georgerobotics/cyw43-driver/tree/main/firmware>