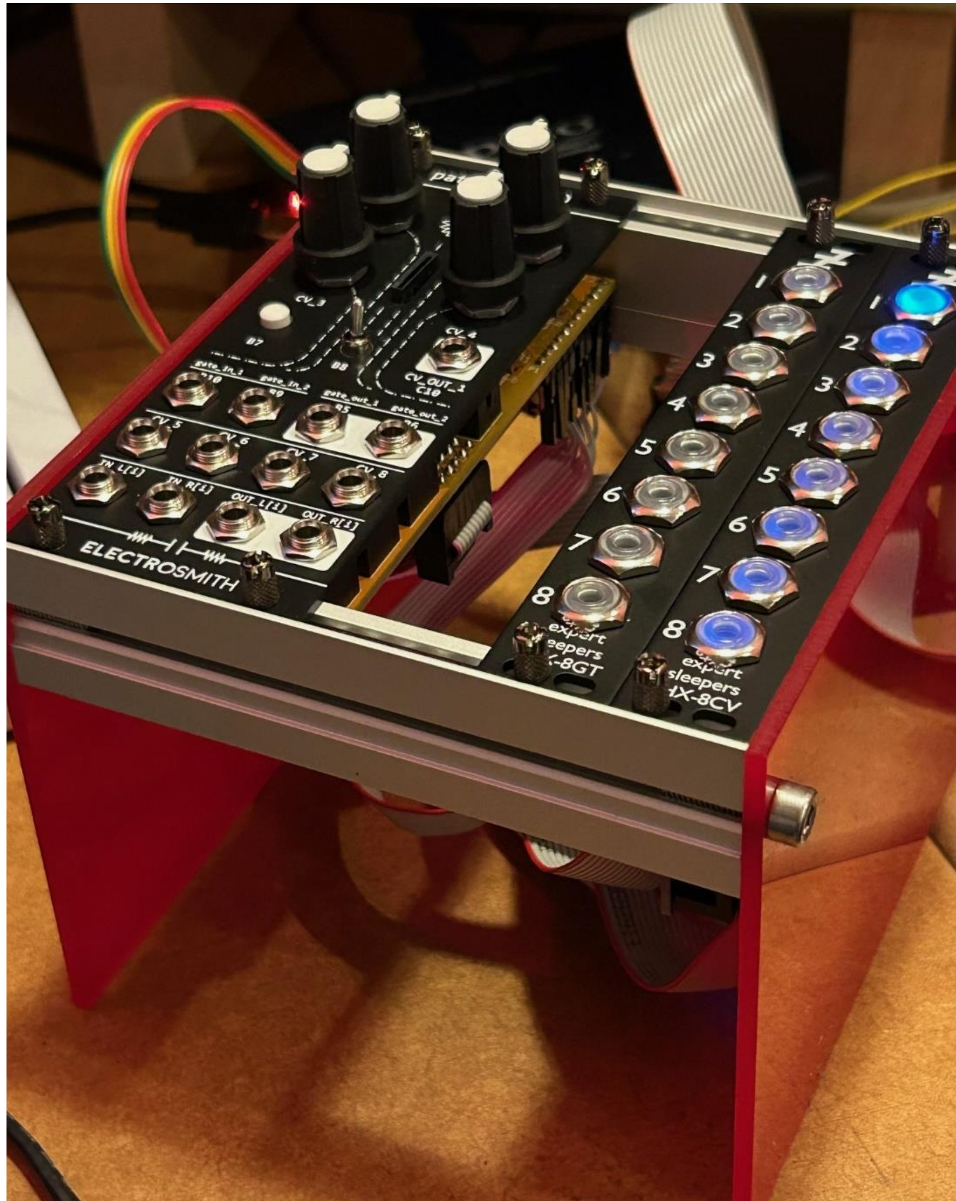
Connecting FHX-8CV to Patch.Init

RG - 19th Nov 2023 Draft 1 v 2

Summary

Patch.Init and its PatchSM are cool but we may need more gate and CV outputs. Expert Sleepers makes really nice expanders for the FH2 Fader Host. These expanders give 8x gate outs (FHX-8GT) and 8x CV outs (FHX-8CV) and we can use them with Patch.Init by making a custom cable and writing some C++ code.



Why use these expander modules versus building our own expanders from scratch? Some reasons include that FHX expanders will go straight into a modular rack, 8-way outputs, very cost-effective for high quality, and they output modular voltage levels (bipolar for the FHX-8CV) from standard 3.5mm jacks. Plus it's relatively easy to use them.

Recapping the SPI protocol

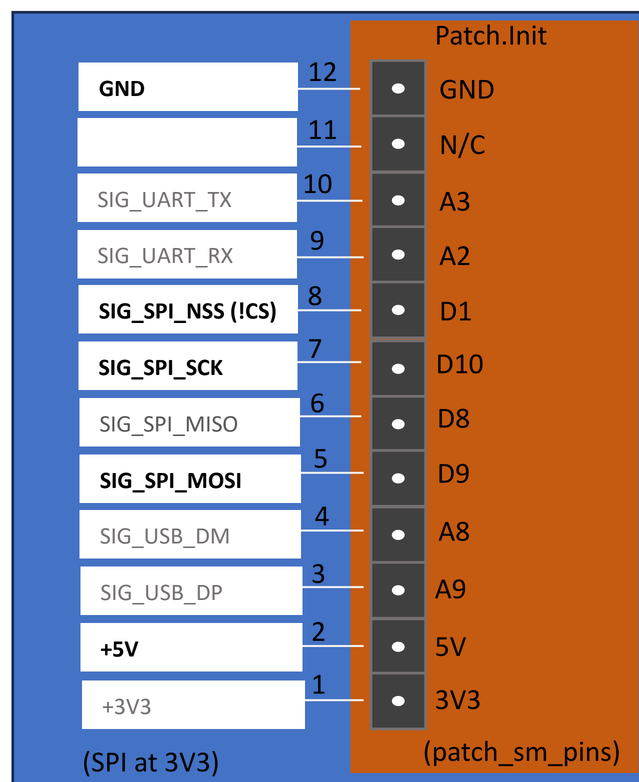
SPI connections for Main and Subnode:

SPI has four logic signals (which go by alternative namings):

- SCLK : Serial Clock (clock signal from main)
- MOSI : Main Out Sub In (data output from main)
- MISO : Main In Sub Out (data output from sub)
- CS : Chip Select (a.k.a. NSS, active low signal from main to address subs and initiate transmission)

Daisy Patch Init

SPI pins / voltages on Daisy Patch SM:



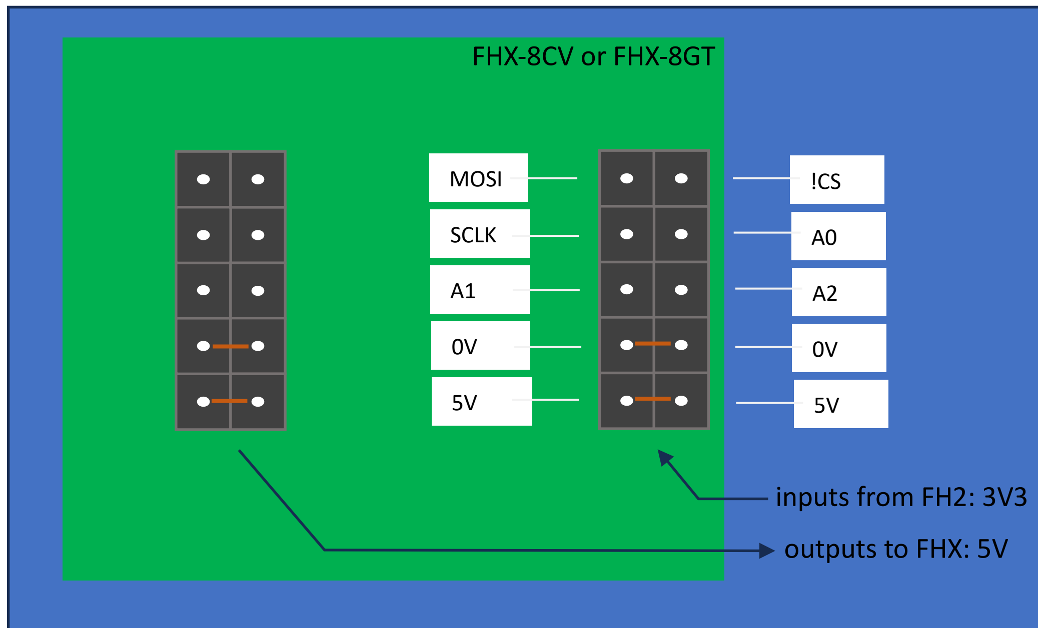
The Patch.Init header is marked P1-M12 on the Patch.Init schematic. The download link for the Electrosmith design files is on this page: https://www.electro-smith.com/daisy/patchinit

- D8 ADC_12 PC2 GPIO SPI2_MISO (0V / 3V3)
- D9 ADC_11 PC3 GPIO SPI2_MOSI (0V / 3V3)
- D1 SPI_CS (0V / 3V3)
- D10 SPI_SCK (0V / 3V3)

This data per Hardware info: ES_Patch_SM_datasheet_v1.0.5.pdf (squarespace.com)

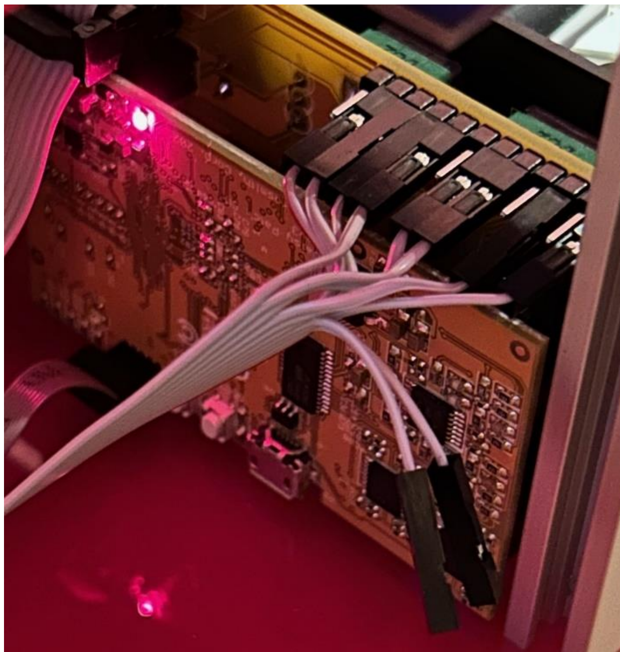
FHX Expanders

SPI pins / voltages on FHX-8CV and FHX 8GT:



The pins labelled A0, A1, A2 are addressing pins, used by the Expert Sleepers FH2 to address multiple different FHX-8CV and/or FHX-8GT expanders. The FH2 uses these pins to address up to 7 FHX-8CV expanders and 4 FHX-8GT expanders.

To hook up the SPI we need to make a custom cable or buy some pre-crimped duPont jump wires.



Summary of Expert Sleepers Expanders

The Expert Sleepers FH2 (Fader Host) supports up to four FHX-8GT gate expanders. Each FHX-8GT provides 16 gate outputs with 8 of the gates brought out to jacks on its front panel and the other 8 gates accessible if an ESX-8GT expander is plugged into the port on the back of the FHX-8GT.

Up to 4 FHX-8GT gate expanders can be daisy-chained via the GT1 and GT2 10-pin connectors on the rear panel. Each FHX-8GT can have a ESX-8GT expander attached to surface its other 8 outputs, for a maximum of 64 gate outputs, all on the same SPI bus. Each FHX-8GT has an addressing jumper on the rear with links that can be positioned to set that expander's address.

Up to 7 FHX-8CV expanders can also be daisy-chained on the FH2 SPI bus for 56 CV expander outputs. Most likely we can daisy-chain 8 FHX-8CV expanders to Patch.Init for 64 CV outs if needed.

Overview of Expert Sleepers SPI Protocol

Expert Sleepers uses a 3-wire SPI with MOSI, SCLK and !CS. MISO is not used. Packets on the SPI bus are 32 bits/4 bytes. Clock polarity is low: data is clocked on the 2nd clock edge which is the falling clock edge. Chip select (!CS) is active low. The bit ordering is MSB first.

Expert Sleepers has augmented the SPI bus with three address lines. These are three GPIO lines which are used to signal one of 8 possible three-bit address values.

The Expert Sleepers FH2 outputs 3V3 signals but the FHX-8GT and FH2-8CV expanders have line drivers on board which send the SPI signals onward at 5V0. Thus all the expander inputs can safely handle 3V3 or 5V0.

FHX-8GT SPI Protocol

Gate Signals and SPI Data Packets

Packets sent to the FHX-8GT comprise 4 bytes or 32 bits.

Byte 3	Byte 2	Byte 1	Byte 0

The most significant byte is sent first and contains a packet identifier that denotes the packet as a packet for the FHX-8GT, then there are two bytes which are interpreted as the bit values for the 16 gate outputs (note that 8 of the 16 bits won't be accessible unless an ESX-8GT is plugged into the FHX-8GT.) The last byte's purpose remains unknown and might be padding as it seems to be always set to 0x00.

FHX-8GT Packet ID (0x04)	8 gate bits 16 thru 9 set/unset for ESX-8GT	Gate bits 8 thru 1 set/unset for FHX-8GT	Padding bits (0x00)

Example 1. To set the FHX-8GT output gates on channel 7 and channel 1 high the bit pattern in byte 1 would be 0100 0001 which is 0x41. The SPI packet would then be:

0x04	0x00	0x41	0x00

Example 2. To reset all FHX-8GT output gates to 0V (low) the bit pattern in bytes 2 and 1 would be 0000 0000 which is 0x00. The SPI packet would then be:

0x04	0x00	0x00	0x00

Addressing the FHX-8GT Modules

Up to four FHX-8GT expanders modules can be addressed. The selection is made using three GPIO signals, A2, A1, A0. These signals cannot be left floating. A2 must be set to 0V while addressing FHX-8GT expanders.

A2	A1	A0	FHX-8GT addr
0	0	0	0
0	0	3V3	1
0	3V3	0	2
0	3V3	3V3	3
N/A	N/A	N/A	N/A
N/A	N/A	N/A	N/A
N/A	N/A	N/A	N/A
N/A	N/A	N/A	N/A

The appropriate expander selection jumpers for each FHX-8GT expander must have been pre-configured on its back panel using jumper pins as specified by Expert Sleepers. See https://expert-sleepers.co.uk/fhx8gtusermanual.html. Then GPIO pins must be set to address the appropriate expander prior to sending the SPI packet data.

FHX-8GT Coding for Patch.Init in C++

Comments on the Dev Environment

To interface the Electrosmith Patch-SM to the FHX-8GT the dev environment needs to be set up as per Electrosmith guidance.

See: https://github.com/electro-smith/DaisyWiki/wiki/1.-Setting-Up-Your-Development-Environment

When flashing firmware to the Patch.Init, and given all the cabling needed, it's not really practical to press the 'boot' button sequence on the Patch-SM, so it's far better to use an ST-Link adapter such as the ST-Link V3 or one of the ST 'Discovery' boards. This would be connected to the Patch-SM JTAG connector (N.B. some Patch-SM boards don't have this fitted and it's quite tricky to solder).

Test the environment by running the Patch-SM blink example.

Setting Up a New Project

Use the Python helper script from Electrosmith. This will configure the shell of an empty project.

See: https://github.com/electro-smith/DaisyWiki/wiki/How-To-Create-a-New-Project

C++ Code Elements

To interface Patch.Init to the FHX-8GT we need to control three SPI lines and three GPIO lines. The notes here give the most key aspects of getting this working.

We are working with the Patch-SM (not Patch) and we are working with the Daisy library.

```
#include "daisy_patch_sm.h"
#include "daisysp.h"
#include <stdbool.h>
```

Instantiate a Patch-SM object we can refer to as ‘hw’

```
/** Hardware object for the patch_sm */
DaisyPatchSM hw;
```

We need three GPIO pins. In theory this could be done with the new daisy GPIO definitions but these seem broken in the Patch-SM implementation so we use old-style dsy_gpio definitions:

```
//Create dsy_gpio objects allowing use of three pins on the Patch-SM port A
dsy_gpio patch_A3; //will use for FHX A0
dsy_gpio patch_A8; //will use for FHX A1
dsy_gpio patch_A9; //will use for FHX A2
```

In the ‘main’ function we need the following:

```
    // Initialize the patch_sm hardware object
    hw.Init();
```

```
    // configure the GPIO pins and set at logic 0
    patch_A3.mode = DSY_GPIO_MODE_OUTPUT_PP; //for FHX A0
    patch_A3.pull = DSY_GPIO_NOPULL;
    patch_A3.pin = hw.A3;
    dsy_gpio_init(&patch_A3);
    dsy_gpio_write(&patch_A3, 0);

    patch_A8.mode = DSY_GPIO_MODE_OUTPUT_PP; //for FHX A1
    patch_A8.pull = DSY_GPIO_NOPULL;
    patch_A8.pin = hw.A8;
    dsy_gpio_init(&patch_A8);
    dsy_gpio_write(&patch_A8, 0);

    patch_A9.mode = DSY_GPIO_MODE_OUTPUT_PP; //for FHX A2
    patch_A9.pull = DSY_GPIO_NOPULL;
    patch_A9.pin = hw.A9;
    dsy_gpio_init(&patch_A9);
    dsy_gpio_write(&patch_A9, 0);
```

```
    // Handle we'll use to interact with SPI
    SpiHandle spi_handle;

    // Structure to configure the SPI handle
    SpiHandle::Config spi_conf;

    spi_conf.mode = SpiHandle::Config::Mode::MASTER; // we're in charge

    spi_conf.periph = SpiHandle::Config::Peripheral::SPI_2; // Use SPI_2
```

```
    // Pins to use. These must be available on the selected peripheral
    spi_conf.pin_config.mosi = DaisyPatchSM::D9; // Use D9 as MOSI
    spi_conf.pin_config.miso = Pin(); // We won't need this
    spi_conf.pin_config.sclk = DaisyPatchSM::D10; // Use pin D10 as SCLK
    spi_conf.pin_config.nss = DaisyPatchSM::D1; // use D1 as NSS

    // data will flow from main to sub over just the MOSI line
    spi_conf.direction = SpiHandle::Config::Direction::TWO_LINES_TX_ONLY;

    // Main will output on the NSS line
    spi_conf.nss = SpiHandle::Config::NSS::HARD_OUTPUT;

    //set some stuff for the SPI mode: ONE_EDGE is when we transition to the
clock polarity. TWO_EDGE is when we transition away from the clock polarity
    spi_conf.clock_polarity = SpiHandle::Config::ClockPolarity::LOW;
    spi_conf.clock_phase = SpiHandle::Config::ClockPhase::TWO_EDGE;

    //Number of bits to tx. Defaults to 8. Must be in the range [4, 32].
    spi_conf.datasize = 32;

    //The clock rate is 25MHz. With a prescaler of 4 the final clock rate is
25MHz / 4 = 6.25MHz (Default SpiHandle::Config::BaudPrescaler::PS_8)
    //SpiHandle::Config::BaudPrescaler::PS_8

    // Initialize the SPI Handle
    spi_handle.Init(spi_conf);
```

Now let’s write some data to the FHX-8GT, first setting the address bits on the GPIO pins

```
    //set up the GPIO pins to  addresss the FHX with its jumpers set as 0/1
    dsy_gpio_write(&patch_A3, 0);
    dsy_gpio_write(&patch_A8, 0);
    dsy_gpio_write(&patch_A9, 0);
```

Then writing SPI packets

```

        // Write some data to the FHX-8GT (Blocking SPI transmit these 4
bytes, buffer[3] sent first and buffer[0] is last)
        my_buffer[3] = 0x04;
        my_buffer[2] = 0x00;
        my_buffer[1] = 0x01; //01 = FHX-8GT ch1 gate on
        my_buffer[0] = 0x00;
        spi_handle.BlockingTransmit(my_buffer, 4);
        System::Delay(500);

        my_buffer[3] = 0x04;
        my_buffer[2] = 0x00;
```

```
    my_buffer[1] = 0x00; //FHX-8GT all gate channels off
    my_buffer[0] = 0x00;
    spi_handle.BlockingTransmit(my_buffer, 4);
```

FHX-8CV SPI Protocol

Two types of SPI data are sent to the FHX-8CV. Config data which sets the voltage polarity of the output signals via an offset, and CV data for routing through to the panel jacks.

Packets sent to the FHX-8CV comprise 4 bytes or 32 bits.

Byte 3	Byte 2	Byte 1	Byte 0

The most significant byte is sent first and contains a packet identifier that denotes the packet as a packet for the FHX-8CV. The packet ID is how Expert Sleepers can accommodate multiple FHX-8GT and FHX-8CV expanders on the same SPI bus – Expert Sleepers SPI packets have an ID targeting them to one type of module or the other. The three addressing GPIO lines are there to select which of multiple modules of the same type is being addressed and the packet ID specifies the type of expander the packet is intended for.

Two different Packet IDs are used to denote packets for the FHX-8CV depending whether the packet contains voltage data or offset data. The offset is what allows CV outs to be unipolar or bipolar.

FHX-8CV Volts Packet ID (0x03)	4 bits channel # and 4 MS bits voltage	8 bits voltage	4 LS bits voltage and 4 bits 0x0

FHX-8CV Offset Packet ID (0x14)	0x00	8 flags signal uni or bi-polar for each channel	0x00

Part 1 – Sending Config Data to the FHX-8CV

The way Expert Sleepers sends a config to the FHX-8CV is to first send each of the 8 channels of the FHX-8CV the code for the midpoint voltage of the desired voltage range (see table below). Then immediately after that, the config packet is sent to set the outputs as unipolar or bipolar.

The voltage ranges used by Expert Sleepers are:

Voltage Range	Voltage Bytes for Min Voltage	Voltage Bytes for Mid Range	Voltage Bytes for Max Range
0V to 1V	0x0000	0x0C96	0x1999
0V to 5V	0x0000	0x3EEE	0x7FFF
0V to 8V	0x0000	0x64AF	0xCCCC
0V to 10V	0x0000	0x7DDC	0xFFFF
-5V to +5V	0x0000	0x7DDC	0xFFFF

Potentially a reason why Expert Sleepers sends the midpoint voltage to each channel ahead of sending the config could be to prevent unexpected voltages being outputted upon changing the config. (point to resolve: why 0x7DDC rather than 0x7FFF etc?)

To implement the Expert Sleepers protocol we must send multiple 32-bit packets per FHX-8CV, setting the midpoint voltage per channel for each individual expander output, then one more packet per expander which incudes a single bit per channel flagging whether the output is unipolar or bipolar. Before sending any SPI packets the GPIO address pins must be set (see part 4 below).

The voltage data is sent under the packet ID 0x03 together with a 4-bit binary code of the associated channel. The SPI packet with the config bits is then sent with 8 channel bits that flag whether each channel is bipolar or unipolar. I.e. nine 32-bit packets are sent.

Example. To set all 8 channels of the FHX-8CV to +/- 5V, nine packets are sent using the structure:

FHX-8CV Volts Packet ID (0x03)	4 bits channel # and 4 MS bits voltage	8 bits voltage	4 LS bits voltage and 4 bits 0x0

The GPIO pins are set to address the expander and packet content would be sent as below. The channel numbers below are highlighted yellow for clarity and are followed by the voltage bytes.

Channel 1 (0x0-) voltage midpoint set 5v (0x7DDC)

0x03	0x07	0xDD	0xC0

Channel 2 (0x2-) voltage midpoint set 5v (0x7DDC)

0x03	0x27	0xDD	0xC0

Channel 3 (0x4-) voltage midpoint set 5v (0x7DDC)

0x03	0x47	0xDD	0xC0

Channel 4 (0x6-) voltage midpoint set 5v (0x7DDC)

0x03	0x67	0xDD	0xC0

Channel 5 (0x7-) voltage midpoint set 5v (0x7DDC)

0x03	0x77	0xDD	0xC0

Channel 6 (0x5-) voltage midpoint set 5v (0x7DDC)

0x03	0x57	0xDD	0xC0

Channel 7 (0x3-) voltage midpoint set 5v (0x7DDC)

0x03	0x37	0xDD	0xC0

Channel 8 (0x1-) voltage midpoint set 5v (0x7DDC)

0x03	0x17	0xDD	0xC0

And finally, with the GPIO pins still addressing the FHX-8GT, the config packet is sent, structured as:

FHX-8CV Offset Packet ID (0x14)	0x00	8 flags signal uni or bi-polar voltage outs	0x00

Offset bits are '1' for bipolar, so to set for +/- 5V on all channels, the config byte is 0xFF, and the SPI packet is:

0x14	0x00	0xFF	0x00

The Expert Sleepers' protocol for the config packet carries the 8 flags in the third byte (byte 1) where the flag is set as 1 for a bipolar voltage and 0 for unipolar voltage. The order of flag bits is the same as the order of channels: bit 0 of the byte is Ch1 and bit 7 of the byte is Ch8. Config data is maintained while the module is powered up but lost on power-down, so after power-up, a config must be sent before the FHX-8CV is used.

Part 2 – Sending CV Data to the FHX-8CV

Sending CV values to the FHX-8CV is accomplished by sending a 32-bit packet with the channel number and the CV value. Prior to sending this packet the address selection on GPIO pins must have been set. The packet structure is:

FHX-8CV Volts Packet ID (0x03)	4 bits channel # and 4 MS bits voltage	8 bits voltage	4 LS bits voltage and 4 bits 0x0

Example. Send channel 6 a CV value of 0x1234

0x03	0x51	0x23	0x40

In the above example the packet ID 0x03 denotes this as a CV packet for the FHX-8CV, the channel number 0x5 specified channel 6 (see below for further information) and the CV value 0x1234 is specified. The last four bits of the packet are set to zero and pad the packet to 32 bits. The actual voltage appearing on the FHX-8CV panel jack depends both on the scale and offset config previously set on that expander channel and upon the 2-byte value sent to the channel.

Tables below give some examples of the voltage values measured on a CV channel for various configs and voltage bytes sent to the channel. A low accuracy voltmeter was used.

Channel Offset 0x00 (all channels unipolar)		
Bytes sent to Channel	***Expected Voltage***	***Measured Voltage***
0x0000	0V	0.00V
0x0CCC	0.5V	0.48V
0x3EEE	2.5V	2.43V
0x6666	4V	3.89V
0x7FFF	5V	4.91V
0xFFFF	10V	9.88V

If software is calculating CV out values which need to be converted to hexadecimal bytes for the SPI, the we can first calculate constant scaling factor x, where:

P is an internal value, that takes maximum value P_{max}

Q is an outputted voltage, that takes maximum value Q_{max} (say 1V or 5V max)

65535.0 is a floating point number equivalent to the unsigned integer 0xFFFF

10 is the magnitude of the full scale output voltage of the FHX-8CV

$$x = \frac{65535.0}{P_{max}} \times \frac{O_{max}}{10} = \frac{O_{max} \times 6553.5}{P_{max}}$$

Then we can just apply that scaling factor to the calculated values of P and cast the result as an unsigned integer of 16 bits (uint16_t in C++).

Example:

$P_{max} = 1$

$O_{max} = 5$ (we don't want to output voltages outside 0V to 5V)

$$x = \frac{O_{max} \times 6553.5}{P_{max}} = \frac{5 \times 6553.5}{1} = 32767.5$$

With this scaling factor $x = 32767.5$ we can see that for internal values of P in the range 0 to 1 we get bytes for sending over SPI in the range 0x0000 to 0x7FFF which gives output CV in the range 0V to 5V:

$P = 0$

$P \times x = 0 \times 32767.5 = 0$

Casting this to unsigned int we get 0x0000 which will give us 0V at the output.

$P = 1$

$P \times x = 1 \times 32767.5 = 32767.5$

Casting this to unsigned int we get 0x7FFF which will give us 5V at the output.

The bipolar case is similar. The table sets out the bytes and voltage outputs:

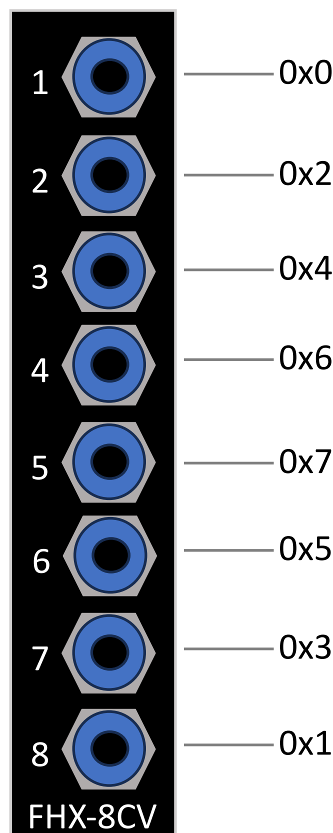
Channel Offset 0xFF (all channels bipolar)		
Bytes sent to Channel	***Expected Voltage***	***Measured Voltage***
0x0000	-5V	-4.94V
0x7DDC	0V	-0.07V
0xFFFF	+5V	4.93V

The span of output voltage is 10V in the -5V to +5V case, so the math is exactly as above, we just have to remember the centre voltage (at byte code 0x7DDC) is going to be 0V at the output.

If the voltage output is to be constrained to a given range such as 0V-5V the software sending bytes over SPI will need to check that the bytes do not represent voltages outside the min-to-max range. The FHX-8CV will operate across the full 10V range if the sent bytes ask it to do so.

Part 3 - FHX-8CV Channel Numbering

The FHX-8CV expander has 8 CV outs on the front panel. The correspondence between channel number data in the SPI packet and the front panel jack is shown in the diagram below.



Part 4 - Addressing the FHX-8CV Expander Modules

Up to 7 FHX-8CV expanders modules can be addressed. Expander selection jumpers on the rear of the module must have been set appropriately.

See: [Expert Sleepers - FHX-8CV Expander for FH-2 User Manual (expert-sleepers.co.uk)]

The addressing of a particular FHX-8CV module to be the subject of SPI bus packets is accomplished using three GPIO signals, A2, A1, A0 connected to the FHX-8CV Expansion In connector. These signals cannot be left floating. The addressing truth table is:

A2	A1	A0	FHX-8CV addr
0	0	0	FH2
0	0	3V3	1
0	3V3	0	2
0	3V3	3V3	3
3V3	0	0	4
3V3	0	3V3	5
3V3	3V3	0	6
3V3	3V3	3V3	7

In the Expert Sleepers FH2 system, there are 8x CV outs on the FH2 itself, hence when connected to the FH2, the address 000 is not used with FHX-8CV expanders: these expanders ship with the jumpers already set up to configure the module as expander 1.

FHX-8CV Coding for Patch.Init in C++

This assumes a working dev environment is set up with a new project created for the Patch.Init/Patch-SM.

See the section on FHX-8GT above for code segments used to set up the GPIO pins and the SPI pins.

Addressing the FHX-8CV as expander 1

```
    //set up the GPIO pins to addresss the FHX-8CV with its jumpers set as 1/1
    dsy_gpio_write(&patch_A3, 1);
    dsy_gpio_write(&patch_A8, 0);
    dsy_gpio_write(&patch_A9, 0);
```

This code sample below sets up a config.

```
        //Change the 8CV config to a 10V output scale on all channels, bipolar
for +/-5V

        //ch1
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x07; // 0 = channel 1 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch2
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x27; // 2 = channel 2 and data = 7DDC
        my_buffer[1] = 0xDD;
```

```
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch3
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x47; // 4 = channel 3 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch4
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x67; // 6 = channel 4 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch5
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x77; // 7 = channel 5 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch6
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x57; // 5 = channel 6 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch7
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x37; // 3 = channel 7 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //ch8
        my_buffer[3] = 0x03;
        my_buffer[2] = 0x17; // 1 = channel 8 and data = 7DDC
        my_buffer[1] = 0xDD;
        my_buffer[0] = 0xC0;
        spi_handle.BlockingTransmit(my_buffer, 4);

        //Config Packet
        my_buffer[3] = 0x14;
        my_buffer[2] = 0x00; //
```

```
        my_buffer[1] = 0xFF;
        my_buffer[0] = 0x00;
        spi_handle.BlockingTransmit(my_buffer, 4);
```

This code outputs some voltages on channel 1 of the FHX-8CV

```
    //send some different CV values to FHX-8CV channel 1

        my_buffer[3] = 0x03;
        my_buffer[2] = 0x00; //0x0- = ch1
        my_buffer[1] = 0x80;
        my_buffer[0] = 0xF0;
        spi_handle.BlockingTransmit(my_buffer, 4);
        System::Delay(500);

        my_buffer[3] = 0x03;
        my_buffer[2] = 0x04; //0x0- = ch1
        my_buffer[1] = 0xCC;
        my_buffer[0] = 0xD2;
        spi_handle.BlockingTransmit(my_buffer, 4);
        System::Delay(500);

        my_buffer[3] = 0x03;
        my_buffer[2] = 0x06; ; //0x0- = ch1
        my_buffer[1] = 0x66;
        my_buffer[0] = 0x60;
        spi_handle.BlockingTransmit(my_buffer, 4);
        System::Delay(500);

        my_buffer[3] = 0x03;
        my_buffer[2] = 0x08; ; //0x0- = ch1
        my_buffer[1] = 0x00;
        my_buffer[0] = 0x00;
        spi_handle.BlockingTransmit(my_buffer, 4);
        System::Delay(500);
```

Footnote: The info in these notes was reverse-engineered using a logic analyser. Prior to publishing the info, a check was made with Expert sleepers and they were OK about publication of it. Thanks! But this info is 100% unofficial so please don't bother Expert Sleepers for support with this unofficial application. BTW this approach should easily map to Raspberry Pi or any other microprocessor with SPI where modular outputs are needed.