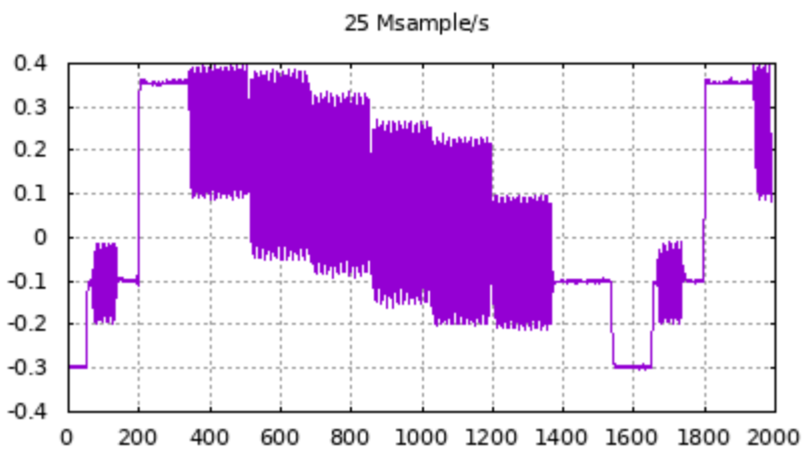
Lean2
Embedded systems without the bloat

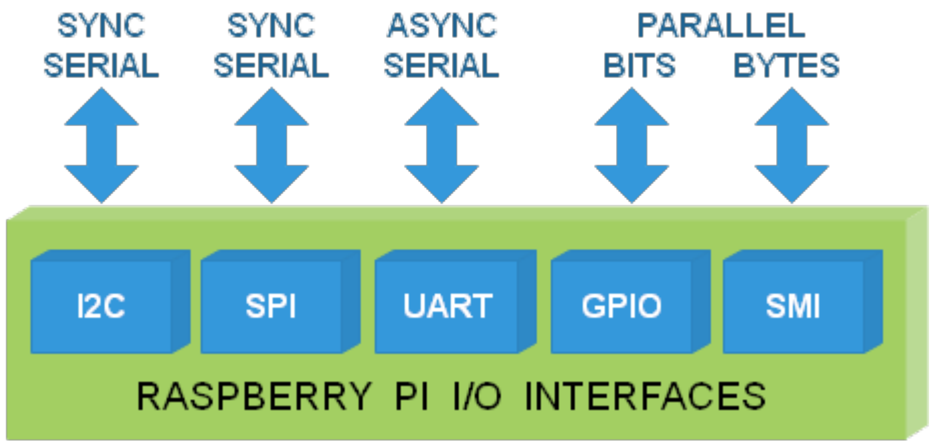
Raspberry Pi Secondary Memory Interface (SMI)



Colour video signal captured at 25 MS/s

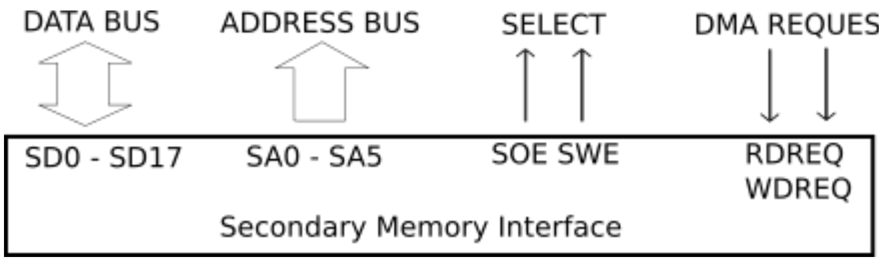
The Secondary Memory Interface (SMI) is a parallel I/O interface that is included in all the Raspberry Pi versions. It is rarely used due to the acute lack of publicly-available documentation; the only information I can find is in the source code to an external memory device driver here, and an experimental IDE interface here.

However, it is a very useful general-purpose high-speed parallel interface, that deserves wider usage; in this post I'm testing it with digital-to-analogue and analogue-to-digital converters (DAC and ADC) but there are many other parallel-bus devices that would be suitable.



To take advantage of the high data rates, I'll be using the C language, and Direct Memory Access (DMA); if you are unfamiliar with DMA on the RPi, I suggest you read my previous 2 posts on the subject, here and here.

Parallel interface

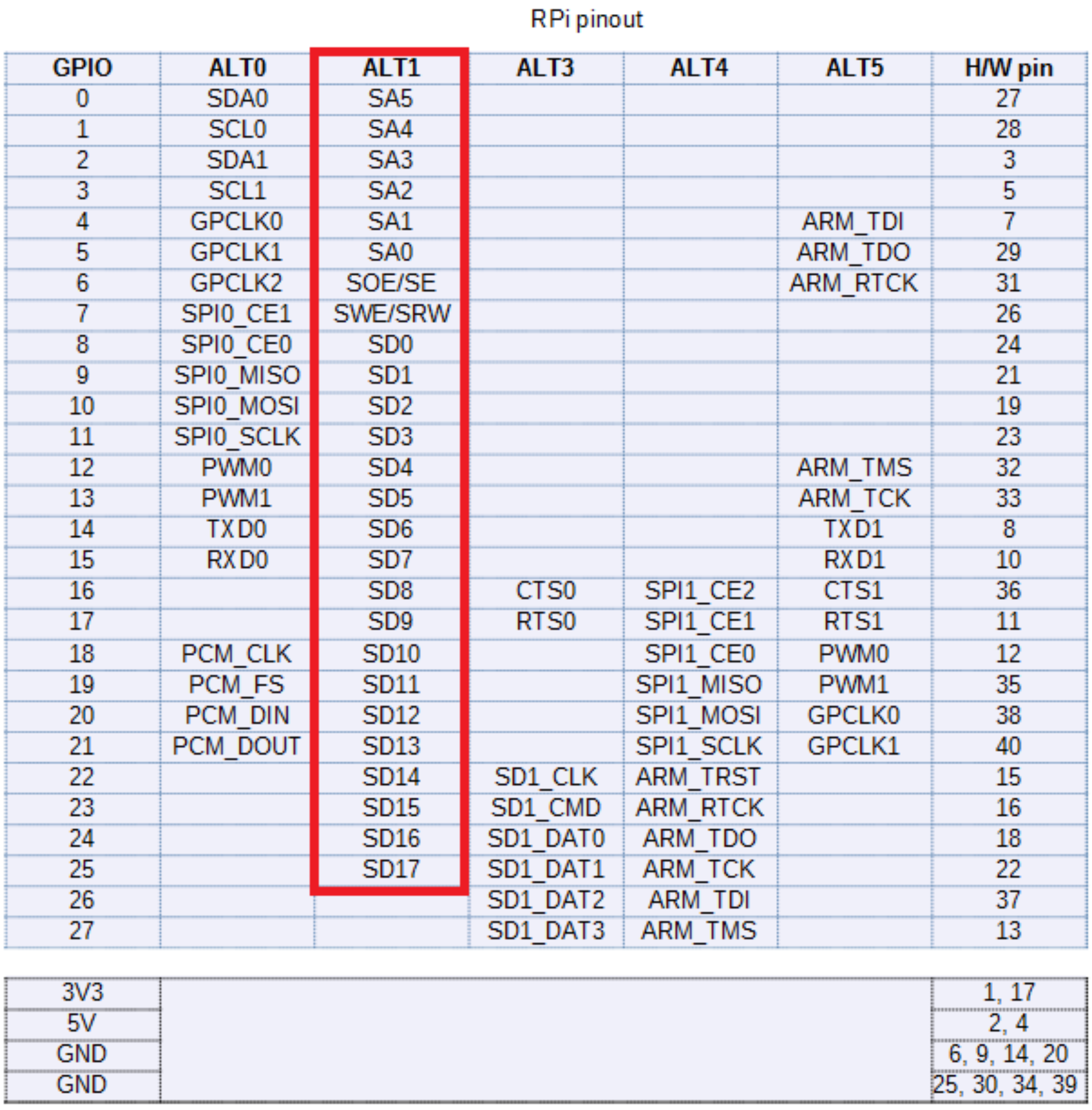


Raspberry Pi SMI signals

The SMI interface has up to 18 bits of data, 6 address lines, read & write select lines. Transfers can be initiated internally, or externally via read & write request lines, which can take over the uppermost 2 bits of the data bus. Transfer data widths are 8, 9, 16 or 18 bits, and are fully supported by First In First Out (FIFO) buffers, and DMA; this makes for efficient memory usage when driving an 8-bit peripheral, since a single 32-bit DMA transfer can automatically be converted into four 8-bit accesses.

If you have ever worked with the classic bus-interfaces of the original microprocessors, you'll feel quite at home with SMI, but no need to worry about timing problems, because the setup, strobe & hold times are fully programmable with 4 nanosecond resolution; what luxury!

The SMI functions are assigned to specific GPIO pins:

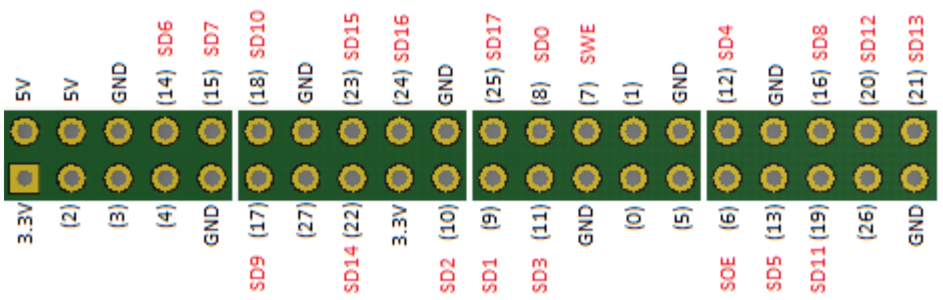


RPi pinout

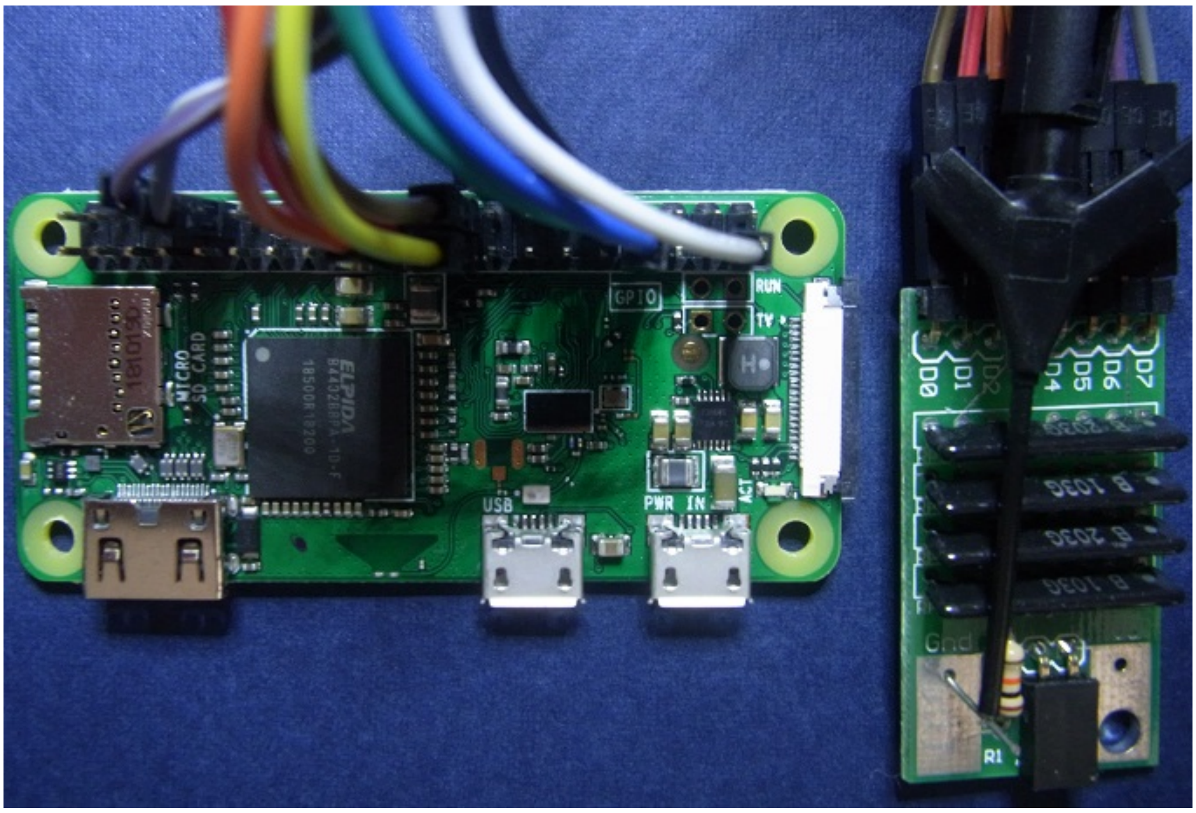
GPIO	ALT0	ALT1	ALT3	ALT4	ALT5	H/W pin
0	SDA0	SA5				27
1	SCL0	SA4				28
2	SDA1	SA3				3
3	SCL1	SA2				5
4	GPCLK0	SA1			ARM_TDI	7
5	GPCLK1	SA0			ARM_TDO	29
6	GPCLK2	SOE/SE			ARM_RTCK	31
7	SPI0_CE1	SWE/SRW				26
8	SPI0_CE0	SD0				24
9	SPI0_MISO	SD1				21
10	SPI0_MOSI	SD2				19
11	SPI0_SCLK	SD3				23
12	PWM0	SD4			ARM_TMS	32
13	PWM1	SD5			ARM_TCK	33
14	TXD0	SD6			TXD1	8
15	RXD0	SD7			RXD1	10
16		SD8	CTS0	SPI1_CE2	CTS1	36
17		SD9	RTS0	SPI1_CE1	RTS1	11
18	PCM_CLK	SD10		SPI1_CE0	PWM0	12
19	PCM_FS	SD11		SPI1_MISO	PWM1	35
20	PCM_DIN	SD12		SPI1_MOSI	GPCLK0	38
21	PCM_DOUT	SD13		SPI1_SCLK	GPCLK1	40
22		SD14	SD1_CLK	ARM_TRST		15
23		SD15	SD1_CMD	ARM_RTCK		16
24		SD16	SD1_DAT0	ARM_TDO		18
25		SD17	SD1_DAT1	ARM_TCK		22
26			SD1_DAT2	ARM_TDI		37
27			SD1_DAT3	ARM_TMS		13

3V3		1, 17
5V		2, 4
GND		6, 9, 14, 20
GND		25, 30, 34, 39

The GPIO pins to be included in the parallel interface are selected by setting their mode to ALT1; there is no requirement to set all the SMI pins in this way, so the I2C, SPI and PWM interfaces are still quite usable.



Parallel DAC



HARDWARE

The simplest device to drive from the parallel bus is a digital-to-analogue converter (DAC), using resistors from each data line to a common output. This arrangement is commonly known as an R-2R ladder, due to the resistor values needed.

I've used a pre-built device from Digilent (details here, or newer version here) but it is easy to make your own using discrete resistors; the least-significant is connected to GPIO8 (SD0), and the most-significant to GPIO15 (SD7).

SOFTWARE

I'll be making extensive use of the dma_utils functions that were created for my previous DMA projects, but before diving into the complication of SMI, it is helpful to test the hardware using simpler GPIO commands:

```
#define DAC_D0_PIN      8
#define DAC_NPINS       8

extern MEM_MAP gpio_regs;
```

```
map_periph(&gpio_regs, (void *)GPIO_BASE, PAGE_SIZE);

// Output value to resistor DAC (without SMI)
void dac_ladder_write(int val)
{
    *REG32(gpio_regs, GPIO_SET0) = (val & 0xff) << DAC_D0_PIN;
    *REG32(gpio_regs, GPIO_CLR0) = (~val & 0xff) << DAC_D0_PIN;
}

// Initialise resistor DAC
void dac_ladder_init(void)
{
    int i;

    for (i=0; i<DAC_NPINS; i++)
        gpio_mode(DAC_D0_PIN+i, GPIO_OUT);
}

// Output sawtooth waveform
dac_ladder_init();
while (1)
{
    i = (i + 1) % 256;
    dac_ladder_write(i);
    usleep(10);
}
```

This is less-than-ideal because we have to use one command to set some I/O pins to 1, and another command to clear the rest to 0, so in the gap between them the I/O state will be incorrect; also we won't get accurate timing with the usleep command.

To my surprise, when I ran this code on a Pi Zero, and viewed the output on an oscilloscope, it didn't look too bad; however, as soon as I moved the mouse, there were very significant gaps in the output, so clearly we need to do better.

SMI REGISTER DEFINITIONS

To use SMI, we first need to define the control registers, and the bit-values within them. The primary reference is bcm2835_smi.h from the Broadcom external memory driver, but I found this difficult to use in my code, so converted the definitions into C bitfields; this makes the code a bit less portable, but a lot simpler and easier to read.

Also, when learning about a new peripheral, it is helpful if the bitfield values can be printed on the console. This normally requires the tedious copying of register field names into string constants, but with a small amount of macro processing, this can be done with a single definition, for example the SMI CS register:

```
#define REG_DEF(name, fields) typedef union {struct {volatile uint32_t f

#define SMI_CS_FIELDS \
    enable:1, done:1, active:1, start:1, clear:1, write:1, _x1:2,\
    teen:1, intd:1, intt:1, intr:1, pvmode:1, seterr:1, pxldat:1, edreq:
    _x2:8, _x3:1, aferr:1, txw:1, rxr:1, txd:1, rxd:1, txe:1, rxf:1
REG_DEF(SMI_CS_REG, SMI_CS_FIELDS);

volatile SMI_CS_REG  *smi_cs;

smi_cs  = (SMI_CS_REG *) REG32(smi_regs, SMI_CS);
```

The last bit of code is needed so that smi_cs points to the register in virtual memory; if you don't understand why, I suggest you read my post on RPi DMA programming here. Anyway, the upshot of all this code is that we can access the whole 32-bit value of the register as smi_cs->value, and also individual bits such as smi_cs->enable, smi_cs->done, etc.

To print out the bit values, we use macros to convert the register definition to a string, then have a simple C parser:

```
#define STRS(x)     STRS_(x) ","
#define STRS_(...)  #__VA_ARGS__

char *smi_cs_regstrs = STRS(SMI_CS_FIELDS);

// Display bit values in register
void disp_reg_fields(char *regstrs, char *name, uint32_t val)
{
    char *p=regstrs, *q, *r=regstrs;
    uint32_t nbits, v;

    printf("%s %08X", name, val);
    while ((q = strchr(p, ':')) != 0)
    {
        p = q + 1;
        nbits = 0;
        while (*p>='0' && *p<='9')
            nbits = nbits * 10 + *p++ - '0';
        v = val & ((1 << nbits) - 1);
        val >>= nbits;
        if (v && *r!='_')
            printf(" %.*s=%X", q-r, r, v);
        while (*p==',' || *p==' ')
            p = r = p + 1;
    }
    printf("\n");
}
```

Now we can display all the non-zero bit values using:

```
disp_reg_fields(smi_cs_regstrs, "CS", *REG32(smi_regs, SMI_CS));
```

..which produces a display like..

```
CS 54000025 enable=1 active=1 write=1 txw=1 txd=1 txe=1
```

SMI REGISTERS

The SMI registers are:

```
CS:  control and status
L:   data length (number of transfers)
A:   address and device number
D:   data FIFO
DMC: DMA control
DSR: device settings for read
DSW: device settings for write
DCS: direct control and status
DCA: direct control address and device number
DCD: direct control data
```

You can specify up to 4 unique timing settings for read & write, making 8 settings in total. The settings are specified by giving a 2-bit device number for each transaction; this selects 1 of the 4 descriptors for read or write. I've only used one pair of settings, and the ADC & DAC don't have address lines, so the address & device register remains at zero.

Direct mode is a simple way of doing accesses using the appropriate timings, but without DMA; it has separate address, data and control registers.

Some notable fields in the control & status register are:

Enable: it is obvious that this bit must be set for SMI to work, but it is less obvious when that should be done. Initially, I assumed it was necessary to enable the interface before any other initialisation, but then it responded with the 'settings error' bit set. So now I do most of the configuration with the device disabled, then enable it before clearing the FIFOs and enabling DMA, otherwise the transfers go through immediately.

Start: set this bit to start the transfer; the SMI controller will perform the number of transfers in the length register, using the timing parameters specified in DSR (for read) or DSW (for write). If there is a backlog of data (FIFO is full) the transaction may stall.

Pxldat: when this 'pixel data' bit is set, the 8- or 16-bit data is packed into 32-bit words.

Pvmode: I have no idea what this 'pixel valve' mode should do; any information would be gratefully received.

DIRECT MODE

As the name implies, SMI Direct Mode allows you to perform a single I/O transfer without DMA. However, it is still necessary to specify the timing parameters of the transfer, specifically:

- The clock period, that will be used for the following timing:
  - The setup time, that is used by the peripheral to decode the address value
  - The width of the strobe pulse, that triggers the transfer
  - The hold time, that keeps the signals stable after the transfer

To add to the complication, the SMI controller can drive 4 peripheral devices, each with its own individual read & write settings, so there are a total of 8 timing registers. I'm keeping this simple by always using the first register pair (for device zero) but it is worth remembering that you can define more than one set of timings, and quickly switch between them by setting the device number.

Likewise, I'm ignoring the address field since it is also redundant for my DAC; for safety, I clear all the SMI registers on startup, in case there are any residual unwanted values.

As it happens, this setup/strobe/hold timing is largely redundant for our simple resistor DAC (since it doesn't latch the data) but we still need to specify something, for example if we want the overall cycle time to be 1 microsecond, this can be achieved with a clock period of 10 nanoseconds, setup 25, strobe 50, and hold 25, since (25 + 50 + 25) * 10 = 1000 nanoseconds. This is the code I use to set the timing:

```
// Width values
#define SMI_8_BITS  0
#define SMI_16_BITS 1
#define SMI_18_BITS 2
#define SMI_9_BITS  3

// Initialise SMI interface, given time step, and setup/hold/strobe cour
// Clock period is in nanoseconds: even numbers, 2 to 30
void init_smi(int width, int ns, int setup, int strobe, int hold)
{
    int divi = ns/2;

    smi_cs->value = smi_l->value = smi_a->value = 0;
    smi_dsr->value = smi_dsw->value = smi_dcs->value = smi_dca->value =
    if (*REG32(clk_regs, CLK_SMI_DIV) != divi << 12)
    {
        *REG32(clk_regs, CLK_SMI_CTL) = CLK_PASSWD | (1 << 5);
        usleep(10);
        while (*REG32(clk_regs, CLK_SMI_CTL) & (1 << 7)) ;
        usleep(10);
        *REG32(clk_regs, CLK_SMI_DIV) = CLK_PASSWD | (divi << 12);
        usleep(10);
        *REG32(clk_regs, CLK_SMI_CTL) = CLK_PASSWD | 6 | (1 << 4);
        usleep(10);
        while ((*REG32(clk_regs, CLK_SMI_CTL) & (1 << 7)) == 0) ;
        usleep(100);
    }
    if (smi_cs->seterr)
        smi_cs->seterr = 1;
    smi_dsr->rsetup = smi_dsw->wsetup = setup;
    smi_dsr->rstrobe = smi_dsw->wstrobe = strobe;
    smi_dsr->rhold = smi_dsw->whold = hold;
    smi_dsr->rwidth = smi_dsw->wwidth = width;
}
```

The clock-frequency-setting code is similar to that I used to set the PWM frequency for my DMA pacing; that peripheral did seem to be really sensitive to any glitches in the clock, so I've been a bit over-cautious in adding extra time-delays, which may not really be necessary.

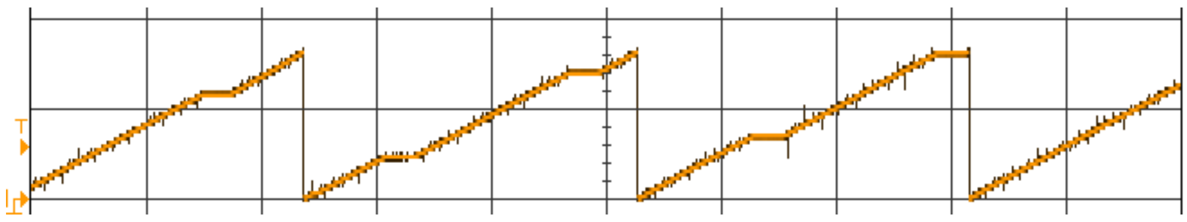
The seterr flag is supposed to indicate an error if the settings have been changed while the SMI device is active; the easiest way to avoid this error is to do most of the settings while the device is disabled, then enable it just before starting; the flag is also cleared on startup, by writing a 1 to it.

Once the timing is set, the following code can be used to initiate a single direct-control write-cycle:

```
// Initialise resistor DAC
void dac_ladder_init(void)
{
    smi_cs->clear = 1;
    smi_cs->aferr = 1;
    smi_dcs->enable = 1;
}

// Output value to resistor DAC
void dac_ladder_write(int val)
{
    smi_dcs->done = 1;
    smi_dcs->write = 1;
    smi_dcd->value = val & 0xff;
    smi_dcs->start = 1;
}
```

The code clears the FIFO, in case there is any data left over from a previous transaction (which isn't unusual, if you have been using DMA), and the FIFO error flag, then enables the device. The transfer is initiated by clearing the completion flag, setting write mode, loading the value into the Direct Mode data register, then starting the cycle.

The transfer then proceeds using the specified timing, and the completion flag is set when complete. If we run this code with usleep for timing, there is very little difference in the DAC output; it is still susceptible to other events, such as mouse movement, as shown in the oscilloscope trace below.

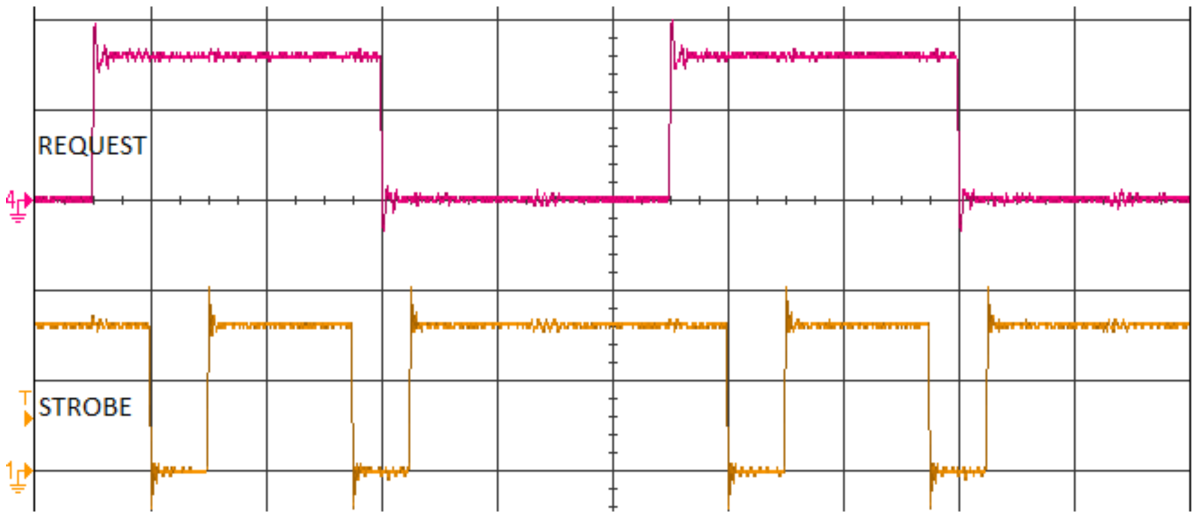


To gain maximum benefit from SMI, we have to use DMA.

SMI AND DMA

When using SMI with DMA, the fundamental question is where the DMA requests will be coming from.

They can be triggered by an external signal, in ‘DMA passthrough’ mode. The data lines SD 16 & 17 can be used as triggers; SD16 to write to an external device or SD17 to read from external device, with a maximum data width of 16 bits. It is important to note that they are level-sensitive signals (not edge-triggered) so if held high, the transfers will carry on at the maximum rate; see the oscilloscope trace below, where a 500 ns request is sufficient to trigger 2 transfers.



Oscilloscope trace of DMA passthrough (200 ns/div)

So DMA passthrough is designed for use with peripherals that assert the request when they have data to send, and negate it when the transfer has gone through. I have experimented with the PWM controller to generate narrow pulses, and it does seem possible to trigger single transfers this way, but more tests are needed to make sure this method is 100% reliable, so for the time being I won’t use it.

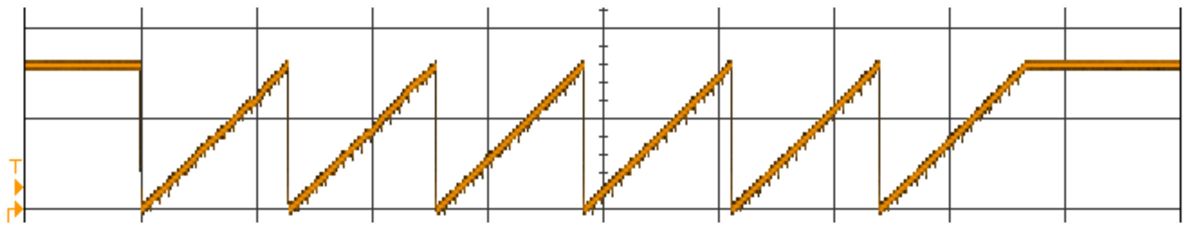
Instead, the requests will originate from the SMI controller itself; the transfer will proceed at the maximum speed defined by the setup, strobe & hold times, with DMA keeping the FIFOs topped up with data. This places a lower limit on the rate at which the transfers go through; the maximum clock resolution is 30 ns, and the maximum setup, strobe & hold values are 63, 127 and 63, giving a slowest cycle time of 7.6 microseconds.

The DMA Control Block is similar to those in my previous projects; it just needs a data source in uncached memory, data destination as the SMI FIFO, and length

```
#define NCYCLES 4

// DMA values to resistor DAC
void dac_ladder_dma(MEM_MAP *mp, uint8_t *data, int len, int repeat)
{
    DMA_CB *cbs=mp->virt;
    uint8_t *txdata=(uint8_t *)(cbs+1);

    memcpy(txdata, data, len);
    enable_dma(DMA_CHAN_A);
    cbs[0].ti = DMA_DEST_DREQ | (DMA_SMI_DREQ << 16) | DMA_CB_SRCE_INC;
    cbs[0].tfr_len = NSAMPLES * NCYCLES;
    cbs[0].srce_ad = MEM_BUS_ADDR(mp, txdata);
    cbs[0].dest_ad = REG_BUS_ADDR(smi_regs, SMI_D);
    cbs[0].next_cb = repeat ? MEM_BUS_ADDR(mp, &cbs[0]) : 0;
    start_dma(mp, DMA_CHAN_A, &cbs[0], 0);
}

smi_dsr->rwidth = SMI_8_BITS;
smi_l->len = NSAMPLES * REPEATS;
smi_cs->pxldat = 1;
smi_dmc->dmaen = 1;
smi_cs->write = 1;
smi_cs->enable = 1;
smi_cs->clear = 1;
dac_ladder_dma(&vc_mem, sample_buff, sample_count, NCYCLES>1);
smi_cs->start = 1;
```

A convenient way of outputting a repeating waveform is to create one cycle in memory, and set the control block to that length. Then the SMI length is set to the total number of bytes to be sent, assuming the pixel mode flag ‘pxldat’ has been set; this instructs the SMI controller to unpack the 32-bit DMA & FIFO values into 4 sequential output bytes.

The following trace was generated by a 256-byte ramp, repeated 6 times, using a 1 microsecond cycle time.



Oscilloscope trace of DAC output (200 us/div)

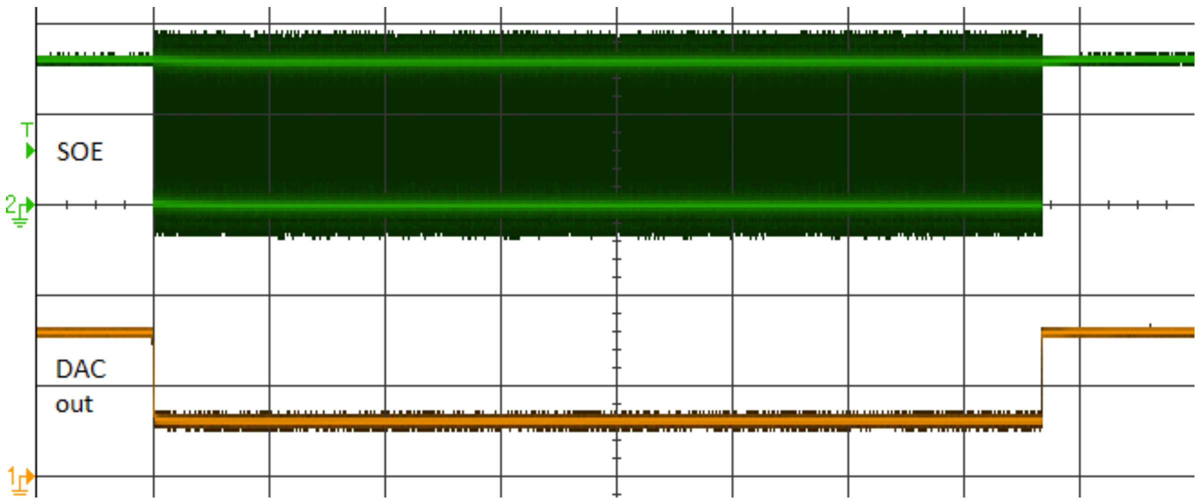
The SMI interface can generate much faster waveforms, but unfortunately they aren't rendered very well by the DAC as it uses 10K resistors; when these are combined with the oscilloscope probe input capacitance, the resulting rise time is around 500 nanoseconds. So for faster waveforms, you need a faster DAC.

READ CYCLE TEST

The last DAC test I'm going to do will seem a bit crazy: a read cycle. The settings are the same as the write-cycle, with the following changes:

```
smi_cs->write = 1;

cbs[0].ti = DMA_SRCE_DREQ | (DMA_SMI_DREQ << 16) | DMA_CB_DEST_INC;
cbs[0].srce_ad = REG_BUS_ADDR(smi_regs, SMI_D);
cbs[0].dest_ad = MEM_BUS_ADDR(mp, txdata);
```

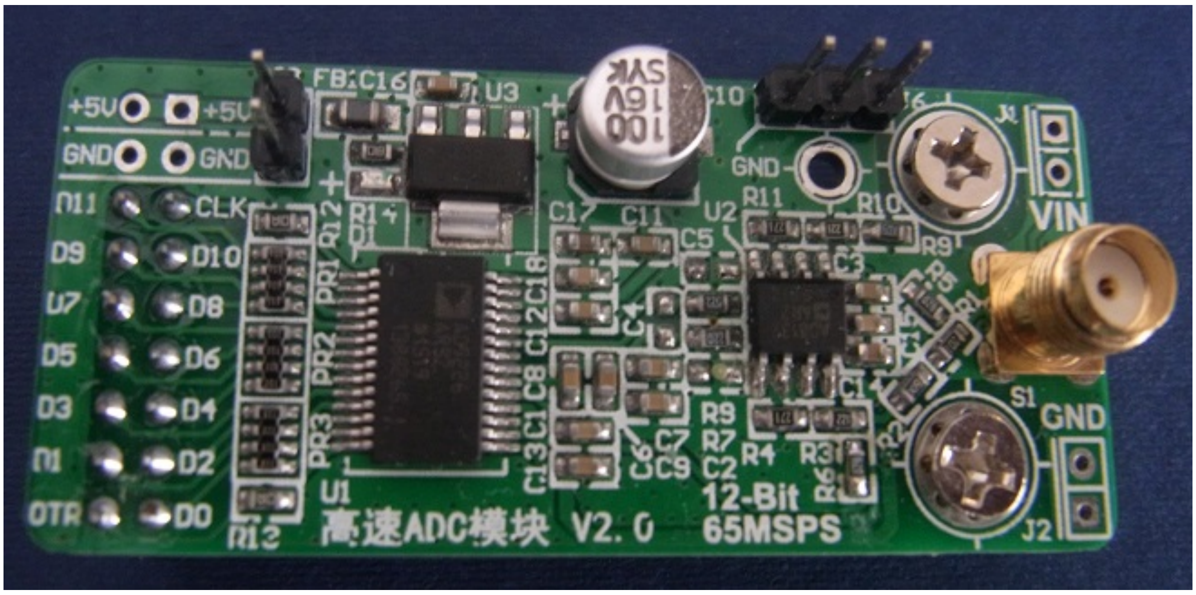
The scope has been set to additionally show the SOE signal as the top trace:



The DAC output starts at 3.3V which was the final value of the previous output cycle. It then drops to 1.2V during the read cycles, as this is the value it floats to when the I/O lines aren't being driven. At the end of the last read cycle, the output is driven back to 3.3V.

This is a very important result; as soon as the input cycles stop, SMI drives the bus. This is because memory chips don't like a floating data bus; a halfway-on voltage can cause excessive power dissipation, and even damage the chip in extreme cases. So it is a sensible precaution that the data bus is always driven, though this is about to cause a major problem...

AD9226 ADC



Searching the Internet for a fast low-cost analogue-to-digital (ADC) module with a parallel interface, I found very few; the best one featured the 12-bit AD9226, with a maximum throughput of 65 megasamples per second. It requires a 5 volt supply, but has a 3.3V logic interface, so is compatible with the Raspberry Pi.

Having worked with the module for a few days, I've found it to be less than ideal, for various reasons that'll be given later, but it is still useful to demonstrate high-speed parallel input with SMI.

Connecting to the RPi isn't difficult, but as we're dealing with high-speed signals, it is necessary to keep the wiring short, preferably under 50 mm (2 inches), especially the power, ground & clock signals.

One minor confusion is that the pin marked D0 is the most-significant bit, and D11 the least significant; I wanted to leave the SPI0 pins free, so adopted the following connection scheme, which puts the data in the top 12 bits of a 16-bit SMI read cycle.:

```
H/W pin Function    AD9226
------- ----------- ------
31  GPIO06 SOE  CLK
16  GPIO23 SD15 D0 (MSB)
15  GPIO22 SD14 D1
40  GPIO21 SD13 D2
38  GPIO20 SD12 D3
```

```
35  GPIO19 SD11 D4
12  GPIO18 SD10 D5
11  GPIO17 SD9  D6
36  GPIO16 SD8  D7
10  GPIO15 SD7  D8
8   GPIO14 SD6  D9
33  GPIO13 SD5  D10
32  GPIO12 SD4  D11 (LSB)
2   5V        +5V
6   GND       GND
```

DIRECT MODE

We'll start by using Direct Mode to obtain an sample without DMA. The ADC is designed to work with a continuous clock signal, but ours is derived from the SMI Output Enable (OE) line, so only changes state during data transfers.

The AD9226 data sheet describes how it stabilises the clock signal, and suggests it may require over 100 cycles when adapting to a new frequency. In practice, when starting up there seems to be a major data glitch after 8 cycles, but after that the conversions appear to have stabilised, so I allow for 10 cycles before taking a reading.

It is necessary to choose timing values for the SMI cycles; my default settings are 10 nanosecond time interval, with a setup of 25, strobe 50, hold 25, so the total cycle time is 10 * (25 + 50 + 25) = 1000 nanoseconds, or 1 megasample/sec.

```
for (i=0; i<ADC_NPINS; i++)
    gpio_mode(ADC_D0_PIN+i, GPIO_IN);
gpio_mode(SMI_SOE_PIN, GPIO_ALT1);

init_smi(SMI_16_BITS, 10, 25, 50, 25); // 1 MS/s

smi_start(10, 1);
usleep(20);
val = adc_gpio_val();
printf("%4u %1.3f\n", val, val_volts(val));
```

VOLTAGE VALUE

The ADC has an op-amp input circuit that can accommodate positive and negative voltages. Converting the ADC value to a voltage is a bit fraught; I determined the following values by experimentation with one module, but suspect they are subject to quite wide component tolerances, so won't be the same for all modules.

```c
#define ADC_ZERO        2080
#define ADC_SCALE       410.0

// Convert ADC value to voltage
float val_volts(int val)
{
    return((ADC_ZERO - val) / ADC_SCALE);
}

// Return ADC value, using GPIO inputs
int adc_gpio_val(void)
{
    int v = *REG32(gpio_regs, GPIO_LEV0);

    return((v>>ADC_D0_PIN) & ((1 << ADC_NPINS)-1));
}
```

It is important to note that the module has a 50-ohm input, so imposes a very heavy loading on any circuit it is monitoring. It can't cope with significant voltages for any period of time; for example, if you apply 5 volts, the input resistor will dissipate half a watt, heat up rapidly, and probably burn out.

So, although the ADC is excellent for fast data acquisition, the module isn't really suitable for general purpose measurement, and would benefit from a redesign with a high-impedance input.

AVOIDING BUS CONFLICTS

The module doesn't have a chip-select or chip-enable input, so the data is always being output; the 28-pin version of the AD9226 doesn't have the facility for disabling its output drivers. In the above code I avoided the possibility of bus conflicts doing a GPIO register read, but for high speeds we have to use SMI read cycles. This is potentially a major problem; when the read cycles are complete, the SMI controller and the ADC will both try to drive the data bus at the same time, causing significant current draw, only limited by the 100 ohm resistors on the module: they are insufficient to keep the current below the maximum values (16 mA per pin, 50 mA total for all I/O) in the Broadcom data sheet.

I've experimented with various software solutions, basically using a DMA Control Block to set the ADC pins to SMI mode (ALT1), then the second CB for the data transfer, then a third to set the pins back to GPIO inputs. The problem with this approach is that at the higher transfer rates the DMA controller is only just keeping up with the incoming data, and there is a sizeable backlog that has to be cleared before the DMA completes. So there is a significant delay before the SMI pins are set back to inputs, and in that time, there is a bus conflict.

For this reason (and to avoid any concerns about hardware damage when debugging new code) I added a resistor in series with each data line, to reduce the current flow when a bus conflict occurs. The value is a compromise; the resistance needs to be high enough to block excessive current, but not so high that it will slow down the I/O transitions too much, when combined with the stray capacitance of the GPIO inputs.

I chose 330 ohms, which combines with the 100 ohms already on the module, to produce a maximum current of 7.7 mA per line. This is well within the per-pin limit of the Broadcom device, but if all the lines are in conflict, the total will actually exceed the maximum chip I/O current, so it is inadvisable to leave the hardware in this state for a significant period of time.

ADC CODE

If you've read my previous blogs on fast ADC data capture, the DMA code will seem quite familiar, with control blocks to set the GPIO pins to SMI mode, capture the data, and restore the pins:

```
// Get GPIO mode value into 32-bit word
void mode_word(uint32_t *wp, int n, uint32_t mode)
{
    uint32_t mask = 7 << (n * 3);
    *wp = (*wp & ~mask) | (mode << (n * 3));
}

// Start DMA for SMI ADC, return Rx data buffer
uint32_t *adc_dma_start(MEM_MAP *mp, int nsamp)
{
    DMA_CB *cbs=mp->virt;
    uint32_t *data=(uint32_t *)(cbs+4), *pindata=data+8, *modes=data+0x1
    uint32_t *modep1=data+0x18, *modep2=modep1+1, *rxdata=data+0x20, i;
    // Get current mode register values
    for (i=0; i<3; i++)
        modes[i] = modes[i+3] = *REG32(gpio_regs, GPIO_MODE0 + i*4);
    // Get mode values with ADC pins set to SMI
    for (i=ADC_D0_PIN; i<ADC_D0_PIN+ADC_NPINS; i++)
        mode_word(&modes[i/10], i%10, GPIO_ALT1);
    // Copy mode values into 32-bit words
    *modep1 = modes[1];
    *modep2 = modes[2];
    *pindata = 1 << TEST_PIN;
    enable_dma(DMA_CHAN_A);
    // Control blocks 0 and 1: enable SMI I/P pins
    cbs[0].ti = DMA_SRCE_DREQ | (DMA_SMI_DREQ << 16) | DMA_WAIT_RESP;
    cbs[0].tfr_len = 4;
    cbs[0].srce_ad = MEM_BUS_ADDR(mp, modep1);
    cbs[0].dest_ad = REG_BUS_ADDR(gpio_regs, GPIO_MODE0+4);
    cbs[0].next_cb = MEM_BUS_ADDR(mp, &cbs[1]);
    cbs[1].tfr_len = 4;
    cbs[1].srce_ad = MEM_BUS_ADDR(mp, modep2);
    cbs[1].dest_ad = REG_BUS_ADDR(gpio_regs, GPIO_MODE0+8);
    cbs[1].next_cb = MEM_BUS_ADDR(mp, &cbs[2]);
    // Control block 2: read data
    cbs[2].ti = DMA_SRCE_DREQ | (DMA_SMI_DREQ << 16) | DMA_CB_DEST_INC;
    cbs[2].tfr_len = (nsamp + PRE_SAMP) * SAMPLE_SIZE;
    cbs[2].srce_ad = REG_BUS_ADDR(smi_regs, SMI_D);
    cbs[2].dest_ad = MEM_BUS_ADDR(mp, rxdata);
    cbs[2].next_cb = MEM_BUS_ADDR(mp, &cbs[3]);
    // Control block 3: disable SMI I/P pins
    cbs[3].ti = DMA_CB_SRCE_INC | DMA_CB_DEST_INC;
    cbs[3].tfr_len = 3 * 4;
    cbs[3].srce_ad = MEM_BUS_ADDR(mp, &modes[3]);
    cbs[3].dest_ad = REG_BUS_ADDR(gpio_regs, GPIO_MODE0);
    start_dma(mp, DMA_CHAN_A, &cbs[0], 0);
    return(rxdata);
}
```

When DMA is complete, we have a data buffer in uncached memory, containing left-justified 16-bit samples packed into 32-bit words; they are shifted and copied into the sample buffer. The first few samples are discarded as they are erratic; the ADC needs several clock cycles before its internal logic is stable.

```
// ADC DMA is complete, get data
int adc_dma_end(void *buff, uint16_t *data, int nsamp)
{
    uint16_t *bp = (uint16_t *)buff;
    int i;

    for (i=0; i<nsamp+PRE_SAMP; i++)
    {
        if (i >= PRE_SAMP)
            *data++ = bp[i] >> 4;
    }
    return(nsamp);
}
```

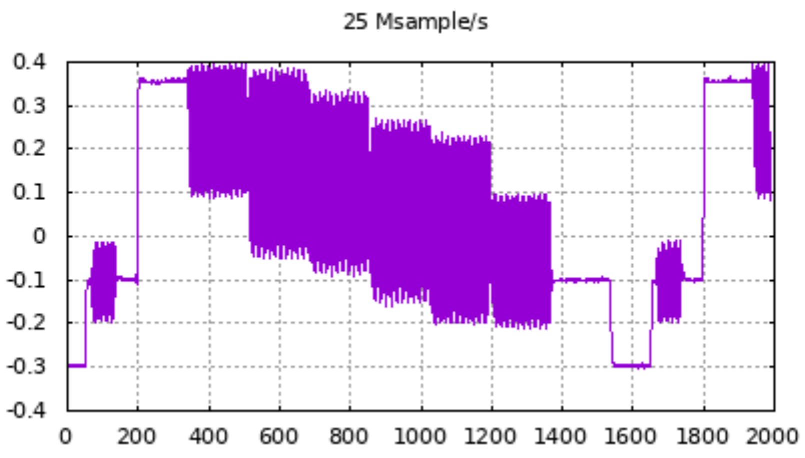
ADC speed tests

The important question is: how fast can we run the SMI interface? Here are the settings for some tests:

```
// RPi v0-3
#define SMI_NUM_BITS    SMI_16_BITS
#define SMI_TIMING      SMI_TIMING_25M
#define SMI_TIMING_1M   10, 25, 50, 25  // 1 MS/s
#define SMI_TIMING_20M   2,  6, 13,  6  // 20 MS/s
#define SMI_TIMING_25M   2,  5, 10,  5  // 25 MS/s
#define SMI_TIMING_31M   2,  4,  6,  4  // 31.25 MS/s
#define SMI_TIMING_50M   2,  3,  5,  2  // 50 MS/s

init_smi(SMI_16_BITS,  SMI_TIMING);
```

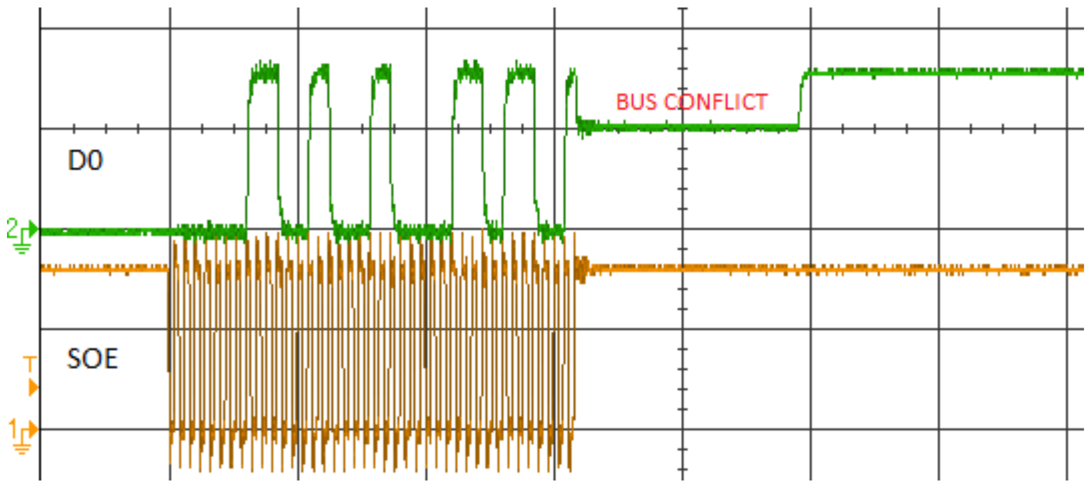
The SMI clock is 1 GHz; the first number is the clock divisor, followed by the setup, strobe & hold counts, so 1000 / (10 * (25+50+25)) = 1 MS/s. Where possible, I've tried to keep the waveform symmetrical by making setup + hold = strobe, but that isn't essential; the ADC can handle asymmetric clock signals.

RPI V3



25 MS/s capture of a video test waveform

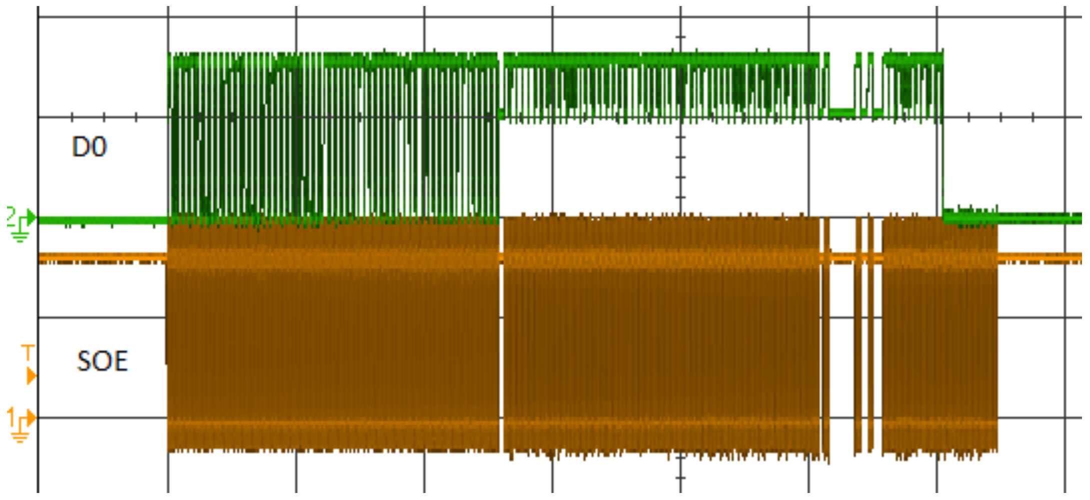
Running on a Raspberry Pi 3B v1.2, the fastest continuous rate that produces consistent results is 25 megasamples per second. The following trace shows a data line and the SOE (ADC clock) line, with a 40-byte transfer at 25 MS/s:



Scope trace 500 ns/div, 2 volts/div

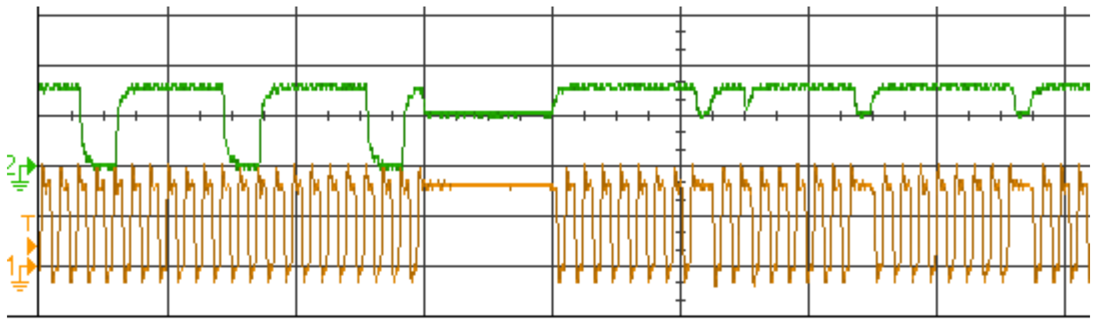
The data line is being measured on the ADC module connector, so when there is a bus conflict, the 100 ohm resistor on the module combines with the 330 ohms on the data line to form a potential divider, that makes the conflict easy to see. It is inevitable that there will be a brief conflict as the read cycles end, and the SMI controller takes control of the bus, but it only lasts 900 nanoseconds, which shouldn't be an issue, given the resistor values I'm using.

However, increasing the rate to 31.25 MS/s does cause a problem:



Scope trace 5 us/div, 2 volts/div

The system seems able to handle this rate fine for about 13 microseconds (400 samples), then it all goes wrong; there is a gap in the transfers, followed by continuous bus conflicts. Zooming in to that area, the SMI controller seems to transition between continuous evenly-paced cycles, to bursts of 8, with a continuous conflict:


Scope trace 200 ns/div, 2 volts/div

In the absence of any documentation on the SMI controller, it is difficult to speculate on the reasons for this, but it does emphasise the need for caution when working with high-speed transfers.

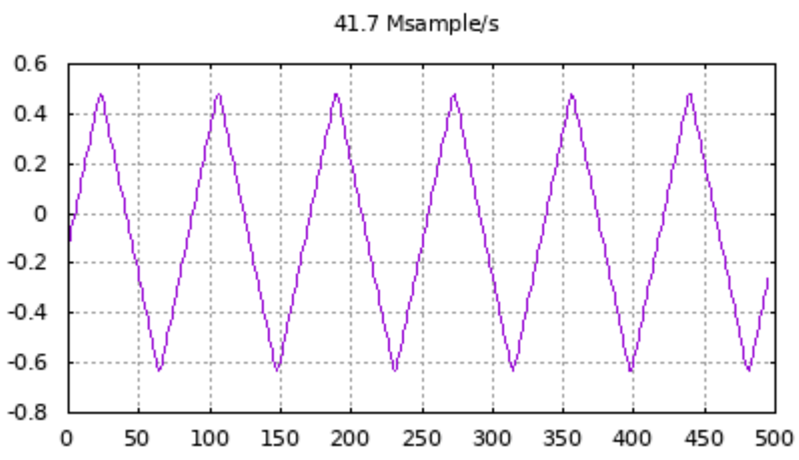
Since 16-bit transfers work at 25 MS/s, it should be possible to run 8-bit transfers at 50 MS/s. This can be tested using the following settings:

```
#define SMI_NUM_BITS    SMI_8_BITS
#define SMI_TIMING      SMI_TIMING_50M
#define SAMPLE_SIZE     1
```

With ADC connections I'm using, this doesn't produce useful data (just the top 4 bits from the ADC), but the waveforms look fine on an oscilloscope, so there doesn't seem to be a problem running 50 megabyte-per-second SMI transfers on an RPi v3.

PI ZEROW

Switching to a Pi ZeroW, the results are remarkably good; here is a 500 kHz triangle wave, captured at 41.7 megasamples per second



Capture of 500 kHz triangle wave

This does seem to be the top speed for a Pi ZeroW, as increasing the transfer rate to 50 MS/s causes some errors in the data. However, being able to transfer over 83 megabytes per second is a remarkably good result for this low-cost computer.

The question is whether this transfer rate is completely reliable; for example, is it disrupted by network activity? The easiest way to generate a lot of network traffic is using 'flood pings' from a Linux PC to the RPi; I did a few data captures with pings running, and they didn't seem to have any effect on the data, but more testing is needed.

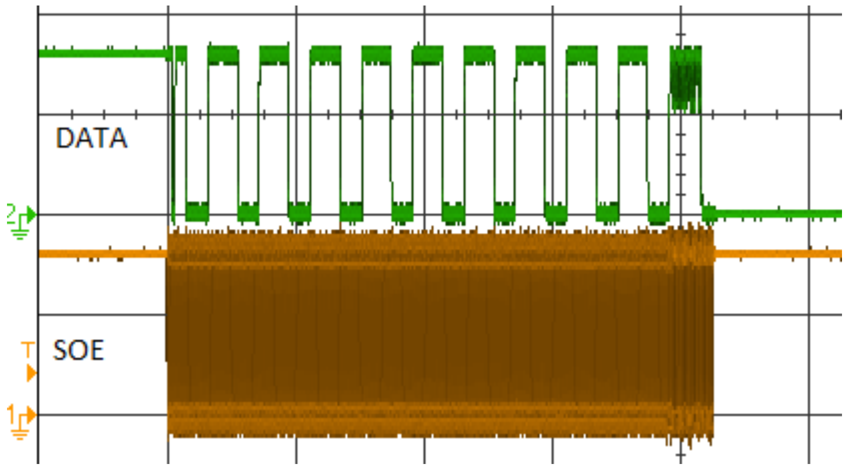
RPI V4

The first test of a Rpi v4 at 1 MS/s actually produced 1.5 MS/s, so the base SMI clock for RPi v4 must be 1.5 GHz. This means a new set of speed definitions:

```
// RPi v4
#define SMI_TIMING_1M   10, 38, 74, 38  // 1 MS/s
#define SMI_TIMING_10M   6,  6, 13,  6  // 10 MS/s
#define SMI_TIMING_20M   4,  5,  9,  5  // 19.74 MS/s
#define SMI_TIMING_25M   4,  3,  8,  4  // 25 MS/s
#define SMI_TIMING_31M   4,  3,  6,  3  // 31.25 MS/s
```

As before, the first number is the clock divisor, followed by the setup, strobe & hold counts, so 1500 / (10 * (38+74+38)) = 1 MS/s.

Unfortunately the maximum throughput with the current code is quite poor; the following trace is for 500 samples at 25 MS/s, and you can see the bus contention towards the end, similar to that I experienced on the RPi v3.



Scope trace 5 usec/div, 2 volts/div

The upper trace is the most significant ADC bit (measured at the module pin), and the analogue input is a 500 kHz sine wave, hence the regular bit transitions.

The key question is: why does the throughput get worse with a faster processor? I'd guess that this is a memory bandwidth issue; with a single core, the DMA controller can effectively monopolise the memory, always getting the data through. On a multi-core processor, it has to cooperate with all the cores that are active during the data capture.

Clearly more work is needed to understand this phenomenon, for example by manipulating the cores and process priorities; alternatively, for maximum performance, just use a Pi Zero!

Running the code

The source code is on Github here. The main files for DAC and ADC are rpi_smi_dac_test.c and rpi_smi_adc_test.c; the other files needed are rpi_dma_utils.c, rpi_dma_utils.h and rpi_smi_defs.h.

It is necessary to edit the top of rpi_dma_utils.h depending on which RPi hardware you are using:

```
// Location of peripheral registers in physical memory
#define PHYS_REG_BASE   PI_23_REG_BASE
#define PI_01_REG_BASE  0x20000000  // Pi Zero or 1
#define PI_23_REG_BASE  0x3F000000  // Pi 2 or 3
#define PI_4_REG_BASE   0xFE000000  // Pi 4
```

There are other settings at the top of the main files, that can be changed as required. The code can then be compiled with gcc, optionally with the -O2 option to optimise the code (which isn't really necessary), and the -pedantic option if you want to check for extra warnings:

```
gcc -Wall -pedantic -o rpi_smi_adc rpi_smi_adc.c rpi_dma_utils.c
```

The code is run using sudo, optionally with the CSV output piped to a file:

```
sudo ./rpi_smi_adc
..or..
sudo ./rpi_smi_adc > test6.csv
```

The CSV file can be imported into a spreadsheet, or plotted using Gnuplot from the RPi command line, e.g.

```
gnuplot -e "set term png size 420,240 font 'sans,8'; \
 set title '41.7 Msample/s'; set grid; set key noautotitle; \
 set output 'test6.png'; plot 'test6.csv' every ::10 with lines"
```

You may have read elsewhere that it is necessary to enable SMI in /boot/config.txt:

```
dtoverlay=smi    # Not needed!
```

This sets the GPIO mode of the SMI pins on startup; it isn't necessary for my code, which does its own GPIO configuration, with the added advantage that the unused pins are unchanged, so are free for use by other I/O functions.

If you want to see an example of SMI being used as a multi-channel pulse generator, see my 16 channel NeoPixel smart LED example here.

July 16, 2020 / Data acquisition, DMA, Raspberry Pi, Secondary Memory Interface

39 thoughts on "Raspberry Pi Secondary Memory

Interface (SMI)”

Pete

September 2, 2020 at 9:54 pm

The question on everyone’s mind is: Can this be used to drive WS2812 RGB LEDs?

★ Like

iosoftcode

September 3, 2020 at 1:05 pm

A very good question. I think the answer is yes, but I’ll do some tests and post the results.

★ Like

Pete

September 3, 2020 at 7:05 pm

Dude, if this is can drive 4 or more channels reliably, you would be a hero within the DIY Christmas Lights community! Especially if it can be scaled down to run on a Pi ZeroW…
Good luck!

★ Like

iosoftcode

September 15, 2020 at 7:37 pm

The tests are very successful; I’m driving 8 channels on a ZeroW without any problems. I just need to add a user interface, and do the writeup. Watch this space…

★ Liked by 1 person

Pete

September 16, 2020 at 2:45 pm

You rock!! Looking forward to your writeup..
Be prepared for a large influx of traffic. You may want to monetize first or get a sponsor.

★ Like

---

iosoftcode

September 29, 2020 at 8:10 pm

I’ve posted the NeoPixel project at https://iosoft.blog/raspberry-pi-multi-channel-ws2812/

★ Like

KOKO. PE.

October 29, 2020 at 3:04 pm

I am working on an fpga for raspberry pi and I already have a implemented a simple half duplex parallel communication. But that tops at 4mb/s.
After reading your post I thought to give smi a go. But there is one little problem. On my board gpio 7 aka swe/srw is used as tck line for fpga programming and cannot be used.
Now heres my question. Do you think I can use any other pin as swe/srw? I cannot change my design easily so I am looking for alternatives. I guess it would require a modified smi module for linux kernel but is it doable?

★ Like

iosoftcode

October 29, 2020 at 8:39 pm

The short answer is no; the SMI pin assignment is done in Pi hardware, so it can not be changed in software.
However, you could use another pin as a substitute by tweaking the communications between the Pi and FPGA; for example, if you always do read cycles from address 0, and write cycles to address 1, then the SA0 pin effectively becomes a read/write indicator.

★ Like

Marcel

February 17, 2021 at 5:33 pm

I was really looking for smething like this to implement an high speed interface to an FPGA to the RPI4. It is really a pity that most of the information is kept "secret" buy Broadcom and the RPI foundation. There is nothing special about a parallel bus like this, so why do not provide the documentation.

★ Like

iosoftcode

February 17, 2021 at 9:14 pm

I’m working on an FPGA project with an SMI interface, so will probably be posting something in the next month or two.

With regard the ‘secret’ information, the Pi foundation would probably publish if they could, but are legally restricted by Broadcom, and that company would need a significant commercial incentive to change the current situation.

How many more chips will they sell, if the information is released? Not many…

★ Like

Michael

May 6, 2021 at 11:17 pm

Thank you for this awesome bit of work. I have one question. I’m working on a project where I’m connecting to another system that needs to send a lot of 16bit data, and I need that system to handle the data clock because it has a lot of other stuff going on. Have you figured out any way to have the smi clocked externally?

★ Like

iosoftcode

May 7, 2021 at 8:46 am

You can trigger an SMI cycle using the external request line, however the timing of that request is important; it must be negated before the end of the

SMI cycle, otherwise a second cycle will start.
So the answer is ‘maybe’, depending on the actual timing of your external clock signal.

★ Like

Jake

May 8, 2021 at 4:52 am

I admire your wonderful research.

I am following your research with RPi4 and AD9226.
However, the AD9226 output signal goes back and forth continuously between -5 and 5. (calculated voltage value)
In my opinion, voltage should output 0 if there is no input to AD9226.
Are there any special actions I should take to stabilize the AD9226’s output signal?

★ Like

---

iosoftcode

May 8, 2021 at 8:58 am

I suggest you take a look at the AD9226 data sheet; the device measures the differential voltage between VINA and VINB; if VINB is greater than VINA, it will report a negative value. Also, if you are using the same ADC module as I did, there is an additional input circuit that shifts the measuring range to around 0V, i.e. it will report +ve and -ve voltages with respect to ground. Sadly the details of this circuit aren’t published, so I don’t know exactly what they have done.

All this is very different to a simple ADC such as the MCP3008, which can only measure positive voltages, but the AD9226 is designed to do a very different job.

★ Like

icarletto

June 9, 2021 at 10:45 am

Hi, this is a very nice work and I am now trying to replicate it, although I am focusing on the DMA passthrough mode.
I have never used a bus-interface before, therefore it would be helpful if could clarify few points:

– the lines SOE_N/SE and SWE_N/SRW seem to be both output: in my understanding the first gives the temporization for read/write operations, but what is the purpose of the second?
– at the beginning of the section "SMI and DMA" you say that the transfer takes place if the request is held low, but I actually see the two strobe pulses take place while the request is high (am I interpreting it wrong?)
– always in the same section, I guess the line REQUEST in the oscilloscope can be either pin SD16 or SD17 but just to be sure, is SOE the line STROBE?
– does the data read/write take place on the rising or falling edge of SOE?
– can you please point me at the settings I need to change to use DMA in passthrough mode?

★ Like

iosoftcode

June 9, 2021 at 7:04 pm

To address your points in turn:
1. SMI is designed to emulate a CPU data bus, and there are 2 types in common usage; those that have a strobe plus a signal that indicates if it is a read or write cycle (e.g. Motorola 68k), and those that have separate strobes for read & write (e.g. Intel 8080). That is why these signals can be configured 2 different ways, probably using mode68 bit in SMI_DSR_REG.

2. I think there is a mistake in the text; transfers occur while the request is

high.

3. Yes

4. Not sure.

5. Set dmap in SMI_DMC_REG

★ Liked by 1 person

icarletto

June 10, 2021 at 8:49 am

Thank you very much! Your reply makes things much more clear. Now this schematic image (https://i.imgur.com/Rm3vkAm.png) that I found makes sense.

Just have one last question. In the same image of the oscilloscope with Strobe and Request, the Strobe line appears to be low for 100 ns, but few lines above you declared (setup 25, strobe 50, and hold 25, since (25 + 50 + 25) * 10 = 1000 or 1 MSps), so I would expect to see the strobe low for 50*10 = 500 ns. Also, the distance between the two strobe is around 350 ns. Did you use different settings in this example?

★ Like

iosoftcode

June 10, 2021 at 8:10 pm

Yes.

★ Like

Michael

June 18, 2021 at 8:46 pm

I stumbled across the smi documentation that Gert pulled out of the broadcom datasheet. Apparently his originals were pulled at some point but this project has a copy https://github.com/cariboulabs/cariboulite/blob/main/docs/Secondary%20Memory%20Interface.pdf

★ Like

Petros

June 19, 2021 at 10:39 am

Gpio transfers can be equally fast with some using aom clever programming.
I tried the smi approach but due to lack of documentation I tried plain gpio transfers
For my thesis I designed a 16 bit parallel protocol that sends a video feed from the raspberry pi to an fpga and back.
Using no memory in between just a small fifo buffer on the fpga I was able to get over 8mb/s.
Using a multithreaded process on the pi I got close to 22mb/s.

★ Like

---

iosoftcode

June 19, 2021 at 10:51 am

I agree that you can get a good peak speeds with GPIO transfers, but there is a major disadvantage compared with SMI & DMA: the absence of FIFOs.
If your application can cope with widely-varying transfer rates, then no problem; but if you need reasonably accurate timing (e.g. regular sampling of an ADC) then the FIFOs are essential, to smooth out the remarkably inconsistent DMA transfer times.

★ Like

João Silva

July 4, 2021 at 2:53 am

Hello,
First of all thank you for this article, it really opens up a lot of possibilities and allows for simpler interface between a Pi and high speed devices, which would otherwise require something like USB or PCIe (on the RPi 4).
I have yet to test this on actual hardware, but I looked into the DMA section of the ARM Peripherals PDF you linked in your DMA article. I noticed the DMA has something called "Panic" signals, that, if I understood it correctly, is a way for a specific peripheral to tell the DMA controller it is running out of data and needs to be prioritized. This priority can be set with bits 23:20 of the DMA Control And Status register (along with the "non panic" transfer priority). I wonder if this panic feature (or even just increasing the "normal" priority) can be used to overcome the bottlenecks you saw in the Pi 3 and Pi 4, without disturbing the other bus masters. I see you do not touch the priority field when initializing the DMA channels, so it stays at 0, which is the lowest priority.
Do you still have the hardware setup? Are you able to test this?
Thank you !

★ Like

iosoftcode

July 4, 2021 at 2:50 pm

I tinkered with these values to no great effect, so I think this may just be a memory bandwidth problem. As the Pi 4 CPU is faster, with more extensive caching, it takes a greater percentage of the memory bandwidth, and if the total is over 100%, then something has to give way – and if DMA stalls, the SMI interface generates incorrect bus cycles.
However, it would be worthwhile doing some more detailed testing, to see if this is true.

★ Like

Ralf

August 2, 2021 at 2:47 pm

There is one thing that did not become clear for me. Is it possible to do continuous gapless streaming from the an ADC via SMI to chained DMA buffers? Or will there always be some minimal time with bus conflict between the transfers and gap in the clock signal?

★ Like

iosoftcode

August 2, 2021 at 6:37 pm

I haven't specifically checked this, but I think that there might be a gap of roughly 200 nanoseconds between one DMA transfer ending and the next one starting, and you are probably right in thinking that SMI might drive the bus in this time.

★ Like

Doug

August 9, 2021 at 5:57 pm

Thanks for your hard work! I'm looking to use the SMI to interface with a FPGA at high speeds. Since that will be the only way of communicating, I plan on using the address lines to talk to various registers in the FPGA. Since you only talk to DACs and ADCs, could you describe how to access the address lines? I see there is a SMA_A_REG. I think there are 6-bit address, 2-bit x, and 2-bit dev fields for this register from your code. Is this correct? I just need to set the 6-bit address at bit locations [9:4]?

Also, can you describe the control lines better? I see there are 2 modes: SOE_N with SWE_N and SE with SRW_N. How are they defined? In the first case, does SOE_N go low for reading from the FPGA into the SMI and SWE_N goes low for writing from the SMI to the FPGA?

Can you switch back and forth quickly between reading and writing?

★ Like

iosoftcode

August 10, 2021 at 8:36 am

I’m afraid I haven’t done any more work on SMI since I wrote the post, so haven’t actually used the address lines. With regard to the strobe signals, they are designed to emulate an 8080 or 68000 bus, depending on the setting of the ’68k’ bit; the main difference is whether there are separate read & write strobes, or a single strobe and a read/write line. The speed of switching between read & write cycles is just down to the software, e.g. time to load a new DMA control block, which is quite fast (around 200 ns) but as my post describes, there is always the risk of a bus clash, so I’d incorporate some series resistors to limit the fault current.

★ Like

Doug

August 23, 2021 at 6:30 pm

I am trying to run your rpi_smi_dac_test on my Raspberry Pi 4. First, I want to get this to work without DMA. I downloaded your files from GitHub.

In file rpi_dma_utils.h, I changed these lines:
#define PHYS_REG_BASE PI_4_REG_BASE
#define CLOCK_HZ 250000000 // Pi 2 – 4

In file rpi_smi_dac_test.c, I commented this line out to not use DMA:
//#define USE_DMA 1

I’m compiling like this: gcc -o rpi_smi_dac_test rpi_dma_utils.c rpi_smi_dac_test.c

I have a high speed scope connected to SOE_N, SWE_N, SD0, and SD7. I am seeing activity on these signals. I'm playing with the parameters to init_smi(), but they don't make sense. You call this here: init_smi(0, 1000, 25, 50, 25); What does the first parameter do? It's zero and is getting set to: smi_dsr->rwidth = smi_dsw->wwidth = width; It doesn't seem to have any affect. The other parameters seems to do something, but I'm not how they work. Could you explain in more detail? THANKS!

★ Like

iosoftcode

August 23, 2021 at 7:03 pm

If you look at the function definition, you'll see the first parameter is the SMI bus width, using the values required by the hardware, a value of zero being defined as SMI_8_BITS.

However, I think there is little point running SMI without DMA. With any meaningful data block size & transfer rate, the CPU won't be able to maintain continuous transfers, and as soon as it misses one, the SMI bus will start inserting dummy write cycles. My blog shows instances of that when the DMA hasn't been able to keep up, and without DMA the situation will probably be worse.

★ Like

icarletto

August 30, 2021 at 11:09 am

I finally had the time to test and learn from your code. Everything works smoothly, but there is something I do not really understand, maybe you have some hint.

I did a test with your ADC code using pull-up/down resistors to define a constant 8-bit input. I then split the instruction smi_start(NSAMPLES, 1) into two transfers with NSAMPLES/2 and a delay in between. Everything works fine, I can check with a probe that the transaction takes place in two stages and at the end the content of rxbuff is the expected one. However, if I try to read the content of rxbuff during the pause, it contains only zeros and it get fully populated only when the DMA transfer is completed. A similar behavior is observed also with external SMI DREQ: if the DREQ pulse is not long enough to accommodate the transfer of all the bytes required by the DMA, rxbuff will contain only zeros.
Maybe I get it wrong, but since the DMA is supposed to write directly to the uncached memory rxbuff, I would expect to see a partially populated rxbuff in both cases. What am I missing here?

PS. There is problem with your DAC code, the lower 2 bits are never written to the output. The output buffer is populated using a uint8_t* pointer but data are not packed (pxldat = 0), so actually only one byte every four is read from the buffer. You can either copy the data using a uint32_t* pointer and copy/transfer NSAMPLES*4, or set pxldat = 1, but then you should reorder the samples to account for the RGB565 ordering.

★ Like

iosoftcode

August 30, 2021 at 1:39 pm

Sadly I don't have time to replicate this issue, and the symptoms as described don't make much sense. Are you sure you're waiting long enough for the DMA to have started? It does take some time for the new control block to be processed, and the FIFO to be filled.

If a lengthy data transfer fails in an all-or-nothing way, then there must be a failure in the DMA or SMI setup (possibly time-delay-related) that is triggered when you modify the code. The key question is whether DMA is really writing a lot of zeros when failing; have you pre-loaded the memory with a different value, and made sure that this preload value has been transferred from the CPU cache to physical memory before running the test? If DMA is really writing a complete block of zeros, the fault is more likely to be on the SMI side.

I realise this is not much help, but it'd need some serious testing to unearth the real cause.

★ Like

icarletto

September 7, 2021 at 11:37 am

I am sure that the DMA has started and that no error occurred. I did some more extended test and it looks like the reason why I observe only zeros is because the data have still not been transferred from the FIFO to the uncached memory. Apparently, on long data transfers (>16 bytes), data are copied to the memory only in multiples of 16 bytes (eg. for a 32 bytes transfer, the first 16 are transferred to the memory only when the 17th byte is acquired).

I first though it might depend on how the SMI threshold/panic values pace the transfer, but it looks like the SMI is not responsible for this behavior. Infact, although inside the buffer I can read only zeros in the middle of a transfer, the SMI FIFO Level in the debug register report zero, therefore the SMI interface does not contain any more the acquired data. By exclusion, if data are not in the buffer, nor in the SMI FIFO, I guess they can only be inside the DMA FIFO because the DMA transfer is not complete. Do you think it is a correct analysis or there might be something else I am not considering?

The reason why I am interested in this is because I want to start one DMA block with source from SMI to acquire N samples with external DREQ pacing. The data acquisition mechanism works properly and I can acquire all the data, but I also want to read/process batches of data in the buffer while they are acquired. The problem is that some samples at the end of the batch are zeros even if they have already been acquired. It would work perfectly if I instead of executing one CB with length N samples, I could iterate N times a CB with length 1 sample, but in this case I would loose the automatic mechanism for the increment of the destination memory address. Do you see a better approach?

★ Like

---

iosoftcode

September 10, 2021 at 6:32 pm

I can understand that part of the data may be in the FIFO; it is flushed when the transfer is complete, so there may well be some data in it when the transfer is incomplete.

Sadly I have no real solution to the 'N times a CB with length 1 sample' problem you describe; I have thought of a few workarounds (such as writing to the Tx of a looped-back UART, and reading the Rx with another DMA channel) but these are very messy, and I'm not sure they'll actually solve the problem.

However, thinking about UART Rx DMA, the symptoms you describe would be a real problem in, say, a console interface, so either the UART has some magic method of forcing the partial DMA to go through, or maybe you just can't use DMA on random-length UART transfers – I'd look at the driver source code if I had time.

★ Like

Harry

November 29, 2021 at 6:20 pm

Great blog post! Learned about smi from the cariboulite that's on crowd supply- it's criminal that this interface isnt more documented, seems really

useful for extending the Pi's hardware capability
How difficult do you think it would be to use e.g a Flashy ADC board https://www.knjn.com/Flashy.html to make an oscilloscope with this? and, of course, how fast could you go? It seems to me you could certainly configure a 100Mhz clock- but something gets in the way of that clearly since you only got ~30MSPS. Would it be any different on bare metal/ with as little as possible other stuff running?

★ Like

iosoftcode

November 30, 2021 at 4:18 pm

You might be able to use a Flashy ADC, but it is difficult to say since I couldn't find a circuit diagram when doing a quick skim of their site.
When building hardware, do bear in mind that high-speed data requires high-quality signals; all wires should be kept short, especially the supply, ground & clock signals.

You might gain some speed if running bare-metal code, but that is quite a bit of work, and I'm not sure the improvement will be dramatic; the main limitation is the memory bandwidth, and if the CPU cache is being refreshed, DMA accesses will be stalled, and you are reliant on the (relatively small) FIFOs to buffer the incoming data.

In the end, I think the name says it all: Secondary Memory Interface. It was originally designed for accessing memory, which can cope with intermittent burst operation, rather than streaming I/O, which can't tolerate interruptions.

★ Like

sulu98

January 22, 2022 at 6:54 pm

If address lines aren't used are the GPIO pins available for other functions? I only need the data line 0-7 and the SOE/SWE lines (8080 mode).

★ Like

iosoftcode

January 25, 2022 at 5:43 pm

Yes, you can use the lines as general purpose I/O

★ Like

Graeme

February 23, 2022 at 12:34 am

Hi there, thank you for this amazing tutorial. I was just wondering if you might be able to share your DMA Passthrough mode example code on your Github page. In particular, I am wondering if you needed to use separate control blocks for the input and output bus activity. I find this a bit confusing because DMA Passthrough mode's input/output bus activity is governed by an external peripheral and so I am not sure how the transition between the input and output control block would work. If DMA Passthrough only uses one control block, do you know what we should set srce_ad and dest_ad to? I am working on a project where I am using a Raspberry Pi to communicate digital audio samples and pixel RGB data to an old video game console (output) and reading controller button press data from the console back to the Pi (input). I believe the DMA Passthrough mode 80 should fit like a glove nicely over this process, but I am trying to make sure that I understand how this will work within the framework of your sample code. Thank you very much.

★ Like

iosoftcode

February 24, 2022 at 9:44 am

I’m afraid I don’t have the code; it was just a quick test. Each SMI transfer is uni-directional, so you have to set the source & destination addresses for the specific transfer direction; every time you turn the link around, you’ll have to load a new descriptor with the addresses for that direction.

★ Like