Analog IO board v1.0

May 23, 2022

1 Overview

The analog IO board v1.0 has two 4-channel 16-bit DACs(AD5764), one 4-channel 16-bit ADC(AD7386), and a Teensy 4.1 board. It has extra 2×4 general digital/analog IO pins directly to Teensy. Since the manipulation of IO pins with Teensy can be found elsewhere, this document would focus how to control DAC and ADC.

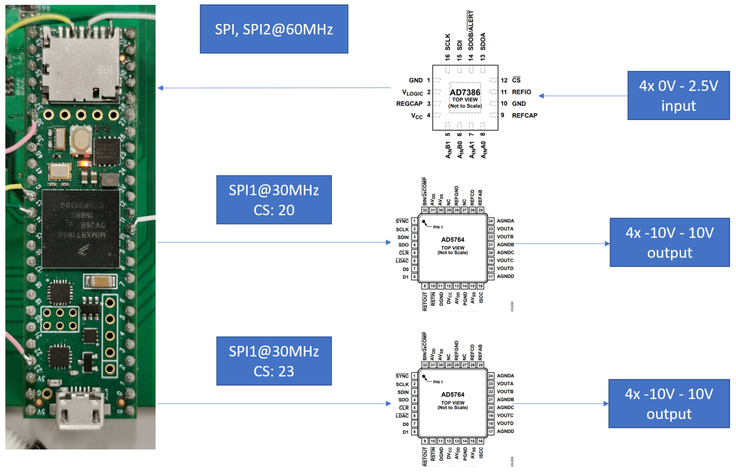


Figure 1: Simplified schematic(no buffers, powers, etc.)

Device	SCLK frequency/ MHz	Bits per transaction	Throughput/ MSPS		
			Theory	Datasheet	Measured
ADC	60*	16	3.75	4.00	3.44†, 5.22§
DAC	30	24	1.25	1.12**	0.99†, 2.24§

Table 1: Summary of throughput. *: default SPI setup supports up to 60 MHz, but the chip can work at 80 MHz; **: calculated from datasheet by adding minimum $\overline{\text{SYNC}}$ time and total transaction time; †, §: see Sec. 4

2 Quick setup

As an example, connect ADC AIN0 to the source(10 kHz square wave). The following code reads four input voltages from ADC, updates DAC channel 2 with voltage from AIN0. and prints four readings to PC through serial communication. On the oscilloscope, CH1(orange) is connected to the source, and CH4(green) is connected to DAC output. The circuit now works as a unity-gain voltage follower.

In this sample code, we identify three components.

Initialization To use ADC and DAC, include `init_chips.hpp`, `read.hpp`, `write.hpp` in the beginning of main Arduino sketch(line 1–3), and invoke `init_chips` function in the `setup` part(line 8).

Read The lastest reading of ADC are stored in the variable of respective channel(`ain0`, `ain1`, `bin0`, `bin1`) and can be used directly(line 12, 13). No other action is needed.

Write To write DAC, call `write`(line 12). The function definition is

`void write(uint8_t ch, uint16_t num)`,

where `ch` is the channel(0–7) and `num` is the DAC number(0-65535).

Table 2: Conversion from physical channel to ch number.

DAC	DAC channel	ch number
IC4	A	0
	B	1
	C	2
	D	3
IC8	A	4
	B	5
	C	6
	D	7

```
1  #include "init_chips.hpp"
2  #include "read.hpp"
3  #include "write.hpp"
4
5  void setup() {
6      while (!Serial);
7      Serial.begin(115200);
8      init_chips();
9  }
10
11 void loop() {
12     write(2, ain0 >> 3);
13     Serial.printf("%u %u %u %u\n", ain0, ain1, bin0, bin1);
14 }
```

main.cpp

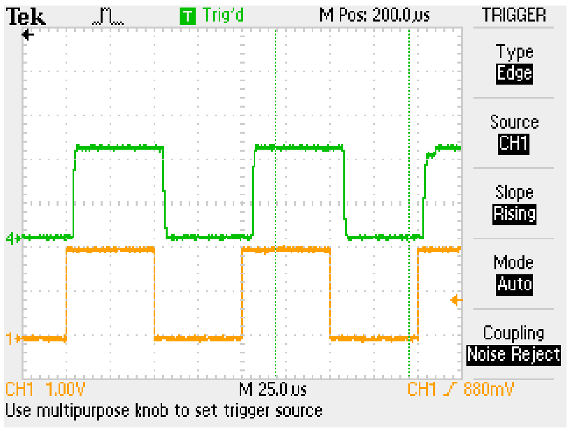
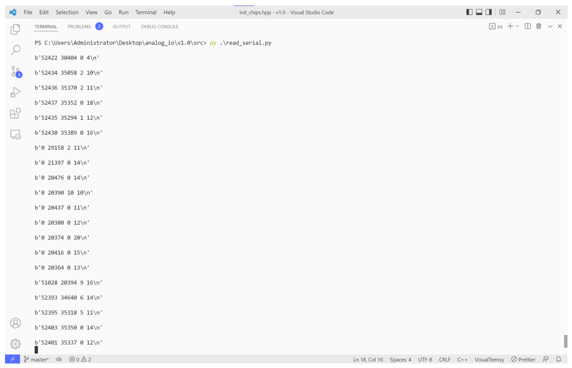


Figure 2: Oscilloscope reading


File Edit Selection View Go Run Terminal Help
init_chips.hpp - v1.0 - Visual Studio Code
TERMINAL PROBLEMS 2 OUTPUT DEBUG CONSOLE
master* 0 2

```
PS C:\Users\Administrator\Desktop\analog_io\v1.0\src> py .\read_serial.py
b'52422 30404 0 4\n'
b'52434 35058 2 10\n'
b'52436 35370 2 11\n'
b'52437 35352 0 18\n'
b'52435 35294 1 12\n'
b'52430 35389 0 16\n'
b'0 29158 2 11\n'
b'0 21397 0 14\n'
b'0 20476 0 14\n'
b'0 20390 10 10\n'
b'0 20437 0 11\n'
b'0 20380 0 12\n'
b'0 20374 0 20\n'
b'0 20416 0 15\n'
b'0 20364 0 13\n'
b'51028 20394 9 16\n'
b'52393 34640 6 14\n'
b'52395 35318 5 11\n'
b'52403 35350 0 14\n'
b'52401 35337 0 12\n'
```

Figure 3: Serial monitor reading.

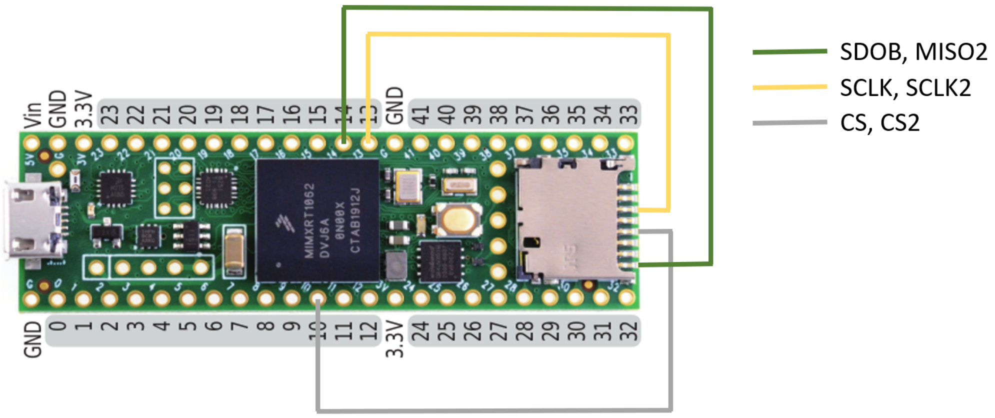


Figure 4: Configuring SPI2 as slave for reading SDOB output from ADC.

3 Modification for ADC

Since ADC yields highest throughput when its two output channels are read simultaneously, we need to configure Teensy such that it can read both output lines(SDOA and SDOB). The modification is done from both hardware and software sides:

Hardware Solder extra wire as shown in Fig. 4. This will route SDOB signal to the MISO line of SPI2 instance of Teensy. Since SPI2 should work synchronously with main SPI, we also connect CS pins and SCLK pins together.

Software The SPI2 instance is configured as a SPI *slave* by manipulating the configuration bits of the SPI module. Refer to later section for detail.

By default, only `ain0` and `bin0` are available(`ADC_CH0_ON`). To change this behavior, define the corresponding macro in `init_chips.hpp`.

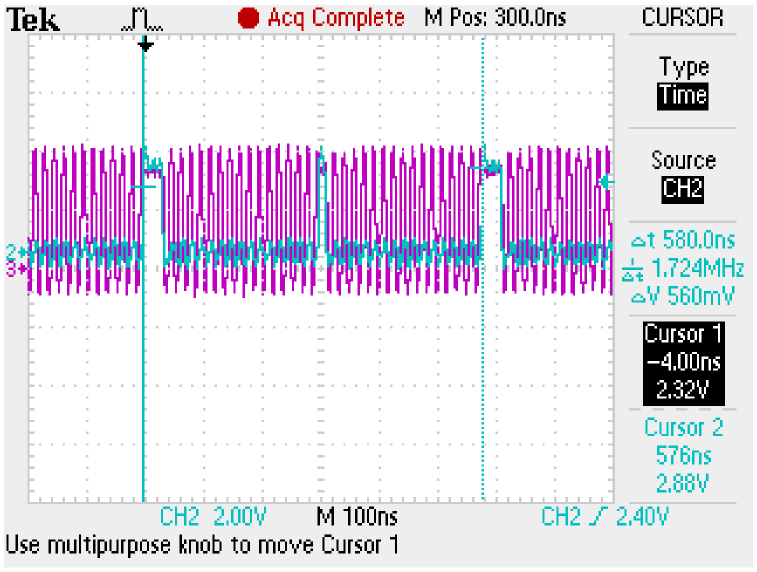


Figure 5: Measurement of ADC throughput. The clock(purple) and chip select(blue) signals are perfectly syncrhonized.

4 Benchmarking throughput

4.1 ADC

The SPI transaction of ADC is solely controlled by DMA controller, which is independent from CPU(indeed, even if CPU is "stopped" with `delay` function, ADC reading is still progressing). There're two ways to measure the throughput of ADC reading:

1. Attach an interrupt at the end of each transaction, set up a counter that increments each time the interrupt is requested and handled, and a timer(an `elapsedMicro` instance, say)that counts one second. (Uncomment the `COUNT_SAMPLE_RATE` macro definition in `init_chips.hpp`)

2. Directly measure the period of transaction on oscilloscope(Fig. 5).

The results are consistent(~ 3.45 MSPS) with each other, and each transaction cycle needs roughly 17.5 clock cycles, in agreement with the CCR register of LPSPI module.

4.2 DAC

Again, there're two ways to measure the throughput of DAC. The first is to set up a timer and do the count from the program. The result is 969 kSPS.

What's of more interest is the second method of physical origin. The waveform of DAC output at $f_{\text{drive}} = 500\,\text{kHz}$ sine wave input to ADC is shown in Fig. 6. A clear beat pattern can be seen. If we continuously increase the frequency of the drive signal f_{drive}, the beat frequency f_{beat} decreases.

By Nyquist theorem,

$$f_{\text{beat}} = f_{\text{drive}} - N f_{\text{sample}}, \quad N \in \{1, \ldots\}. \tag{1}$$

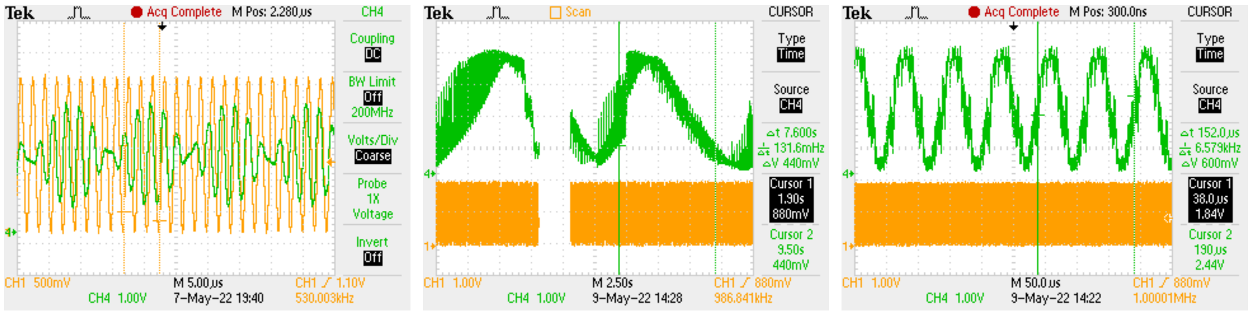


Figure 6: Beat signal at different drive frequency f_{drive}. Left: $f_{\text{drive}} = 530\,\text{kHz}$; middle: $f_{\text{drive}} = 986.8343\,\text{kHz} = f_{\text{sample}}$; right: $f_{\text{drive}} = 1000\,\text{kHz}$. Note the huge timescale difference between figures.

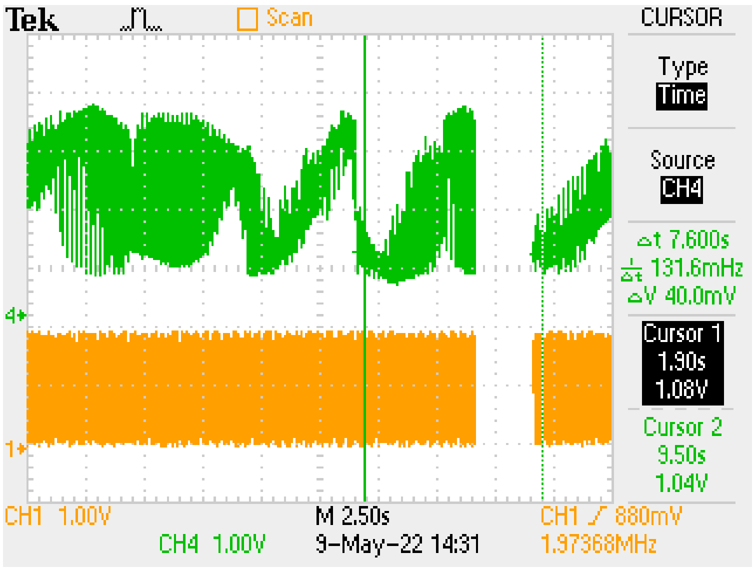


Figure 7: Beat signal at $f_{\text{drive}} \approx 2f_{\text{sample}}$. The same slow beat signal can be observed at $N = 2$.

For the case shown in Fig. 6 we should take $N = 1$. And the sampling frequency is $f_{\text{sample}} = 986.8343\,\text{kSPS}$. This method is significantly more accurate than the counting method. Also note that this beat method yields slightly higher throughput, this is because the timer and counter exerts small overhead to the system. Again, we see when the transactions are handled by CPU, every irrelevant intruction CPU executes takes away the throughput of SPI.

If we let $f_{\text{drive}} \approx 2f_{\text{sample}}$, strong beat pattern can also be observed(Fig. 7). In fact, if the DAC sampling can be triggered periodically(feasible with DMA, see below), the ADC-DAC system is effectively a mixer. And when integrated with an on-board low-pass filter in the next generation, the board can work as a demodulator(if we can also remove the phase jittering).

Table 3: Relation between internal `IMXRT_LPSPI_t` object and `SPIClass` object

`IMXRT_LPSPI_t` object	`SPIClass` object
`IMXRT_LPSPI4_S`	`SPI`
`IMXRT_LPSPI3_S`	`SPI1`
`IMXRT_LPSPI2_S`	–
`IMXRT_LPSPI1_S`	`SPI2`

5 Implementation

Since the advanced usage of SPI, DMA, and clocking is only completely documented in the iMX.RT manual(some are scattered in PJRC forum) and the jargon seems esoteric at first read(partly due to the terrible order of contents in the manual), some comments can be useful. The manual covers SPI in chapter 48(some pin setup in chapter 11), and DMA in chapter 6(some DMA source in chapter 5). The idea is to provide minimal working examples of each technique, while the manual(so does the one for Arduino Nano) provides more hacky features.

5.1 SPI

In the world of Arduino: SPI is disguisingly easy to use, calling `begin()`, `beginTransaction()`, and `transfer()` solves almost every problem. The simplicity reflects the success of the API design but also masks away finer features that the hardware provides. Since we know our chip better than the library writer does, we could leverage the knowledge for better performance.

The following discussion applies to i.MX RT1060 chip only, but the features can also be traced in 8-bit ATemega328P processor that drives Arduino Nano.

5.1.1 SPI devices

Teensy has three `SPIClass` objects available to users(`SPI`, `SPI1`, `SPI2`), while i.MX RT chip has four SPI devices(LPSPI1, LPSPI2, LPSPI3,LPSPI4). There's a relation between them(Tab. 3). Note that `IMXRT_LPSPI2_S` object is not accessible.

5.2 Setup

We start our investigation of SPI by looking at the source code. In line 2, we define a `IMXRT_LPSPI_t` object `spi_regs` that saves all the registers related with `SPI1`. To make transactions more efficient than the standard implementation, we change the clock configuration register(`CCR`) and transmit command register(`TCR`) from the default behavior.

```cpp
3 static void prepare_fast_spi_transfer24() {
4     IMXRT_LPSPI_t* spi_regs = &IMXRT_LPSPI3_S;
5     spi_regs -> CCR = (spi_regs -> CCR & 0xff);
6     uint32_t tcr = spi_regs -> TCR;
7     spi_regs -> TCR = (tcr & 0xfffff000) | LPSPI_TCR_FRAMESZ(23);
8 }
```

init_chip.cpp

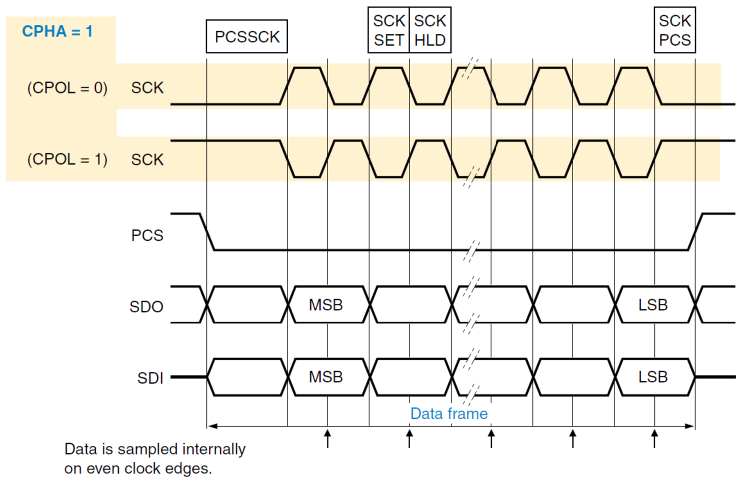


Figure 8: Definition of fields in `CCR`.

5.2.1 CCR

The 32-bit CCR is defined as a combination of four 8-bit fields: SCKPCS, PCSSCK, DBT, SCKDIV. Fig. 8 shows the definition of the first two fields, which are just delays. They get non-zero values when `SPI.begin()` is called and are cleared in line 3.

```
// calculates div
_ccr = LPSPI_CCR_SCKDIV(div) | LPSPI_CCR_DBT(div/2) | LPSPI_CCR_PCSSCK(div/2);
// saves _ccr to CCR
```
SPI.h

5.2.2 TCR

The 32-bit TCR is more interesting. On line 4, we first save a copy of TCR in `tcr`; and on line 5, we change the FRAMESZ field, save the updated value back to register. The FRAMESZ field says how many bits every transaction needs. Since the DAC chip needs 24 bits per transaction, we set this number to $24 - 1 = 23$ from the beginning. In fact, this is one major optimization from the standard library: before we can only do a 8-bit plus 16-bit transaction, but now a single 24-bit transaction is all we need.

```
uint16_t transfer16(uint16_t data) {
    uint32_t tcr = port().TCR;
    port().TCR = (tcr & 0xfffff000) | LPSPI_TCR_FRAMESZ(15); // turn on 16 bit mode
    port().TDR = data; // output 16 bit data.
    while ((port().RSR & LPSPI_RSR_RXEMPTY)) ; // wait while the RSR fifo is empty...
    port().TCR = tcr; // restore back
    return port().RDR;
}
```
SPI.h

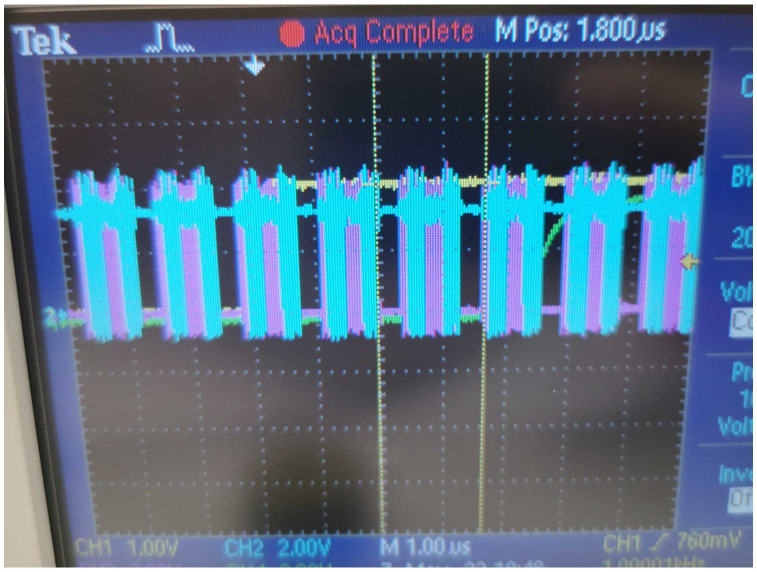


Figure 9: Different SPI devices can transmit independently(the fundamental clock is the same, but it's scaled and divided in each device)

5.2.3 TDR, RSR, RDR

Three registers are of interest to SPI transactions. The transaction starts with writing the transfer data to the 32-bit transfer data register(`TDR`). Next, we wait until data is received. When it does, the RXEMPTY field in receive status register(`RSR`) will be cleared. Finally, we can read from the receive data register(`RDR`) to get the SDO.

At first sight, reading `RDR` at the end of transaction seems unnecessary if SDO is not used. But the fact is that both transmit data and receive data are stored in a queue. The queue has a finite size(16 $\times$ 32-bit words each), and overflow would result in error.

Thus, writing to `TDR` merely pushes data into the transmit data queue and the CPU isn't blocked when the LPSPI device is doing transactions. This means we can overlay transactions of different SPI devices in time, as the following example shows. Here when SPI1 transfer once, two SPI transfers take place. When this function is put in `loop` function, the clock of `SPI`(purple) and `SPI2`(blue) looks like Fig. 9.

```
int cycle(uint16_t num) {
    IMXRT_LPSPI_t* spi_regs  = &IMXRT_LPSPI3_S; // SPI1, driving DAC, slower
    IMXRT_LPSPI_t* spim_regs = &IMXRT_LPSPI4_S; // SPI, drive ADC, faster
    spi_regs->TDR = ((((uint32_t)((2 & 3) | DAC_DATA_REG)) << 16)+ (uint32_t)num);
    spim_regs ->TDR = 0;
    while ((spim_regs->RSR & LPSPI_RSR_RXEMPTY));
    spim_regs ->TDR = 0;
    spim_regs -> RDR;
    while ((spim_regs->RSR & LPSPI_RSR_RXEMPTY));
    int ret = spim_regs -> RDR;
    while ((spi_regs->RSR & LPSPI_RSR_RXEMPTY));
    spi_regs -> RDR;
    return ret;
}
```

cycle.cpp

```
static void transfer_dac24(uint32_t data) {
    spi_regs->TDR = data;
    while ((spi_regs->RSR & LPSPI_RSR_RXEMPTY));
    spi_regs -> RDR;
}
```

write.cpp

5.2.4 CFGR1 and SPI slave mode

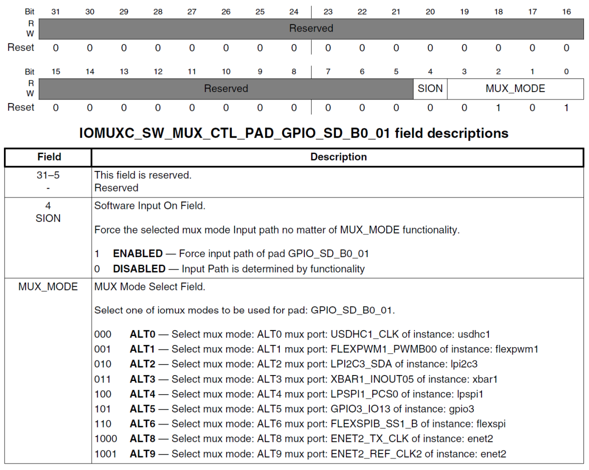
When `SPIClass` object is initialized with `begin()` function, CFGR1 register gets its initial value as a master. To make the device work as slave, we need to clear the bit(line 40).

The full setup is as follows. Here `SPIS` is `SPI2`, `ChipSelectSlave` is `44`, `spis_regs` is `IMXRT_LPSPI1_S`. On line 34, `setCS` function will set pin 44 as the the chip select pin(PCS0) of LPSPI1. To see how this works, first note that each pin is connected to more than one module inside the chip. Which way the signal goes to is controlled by a multiplexer(IOMUXC), or a multichannel switch, and the switch has a default position. Fig. 10 shows how the signal from pin 44 can be routed inside the chip. By default it is a GPIO pin in ALT5 mode and you can use, say, `digitalWrite` to control it; invoking `setCS` will automatically route pin 44 in ALT4 mode and the pin will serve as PCS. Now the pin **won't** respond to `digitalWrite`, but the LPSPI module will **automatically** take care of the edges needed for successful transaction.

```
31 void initSPISlave(uint8_t dataMode) {
32     SPIS.begin();
33     SPIS.setCS(ChipSelectSlave);
34 
35     uint32_t tcr = LPSPI_TCR_FRAMESZ(15);
36     if (dataMode & 0x08) tcr |= LPSPI_TCR_CPOL;
37     if (dataMode & 0x04) tcr |= LPSPI_TCR_CPHA;
38     spis_regs->TCR = tcr;
39 
40     spis_regs->CFGR1 = 0;
41     spis_regs->DER   = LPSPI_DER_RDDE;
42     spis_regs->CR    = LPSPI_CR_MEN;
43     initSPISlaveDMA();
44 }
```

SPISlave.cpp

To get a closer look of the function, the `hardware()` will return a bunch of useful info of the LPSPI module: the `cs_pin` attribute is an array of pin numbers, among which is 44; `cs_mux` contains the ALT mode needed for each entry in `cs_pin` to funcion as PCS, namely 4 in our case;



Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

IOMUXC_SW_MUX_CTL_PAD_GPIO_SD_B0_01 field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad GPIO_SD_B0_01 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	MUX Mode Select Field. Select one of iomux modes to be used for pad: GPIO_SD_B0_01. 000 **ALT0** — Select mux mode: ALT0 mux port: USDHC1_CLK of instance: usdhc1 001 **ALT1** — Select mux mode: ALT1 mux port: FLEXPWM1_PWMB00 of instance: flexpwm1 010 **ALT2** — Select mux mode: ALT2 mux port: LPI2C3_SDA of instance: lpi2c3 011 **ALT3** — Select mux mode: ALT3 mux port: XBAR1_INOUT05 of instance: xbar1 100 **ALT4** — Select mux mode: ALT4 mux port: LPSPI1_PCS0 of instance: lpspi1 101 **ALT5** — Select mux mode: ALT5 mux port: GPIO3_IO13 of instance: gpio3 110 **ALT6** — Select mux mode: ALT6 mux port: FLEXSPIB_SS1_B of instance: flexspi 1000 **ALT8** — Select mux mode: ALT8 mux port: ENET2_TX_CLK of instance: enet2 1001 **ALT9** — Select mux mode: ALT9 mux port: ENET2_REF_CLK2 of instance: enet2

Figure 10: The mux table for pin 44(GPIO_SD_B0_01). It supports multiple feature and by default it works as a GPIO pin.

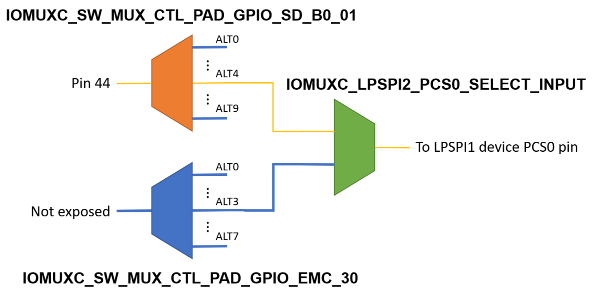

IOMUXC_SW_MUX_CTL_PAD_GPIO_SD_B0_01
ALT0
ALT4
ALT9
Pin 44
IOMUXC_LPSPI2_PCS0_SELECT_INPUT
To LPSPI1 device PCS0 pin
ALT0
ALT3
ALT7
Not exposed
IOMUXC_SW_MUX_CTL_PAD_GPIO_EMC_30

Figure 11: The `setCS` function first route pin 44 to ALT4 mode(line 1402, orange muxer), and route the input for PCS0 of LPSPI2 as pin 44(line 1404, green muxer). In this way the LPSPI2 takes control of pin 44 just as normal `SPI` takes control of MOSI(pin 11), MISO(pin 12), and SCLK(pin 13) pins.

`pcs_select_input_register` and `pcs_select_val` controls another mux for PCS0 ; `cs_mask` is just an identifier of the pin.

```
1399 uint8_t SPIClass::setCS(uint8_t pin) {
1400     for (unsigned int i = 0; i < sizeof(hardware().cs_pin); i++) {
1401         if (pin == hardware().cs_pin[i]) {
1402             *(portConfigRegister(pin)) = hardware().cs_mux[i];
1403             if (hardware().pcs_select_input_register[i])
1404                 *hardware().pcs_select_input_register[i] = hardware().pcs_select_val[i];
1405             return hardware().cs_mask[i];
1406         }
1407     }
1408     return 0;
1409 }
```

SPI.cpp

5.3 DMA

Direct memory access(DMA) is a technique that allows peripheral deivce to read/write main memory directly, bypassing the possible copies to/from CPU. The technique is implemented through a device called DMA controller, and on the i.MX RT1060 chip this is the enhanced DMA(eDMA) controller. This bypass greatly reduces the load of CPU.

32 DMA channels are available on Teensy, and each, once started, copies data from one part of the memory to another part according to certain rules. Typical components of a channel are:

Trigger When to start a channel: a channel can start manually, periodically, from certain request(e.g., SPI device receives data), or even at the start/end of another channel;

Address The start/end address of source and destination, and rules of copying(e.g. from the received SPI data in `RDR` to `ain0` variable);

Interrupt(optional) At the end or halfway of transfer, the eDMA controller can let CPU knows and perform certain instructions(e.g. increment the counting variable).

As an example, when the ADC sequencer mode is on, the `SPI2`(defined on line 1) would receive the ADC readings of BIN0 and BIN1 alternatingly. Note that in line 2, the address of variable `bin0` and `bin1` are adjacent to each other(i.e. `&bin1-&bin0=1`). Finally, we define `DMAChannel` object `rx` without initialization.

In function `initSPISlaveDMA`, we first initialize the channel by calling `rx.begin(true)`, which will set up reasonable initial value of the channel. In line 7, we set up the trigger of the channel, meaning to initiate a transfer on this channel every time the controller receives a signal that LPSPI1 should receive data.

In line 9, 10, we specify the source and destination of the DMA transfer. Since we want to transfer data continuously out from receive queue, the source is just `RDR`. The destination setup is tricky in that we want it to alternate between `bin0` and `bin1`:

Line 10 set up the initial address to `&bin0`;

Line 12 every cycle consists of 2 copies;

Line 13 after each copy, the destination address offset by +4, i.e. the address of `bin1`;

Line 15 after each cycle, destination address is offset by -8, effectively going back to `&bin0`.

Note that the "cycle" is the same as "major loop" as in Fig. 12; and "copy" means both single "transfer" and "minor loop".

```
1  static IMXRT_LPSPI_t* spis_regs    = &IMXRT_LPSPI1_S;
2  volatile uint32_t bin0, bin1;
3  static DMAChannel rx(false);
4  static void initSPISlaveDMA() {
5      rx.begin(true);
6      // trigger
7      rx.triggerAtHardwareEvent(DMAMUX_SOURCE_LPSPI1_RX);
8      // address
9      rx.source(spis_regs->RDR);
10     rx.destination(bin0);
11     // rules
12     rx.transferCount(2);
13     rx.TCD->DOFF     = 4;
14     rx.TCD->DLASTSGA = -8;
```

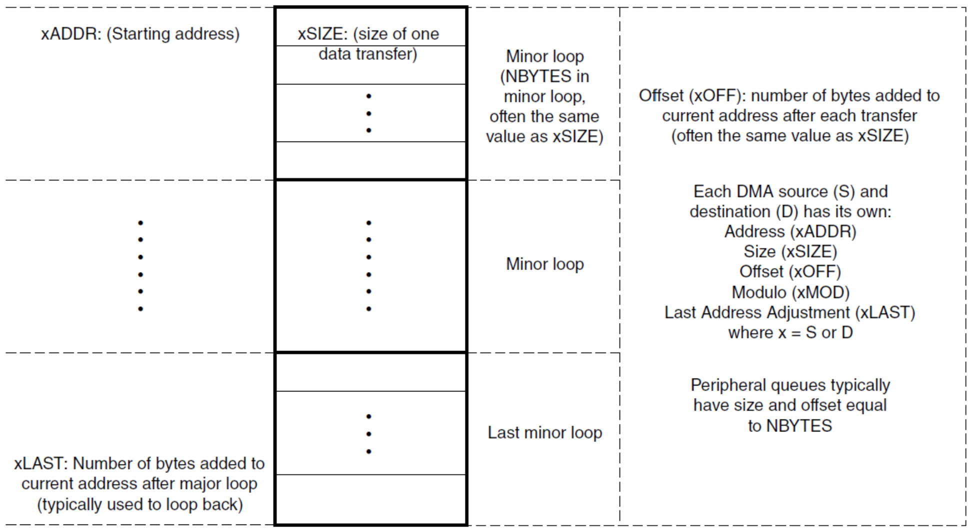

xADDR: (Starting address)
xSIZE: (size of one data transfer)
Minor loop (NBYTES in minor loop, often the same value as xSIZE)
Offset (xOFF): number of bytes added to current address after each transfer (often the same value as xSIZE)
Each DMA source (S) and destination (D) has its own:
Address (xADDR)
Size (xSIZE)
Offset (xOFF)
Modulo (xMOD)
Last Address Adjustment (xLAST)
where x = S or D
Minor loop
Peripheral queues typically have size and offset equal to NBYTES
Last minor loop
xLAST: Number of bytes added to current address after major loop (typically used to loop back)

Figure 12: Terms involved in DMA C programming. They control precisely how data moves and appear as register names of the transfer control descriptor(TCD)

```
    rx.enable();
}
```

SPISlave.cpp

In this way, every time we need the ADC reading from channel B, read the variable `bin0`, `bin1` suffices. Similarly we can let DMA handle the transmit and reading from channel A. But note that this way the whole transaction is syncrhonized to the LPSPI functional clock but not bus clock so the traditional way of `digitalWrite(CS, HIGH/LOW)` at the end/start of transaction generally won't work and `setCS` must be called.

5.4 Overclocking

The term "overclocking" in general refers to the practice of increasing the clock frequency above the manufacturer specification. In our case, this means to drive SPI transaction at higher clock frequency. But the story turns out to be more complicated than naïvely instantiating an SPI transaction with `SPISettings(WHATEVER_HIGH_FCLK, ...)`.

5.4.1 Pin driver

Standard SPI library supports ADC clock frequency at 60 MHz, and any attempt to drive at 80 MHz would result in transmission failure. The waveform from clock pin looks like strong overshoot, leading to the false conclusion that the PCB is of poor layout. But the fact is that attached to each pad that Teensy exposes there's a driver with variable impedance, and the overshoot is due to the impedance mismatch. By default, invoking `pinMode(..., OUTPUT)` would only change the direction of dataflow but not the impedance. The driver setup is summarised in Tab. 4.

Unfortunately, the default setup isn't satisfactory for our purpose, since the pin is expected to flip fast and match the impedance of the PCB trace(or BNC cable for a test setup).

Table 4: Pin driver setup. Changing the drive strength improves impedance match.

Field	Default	This work	Description
SPEED	2	3	Speed
DSE	6	4	Drive strength
SRE	0	1	Slew rate

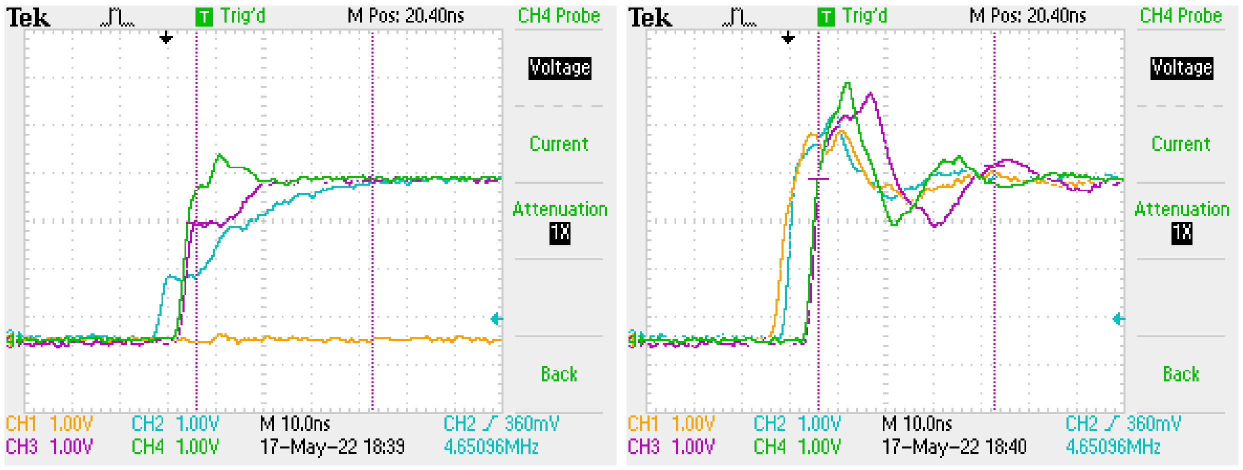


Figure 13: Short time response with different DSE field. On the left, DSE fields are: 0(CH1), 1(CH2), 2(CH3), 3(CH4); on the right: 4(CH1), 5(CH2), 6(CH3), 7(CH4).

So we set SPEED field to 3 and set the slew rate bit. Next, DSE field is varied and the short time response is summarised in Fig. 13. Evidently, the larger DSE field is, the faster output changes, so does the overshoot get stronger. In our implementation, the SPI slave pin needs to work with `DSE=4`. Larger overshoot is not acceptable for high speed purpose. Fig. 14 compares the new setup against default.

Changing pin driver is done with custom function `void set_fastio_pin(uint8_t)`, the implementation is as follows. Here, `portControlRegister(pin)` is macro function that expands to the `IOMUXC_SW_PAD_CTL_PAD_GPIO_XX` register corresponding to pin `pin`(replace `XX` with the native name, e.g. pin 13 to B0_03)

```
#define FAST_IO IOMUXC_PAD_DSE(4) | IOMUXC_PAD_SPEED(3) | IOMUXC_PAD_SRE
void set_fastio_pin(uint8_t pin_num) {
        *(portControlRegister(pin_num)) = FAST_IO;
}
```

init_chips.cpp

5.4.2 Clock generation

In Sec. 5.2.1, the CCR of LPSPI module is introduced. It's natural to ask from where the SCLK signal comes, and how the factor `div` is calculated in `SPI.h`. Fig. 15 shows an excerpted clock tree of the i.MX RT chip. Of interest to LPSPI module is the `LPSPI_CLK_ROOT` signal. We see this root clock comes from `CBCMR[LPSPI_CLK_SEL]` and gets divided by

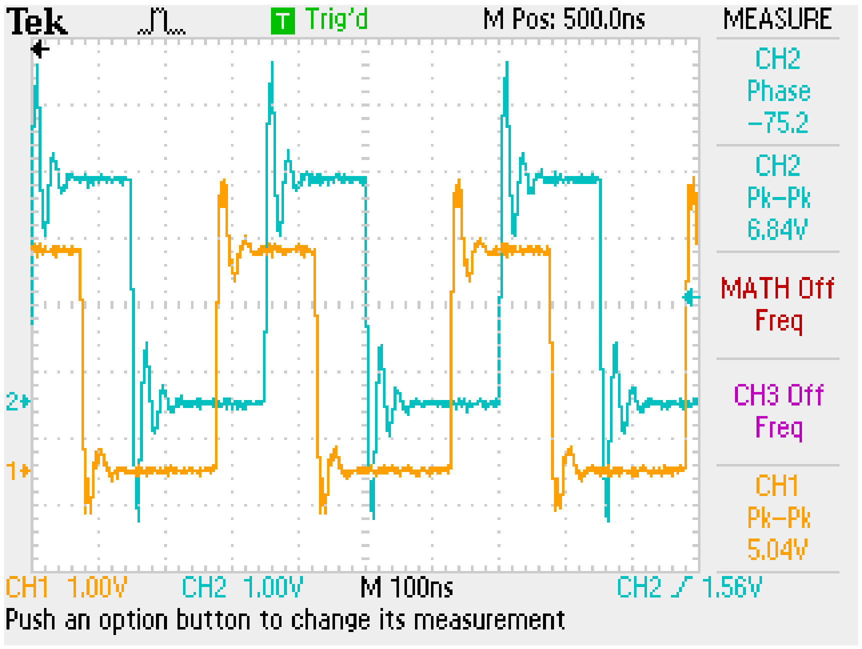


Figure 14: Comparison of the short time response between new and default setup. Blue curve is default and orange curve is our setup. The overshoot is mitigated.

`CBCMR[LPSPI_PODF]`. In `SPIMaster.cpp`, we change this clock to the max frequency the chip provides, bacuase setting a higher frequency allows more precise control and finer tuning of the SPI transaction.

```
CCM_CBCMR = (CCM_CBCMR & ~(CCM_CBCMR_LPSPI_PODF_MASK | CCM_CBCMR_LPSPI_CLK_SEL_MASK)) |
    CCM_CBCMR_LPSPI_PODF(0) | CCM_CBCMR_LPSPI_CLK_SEL(1);
```

SPIMaster.cpp

Values	Default	This work
`CBCMR[LPSPI_CLK_SEL]`	0	1
`CBCMR[LPSPI_PODF]`	5	0
`LPSPI_CLK_ROOT` / MHz	111	720

Similarly, we change `IPG_PODF` to 1(a divisor of 2 to core ARM frequency). This would enable faster DMA module. Note that for this important bus clock I currently find no way to set the divider to 1.

5.4.3 Result

After applying the above techniques we achieve shorter latency. In Fig. 16, the latency drops from 3 μs to 1.5 μs. Notable features from Bode plot(Fig. 17) are :

1. Precise measurement from Bode plot shows that the latency actually decrease from 3.4 μs to 1.3 μs.

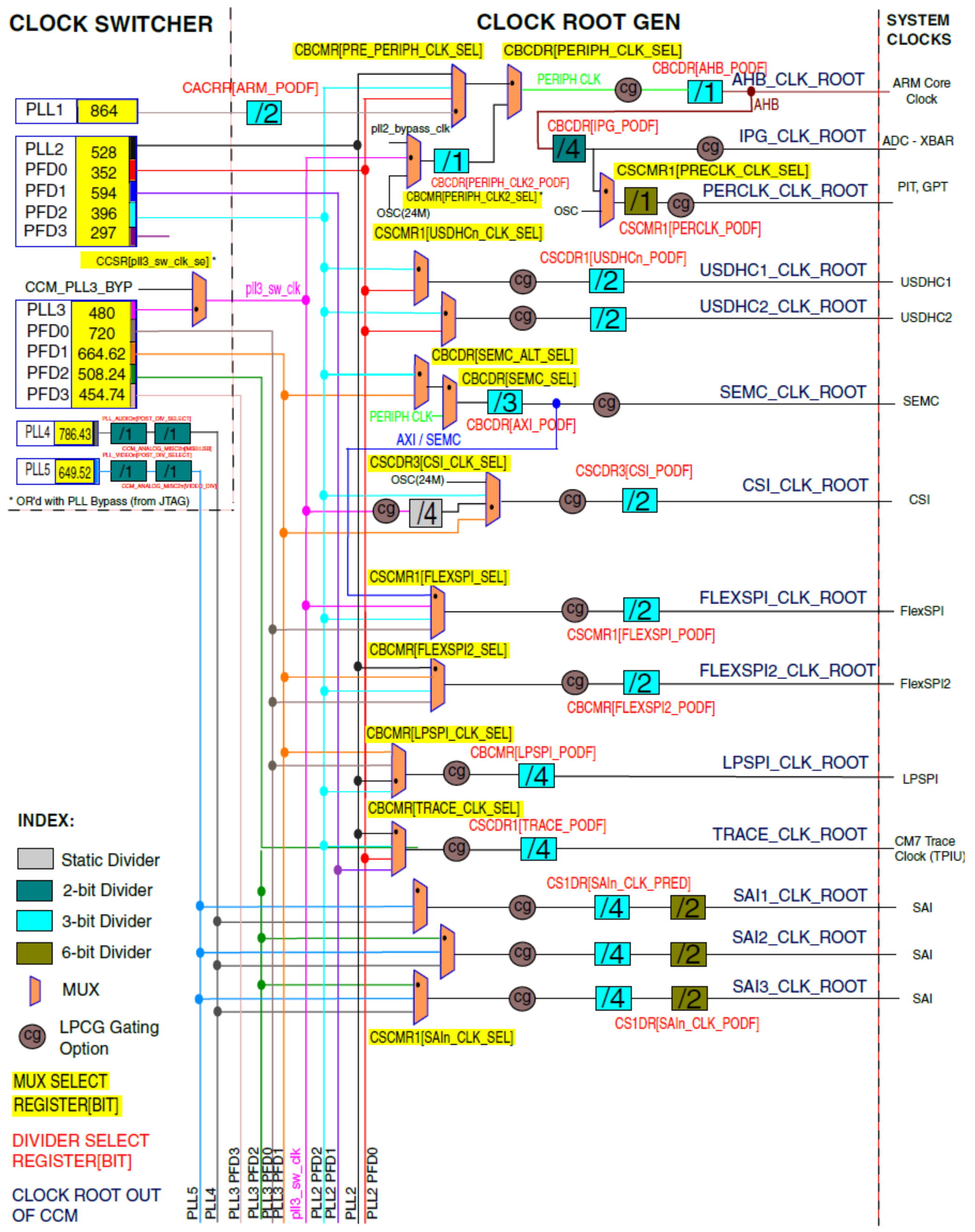


Figure 15: Selected clock tree of i.MX RT chip. If a functional module needs a clock, the clock is derived from one of the system root clock. The root clocks are derived from one of the 7 PLLs(Phase Locked Loop) or their PFDs(Phase Fractional Divider) through demuxes and PODFs(POst Divider Fraction). For our purpose, we just need to know the functional clock can be increased from default by changing the demux or PODF values.

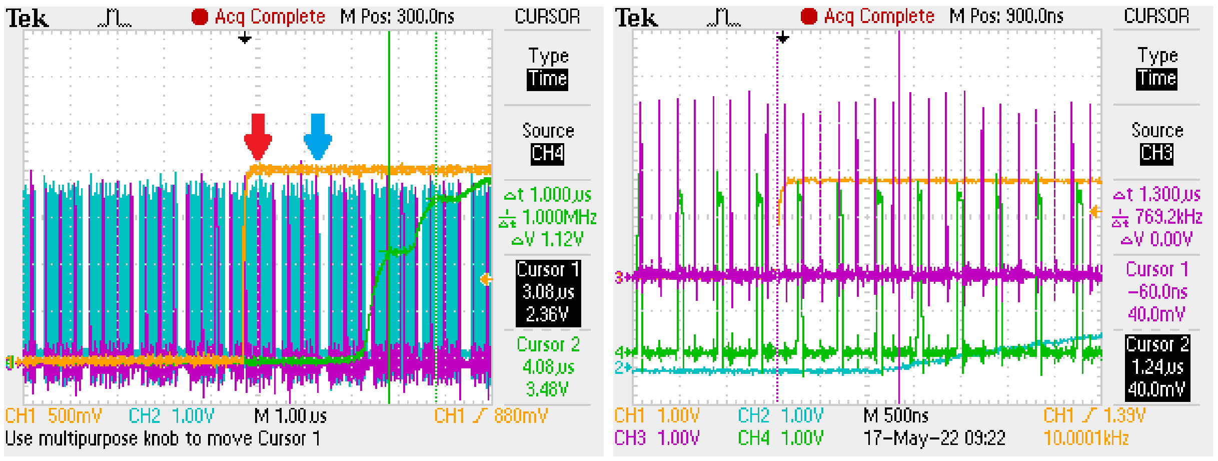

Tek
Acq Complete M Pos: 300.0ns
CURSOR
Type Time
Source CH4
Δt 1.000μs
1/Δt 1.000MHz
ΔV 1.12V
Cursor 1 3.08μs 2.36V
Cursor 2 4.08μs 3.48V
CH1 500mV CH2 1.00V M 1.00μs CH1 ∫ 880mV
Use multipurpose knob to move Cursor 1
Tek
Acq Complete M Pos: 900.0ns
CURSOR
Type Time
Source CH3
Δt 1.300μs
1/Δt 769.2kHz
ΔV 0.00V
Cursor 1 −60.0ns 40.0mV
Cursor 2 1.24μs 40.0mV
CH1 1.00V CH2 1.00V M 500ns CH1 ∫ 1.39V
CH3 1.00V CH4 1.00V 17-May-22 09:22 10.0001kHz

Figure 16: Latency with overclock on(right) and off(left). Turning on overclock halves the latency. Also note the extremely slow slew rate of DAC.

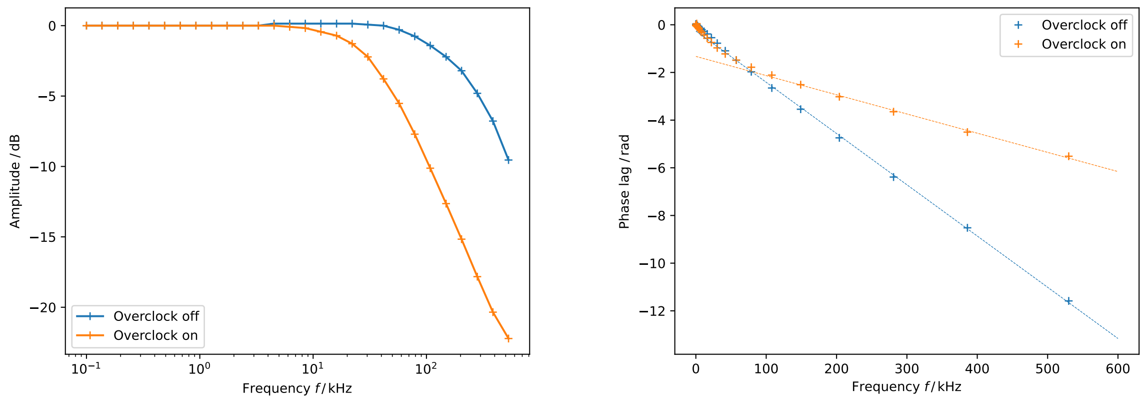


Figure 17: Bode plot. See text for detail.

2. When overclock is turned on, the features of a first-order low-pass filter emerges: the gain shows a typical 20 dB/Decade roll-off, and the intercept of phase is -1.33, close to $\pi/2$.

3. Although the frequency where gain flips sign increases from 140 kHz to 210 kHz, the 3 dB compression point decrease from 191 kHz to 36 kHz.

6 Next step?

The LPSPI module is capable of half-duplex transmission and the MISO pin and also be configured as data output. For DAC transmission, there's minimum need of its readback. Thus we can double the DAC throughput for free. I've currently completed the firmware development(see branch `multi-datapin`) but more radical modification will be made to hardware, as shown in Fig. 18.

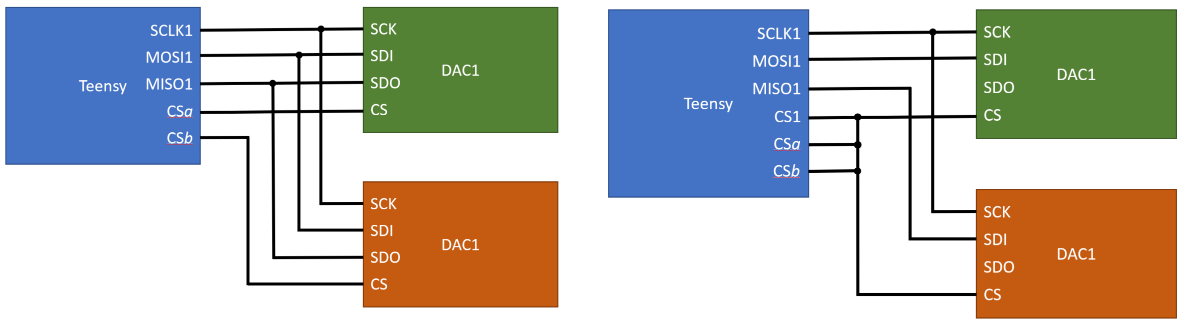


Figure 18: Full-duplex connection vs half-duplex connection. In full-duplex connection, CSa and CSb are two distinct pins controlling independently the update of DACs. In half-duplex connection, we disconnect the SDO of DAC, connect MISO to the SDI of the second DAC, and connect CSa and CSb together to the CS1 pin of `SPI1` instance and update both DAC data register in one transaction. A runtime merge of two DAC registers is required, see `bit_mangler.c`.

References

[1] AD5764 Datasheet. URL: `https://www.analog.com/media/en/technical-documentation/data-sheets/AD5764.pdf`.

[2] AD7386 Datasheet. URL: `https://www.analog.com/media/en/technical-documentation/data-sheets/AD7386-7387-7388.pdf`.

[3] i.MX RT1060 Datasheet. URL: `https://www.pjrc.com/teensy/IMXRT1060CEC_rev0_1.pdf`.

[4] i.MX RT1060 Manual. URL: `https://www.pjrc.com/teensy/IMXRT1060RM_rev3.pdf`.

A Modification history

A.1 May 17, 2022

Removed section "Current issues", replaced with section "Overclocking".