V2CPSE1 Eindopdracht Verslag
Naam: Matthijs Koelewijn
Studentnr: 1716853

Inleiding:
In dit verslag wordt een ST7789 SPI LCD display onderzocht. Hier is een bestaande driver voor, echter is deze onaanvaardbaar langzaam. Met wat opdracht-stappen heb ik onderzocht waar dit door veroorzaakt wordt.

De bestaande driver
De demo runt erg langzaam. In de demo zijn 2 functies die dit zouden kunnen veroorzaken: clear() en flush(). Hiervoor heb ik twee metingen gedaan om te checken welke van de twee functies het traagst is.

Meting 1:
Voor de eerste meting heb ik de functie hwlib::now_us() gebruikt om te bepalen hoe lang de beide functies duren.

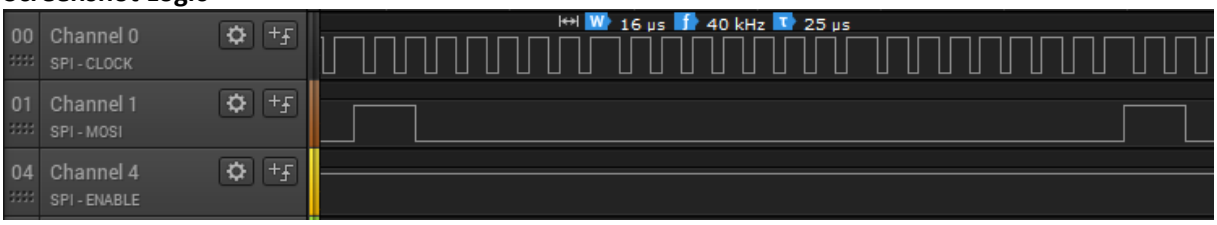
Gebruikte code voor meting:
```
    for(;;){
        hwlib::cout << "1: " << hwlib::now_us() / 1'000  << "\n";
       display.clear( hwlib::red );
         hwlib::cout << "2: " << hwlib::now_us() / 1'000 << "\n";
       display.flush();
         hwlib::cout << "3: " << hwlib::now_us() / 1'000 << "\n";

       display.clear( hwlib::green );
       display.flush();

       display.clear( hwlib::blue );
       display.flush();
    }

}
```

Output:
```
ST7789 demo
1: 2846
2: 2936
3: 38733
```

De output cijfers zijn uitgedrukt in milliseconden, hierin zie je het verschil tussen 1 en 2 is 90ms.
Dat betekent dat de clear() functie slechts 90ms duurt in deze meting.
Daarentegen is het verschil tussen 2 en 3, 35.797ms.
De flush() functie kost ongeveer 36 seconden.
Dus de flush() functie neemt aanzienlijk meer tijd in beslag, want 35.797ms > 90ms.

Meting 2:
Voor de tweede meting heb ik de logic analyzer gebruikt om te kijken wat de SPI klok frequentie is.

Screenshot Logic



Per pixel worden er 3 bytes verstuurd over de SPI bus. Aangezien onze resolutie 240 x 240 is. Betekent het dat er bij iedere flush 57600 pixels aangezet moeten worden.

SPI klok frequentie is 40khz, wat gelijk is aan 40.000 bits per seconde. Er worden 8 * 3 * 57.600 = 1382400 bits verstuurd. Dus dat duurt dan 1.382.400 / 40.000 = 34,56 seconden.

Conclusie: De cijfers van meting 1 en meting 2 komen met een klein verschil overeen. Dit komt omdat er ook weleens uitschieters tussen zitten.

Zonder delay in de SPI driver

Om onze code beter te maken hebben we de spi_bus_bit_banged_sclkl_mosi_miso klasse uit de hwlib SPI library gekopieerd onder een aangepaste naam. spi_bus_matthijs. Hiervoor moest de constructor ook voor gewijzigd worden. En de aanroep van de klasse in de main file.
Daarbij was ik even vergeten om het ook nog in de makefile aan te passen en als include mee te geven in de main.

Hierna heb ik kunnen verifiéren dat je de klasse kan gebruiken en dat de timing die de demo print nog het zelfde is.

Na het verwijderen van de delay calls in de functie write_and_read() is de output aanzienlijk veranderd:

Voor de metingen 3 en 4 heb ik hetzelfde gedaan als meting 1 en 2.

Meting 3:

```
ST7789 demo
1: 2840
2: 2930
3: 2945
```

De getallen komen helemaal niet meer overeen met elkaar.
Hierin duurt de flush nog maar 15 ms in plaats van 36 seconden.

Meting 4:`

Zoals je kunt zien is hier de SPI klok frequentie 250kHz, zoals in de vorige berekening doen we 250 * 1000 = 250.000, 8 * 3 * 57.600 = 1.382.400 bits, dus dan duurt het 1.382.400 / 250.000 = 5,5296 seconden.
Hoewel er nog weleens wat verandering in de SPI klok frequentie kan zitten.

Conclusie: De getallen van meting 3 en meting 4 zijn ten opzichte van meting 1 en meting 2 sterk verbeterd, hoewel de getallen van meting 3 en meting 4 niet echt met elkaar overeen komen.

Werking van hwlib::now_us()

De hwlib::now_us() functie maakt gebruik van een 24 bits timer die op de CPU klok frequentie 84 MHz loopt. De now_us() functie geeft een waarde terug die is samengesteld uit de 24 bits van de timer, aangevuld tot 64 bits met een waarde die de functie zelf bijhoudt.

Iedere keer dat de functie concludeert dat de timer(sinds de vorige call) is overgelopen wordt er 1 opgeteld bij die hogere bits. Als die functie vaak genoeg wordt aangeroepen worden er geen overflows gemist.

Aan de hand van deze informatie kunnen we de discrepantie in de meetwaarde verklaren.
Hieruit kunnen we concluderen dat de 64 bits van de functie zelf niet goed wordt bijgehouden.
De functie moet vaak genoeg worden aangeroepen wil je dat er geen overflows worden gemist. De functie concludeert niet dat de timer is overgelopen, want hij wordt niet vaak genoeg aangeroepen. Dus er worden te veel overflows gemist.

Interval

Uit de Interval zie je dat A1 – A2 = 5,5831.. seconden duurt.
Dit komt wel overeen met mijn handmatige waarneming.

Pas write_and_read() aan

In de functie write_and_read wordt gelezen en/of geschreven en er kunnen met 1 aanroep meerdere bytes worden overgedragen. Om de functie sneller te maken gaan we de dingen verwijderen die voor de LCD aansturing niet nodig zijn.

Aangepaste functie

```
  //Aangepaste functie
void write_and_read(
   const size_t n,
   const uint8_t data_out[],
   uint8_t data_in[]
) override {
      uint_fast8_t d = *data_out++;

      for( uint_fast8_t j = 0; j < 8; ++j ){
         mosi.write( ( d & 0x80 ) != 0 );
         sclk.write( 1 );
         d = d << 1;
         sclk.write( 0 );
      }

}
```

Meting 5

Uit meting 4 zagen we dat de SPI klok frequentie 250 kHz was.
De SPI klok frequentie is nu 333,3 kHz.
De arduino due heeft 84MHz
84 MHz = 84.000.000 hz
333,3 kHz = 333.333 hz
84.000.000 / 333.333 = 252
Nu worden er 252 CPU klok pulsen gebruikt per SPI clock cyclus.

De SPI klok frequentie was 250 kHz.
84.000.000 / 250.000 = 336

Dus er worden nu 336 – 252 = 84 minder CPU klok pulsen gebruikt per SPI clock cyclus.

Analyseer de verschillen in assembler

We hebben nu aan de makefile de regel RESULTS += main.lss toegevoegd. Hierdoor wordt bij het builden ook een assembly listing gegenereerd van het totale project.

Om te checken of de vorige berekening klopt, gaan we alle CPU klok pulsen tellen uit de assembly code.

Voor de aangepaste functies tel ik 66 CPU klok pulsen en voor de oude functie 90. Dat betekent dat er 24 meer CPU klok pulsen gedaan worden in de oude functie.
Dit klopt niet met mijn vorige berekening, maar qua verhouding wel. Want als we (90 / 4)* 3 = 67,5, dit ligt dicht op de 66.

Optimaliseer in C++ voor specifieke pinnen

De write_and_read() functie zelf is nu ongeveer zo optimaal als hij kan zijn, de meeste CPU tijd zit nu in de GPIO pin write() functies die worden aangeroepen. Hier wordt een prijs in CPU tijd betaald voor de flexibiliteit: de pin parameters die door spi_bus_bit_banged_sclk_mosi_miso constructor worden vereist zijn abstracte klassen, wat er aan de constructor call wordt meegegeven zijn concrete afgeleiden van die abstracte klassen. De write() functie van die objecten is virtual, dus het aanroepen ervan gaat via een Virtual Function Table. Dit vereist (een paar) machine instructies. En wat belangrijker is: de optimizer kan de write() functie niet in-linen omdat hij niet kan weten welke write functie wordt aangeroepen.

Om dit op te lossen maken we een versie van de SPI klasse waarin de write acties(voor de MOSI en klok pinnen die we nu gebruiken) direct op de Due hardware registers.

Voorbeeld van vernieuwde klasse.

```
    //Aangepaste functie
    void write_and_read( const size_t n, const uint8_t data_out[], uint8_t data_in[] ) override {
        uint_fast8_t d = *data_out++;

         for( uint_fast8_t j = 0; j < 8; ++j ){
            if((d & 0x80) != 0){
                PIOC->PIO_SODR = 0x01 << pin_number_mosi; // Set pin HIGH
            }
            else{
                PIOC->PIO_CODR = 0x01 << pin_number_mosi; // Set pin LOW
            }

            PIOC->PIO_CODR = 0x01 << pin_number_sclk; // Set pin LOW(because inverted HIGH)
            d = d << 1;
            PIOC->PIO_SODR = 0x01 << pin_number_sclk; // Set pin HIGH(because inverted LOW)
        }
    }
```

De flush tijd is nu 0,31 seconde
De SPI klok frequentie is onstabiel en gaat van 153kHz tot 182kHz.

Minder geheugen gebruiken

De hwlib driver gebruikt een pixel buffer met 1 bytes per pixel. In die byte zitten 2 bits voor rood, 2 bits voor groen, en 2 bits voor blauw, en nog 2 bits zijn ongebruikt. Voor het hele LCD is dat dus 240 x 240 bytes = 57.6 Kb. Een Arduino Due heeft 96 Kb RAM, dus dat past.
Voor chips met veel minder RAM dan een Due is de huidige driver niet geschikt. Voor zulke chips zou het LCD als 1-bit-per-pixel (zwart-wit) gebruikt kunnen worden. Hoeveel RAM is dan nodig voor de buffer?

Als er 1 bit per pixel wordt gebruikt in plaats van 8 bits per pixel.
Dan doen 57,6 Kilobytes / 8 = 7,2 Kilobytes.
Dan is er theoretisch gezien maar 8 Kb RAM nodig minimaal.

Hier gaan we een nieuwe zwart-wit versie van de st7789 driver maken.
Met de C-style array worden waarden opgeslagen in bytes, in dit geval willen we waarden opslaan als bits. Hiervoor is een eigen array klasse geschreven die dat wel doet.

Demo:

Youtube link: https://www.youtube.com/watch?v=dMLZbF4EUws

Resultaat

Wat hebben we nu? We hebben nu uiteindelijk een aangepaste SPI klasse die direct op de pinnen werkt, een noncolor driver van ST7789 die 8 keer minder RAM verbruikt. En uiteindelijk hebben we de totale snelheid van de flush() functie van 36 seconden naar 0,31seconde gebracht.
Ik ben erg blij met het resultaat en vond het een leuk onderzoek.