2020-2021 V2CPSE1 eindopdracht LCD

* Optimalizatie
* assembler
* Template
* Constexpr - fractal

Deze opdracht gaat over een st7789 SPI LCD display. Met de standaard hwlib is het flush()-en van dit display onaanvaardbaar langzaam. Aan de hand van de gegeven opdracht-stappen ga je onderzoeken waar dit door veroorzaakt wordt, en een snellere versie maken. Maak van je onderzoek een verslag. Zorg dat al je metingen herhaalbaar zijn (respository met alle versies van je code, noteer alle relevante omstandigheden). Neem waar relevant de ruwe metingen (bv screenshots van de local analyzer, assembler fragmenten) op in je veslag en/of verwijs naar files in je repository. Bekijk na iedere stap je meetwaarden, ga na of die (ongeveer) in overeenstemming met elkaar zijn, en benoem belnagrijke verschillen.

Benodigheden: ontwikkelomgeving, Arduino Due, logic analyser, kleuren LCD display (240\*240 pixels, SPI aansluiting, ST7789), beadboard, draadjes.

Clone of pull de 2cpse1-examples en hwlib repositories. (Als je voor IPASS in de hwlib SPI file de delay had weggehaald, dan staat die er nu weer. Dat is de bedoeling voor deze opdracht.) In de examples is een st7789 demo opgenomen. Bouw het circuit en run de demo met het display.

De demo runt erg langzaam. In de demo zijn 2 functies die dit zouden kunnen veroorzaken: clear() en flush(). Gebruik de functie hwlb::now\_us() om te bepalen hoe lang de beide functies duren. Bepaal ook zelf (door te tellen, of met een stopwatch) hoe lang de functies duren. Gebruik de logic analyser om te kijken wat de SPI clock frequentie is. Bereken ahv. die frequentie hoe lang een flush() zou moeten duren. Per pixel worden er 3 bytes verstuurd over de SPI bus. Kloppen je getallen (een beetje) met elkaar?

65 ms, 35 sec, klopt met de zelf, 41 kHz, 33 sec, SPi ahv. Delay: 500 kHz, dat klopt niet

Maak in je project een eigen copie van de spi\_bus\_bit\_banged\_sclk\_mosi\_miso classe uit de hwlib SPI library, onder een aangepaste naam. (Als je de naam van een klasse aanpast moet je ook de constructors hernoemen.) Verifieer dat je die eigen klasse kan gebruiken en dat de timing die je demo print nog het zelfde is. De functie write\_and\_read() is verantwoordelijk voor het verzenden van 1 byte over de SPI bus. Haal uit deze functie de delay calls weg en voer de metingen van de vorige stap nog eens uit. Kloppen je getallen weer (een beetje) met elkaar?

65 ms, 24 ms, klopt niet: 5-10 sec, 225 kHz, 6 sec, 24 ms klopt beslist niet

De hwlib\_now\_us() functie maakt gebruik van een 24 bits timer die op de CPU clock 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. Verklaar aan de hand van deze informatie de discrepantie in je meetwaarden.

20 ms, 6 sec >> 20 ms, dus er worden heel veel overflows gemist

Meet mbv. de logic analyser hoe lang een flush() duurt. Hiertoe moet je over een langere tijd data verzamelen. Zet de T1 en T2 markers aan het begin en einde van een blok klok pulsen, dan kan je de verstreken tijd aflezen. Klopt dit met je waarneming?

6.4 sec, klopt

De write\_and\_read() functie is algemener dan nodig is voor het aansturing van het LCD: er wordt gelezen en/of geschreven, en er kunnen met 1 aanroep meerder bytes worden overgedragen. Pas de functie body aan zodat er geen rekening wordt gehouden met meerdere bytes, er niet wordt ontvangen, en er geen rekening mee wordt gehouden dat de dat\_out een nullptr zou kunnen zijn. Wat is nu de SPI klok frequentie? Hoeveel CPU clock pulsen worden er nu minder gebruikt per SPI clock cyclus?

Is nu 258 kHz. Verschil in periode is 1/225e3 – 1/258e3 = 4.44us – 3.88us = 560ns => 46 klok pulsen.

Voeg in de makefile een regel **RESULTS += main.lss** toe. Hoerdoor wordt bij het builden ook een assembly listen gegenereed van het totale project. De syntax en instructies hierin komen niet altijd overeen met wat je geleerd hebt (oa. omdat de Due een Cortex-M3 heeft), maar genoeg om conclusies te trekken. Vergelijk de assembly van beide versies voor de write\_and\_read() functie. Tel de clock pulsen **in de loop** die nodig waren voor de instructies de je hebt weggehaald. Klopt dit met je vorige berekening?

Ongeveer 10 instructies weggehaald, dus op het eerste gezicht nee. Maar 2 ervan waren blx instructies, die roepen een subroutine aan => kost veel meer tijd.

Hoeveel instructies bevat de loop in de laatste versie (zonder de overbodige franje voor lezen en meerdere bytes)? Vergelijk dit met het aantal CPU klok pulsen per SPI clock periode. Het verschil zit ‘m in de bl instructies, die roepen een subroutine aan.

12 instructies (5 zijn 32 bits, 1 jump, dus gok: 12 + 5 + 2 = 19 cycles ); 3.88 us \* 82 MHz = 318 instructies!

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 era an 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) machinje instructies,. En wat belangrijker is: de optimizer kan de write(0 functie niet in-linen omdat hij niet kan weten welke write functie wordt aangeroepen.

Zoek in de Arduino Due library file op hoe het schrijven naar een pin precies gaat (dit werd deels al uitgelegd in de V1OOPC reader). Maak een versie van de SPI klasse waarin je de write acties (voor de MOSI en clock pinnen die je nu gebruikt) direct doet op de Due hardware registers. Hoe welke clock frequentie en flush() tijd kan je Hiermee bereiken?

=

Deze oplossing is best snel, maar