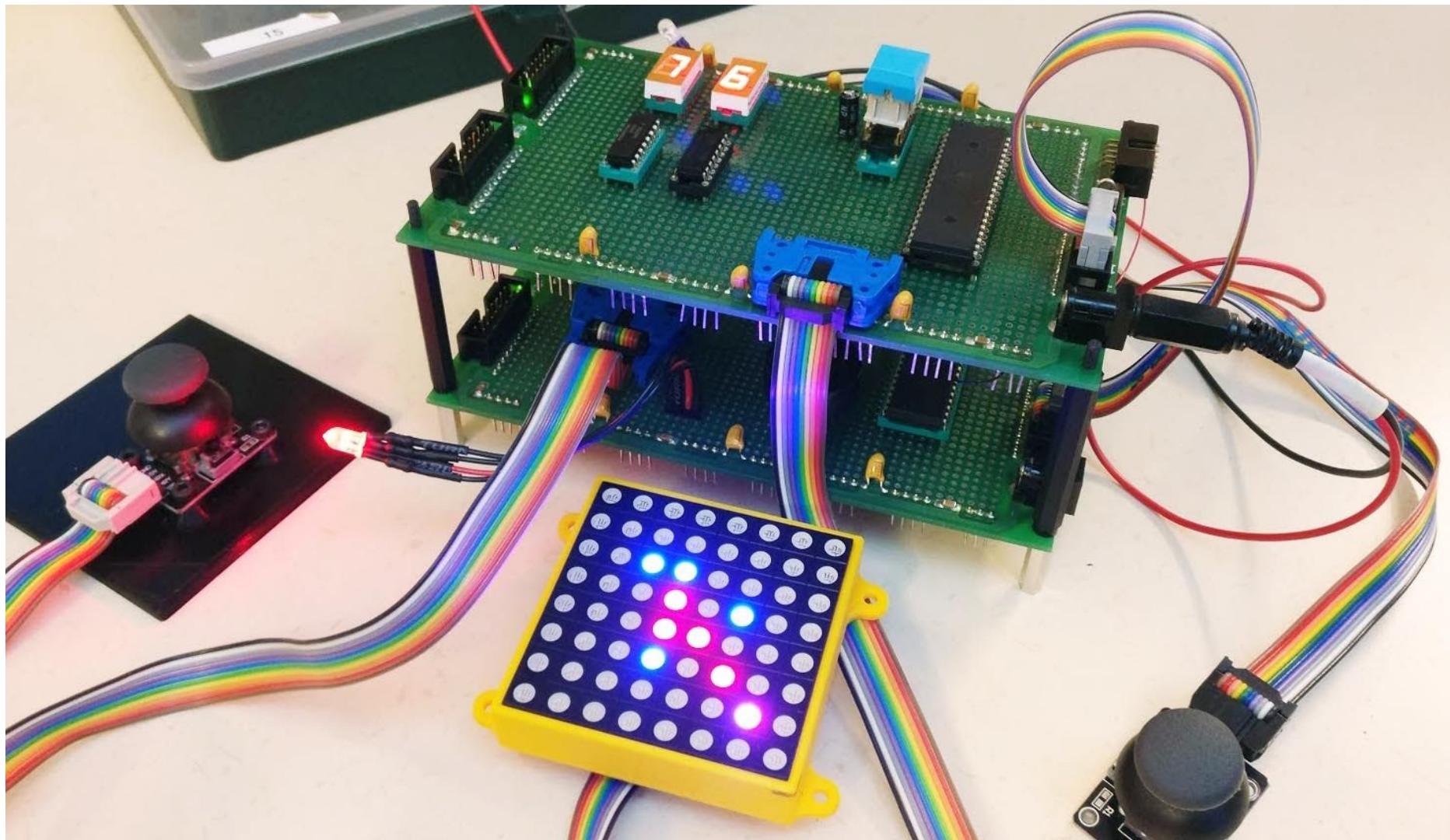


RETUR

Mikrodatorprojekt: fyra i rad

Rapport



OVANLIGT OCH PÅENKEDLT KLUMPIG LÖSNING

Med HW & SW I SAMMA RUBRIK MEN DET FLYTER OK

Felix Bergström
Axel Brinkeby
Linus Classon
Linus Nygård

OVANLIGT SNYGG KOD
KAN NOG OPTIMERAS

Linköping universitet

Linköping

2020-04-03

Allt man kan önska av en fram sida. Bild & ALLT

!

Version 1.0

Kontaktperson: linc1896@student.liu.se

Sammanfattning

Denna rapport beskriver uppbyggnaden, skapandet och felsökningen av ett fyra i rad spel skapat med hjälp av två ~~stycken~~ mikrodatorer och diverse komponenter. Målet med projektet var både att konstruera hårdvaran och välja komponenter samt att skriva programkoden i AVR Assembler som krävs för att två personer ska kunna spela flera runder av fyra i rad. En viktig del i projektet var att ha två mikrodatorer som kommunicerade med varandra. Projektet utfördes av en grupp på fyra personer.

Lång

SKRIVER NI DEDA FÖR ATT
POÄNGTERA YVETE DEE ?
2) 2 MIKRODATORER

Innehållsförteckning

Figur- och tabellförteckning	3
1. Inledning	4
1.1 Sammanfattning av spelets funktion	4
1.2 Syfte med projektet	4
1.3 Kravspecifikation	4
2 Projektet	6
2.1 Blockschema	6
2.2 Kopplingsschema	7
2.3 Arbetsfördelning	8
2.4 Projektets delar	8
2.4.1 Display	8
2.4.1.1 Skrivning till displayen	9
2.4.2 Analoga styrspakar	10
2.4.3 Högtalare	10
2.4.4 Seriell datakommunikation mellan processorerna	10
2.4.4.1 Dataprotokoll	11
2.4.5 7-segmentsdisplay	11
2.5 Projektets kod	13
2.5.1 Programmering av spel-processorn	13
2.5.1.1 Spelplanen	13
2.5.1.2 Flytta markör	14
2.5.1.3 Markera en plats	14
2.5.1.4 Koll om spelet är slut	14
2.5.2 Programmering av video-processorn	15
2.5.2.1 Struktur	15
2.5.2.2 Videominnet	15
2.5.2.3 Skrivning till Videominnet	16
2.5.2.4 Instruktionscheck och utförande	17
Lätt att hitta det man söker – och det är ju meningen.	
3 Resultat	17
4 Avslutande tankar	18
4.1 Misstag	18
4.2 Möjliga förbättringar och avslutande diskussion	18
5 Referenser	20
6 Bilagor	21
Bilaga 1: Programkod för spel-processorn	21
Bilaga 2: Spellogik (spel-processorn)	27
Bilaga 3: Joystick driver (spel-processorn)	35
Bilaga 4: Programkod för video-processorn	39
Bilaga 4: UART implementation (video-processorn)	49

Figur- och tabellförteckning

Figur 1.1. Kravspec blockschema	5
Figur 2.1. Blockschema	6
Figur 2.2. Kretsschema	7
Figur 2.3. Lysdiodsmatris schema	8
Figur 2.4. Lysdiodsmatris representation	9
Figur 2.4.2. Kopplingsschema styrspak	10
Tabell 1. Beskrivning dataprotokoll	11
Figur 2.5. Schema 7-segment	11
Figur 2.6. Minnesstruktur spelprocessor	13
Figur 2.7. Videoprocessor main-loop	16
Figur 2.8. Videoprocessor SRAM	16
Figur 2.9. Videoprocessor lookup-tabell	17
Figur 4.1. Fyra i rad	19

(Har nog aldrig riktigt förstått
poängen med figurlista)

1. Inledning

Den här rapporten beskriver hur ett digitalt fyra i rad spel för två personer konstruerades som en del i kursen TSIU51.

FORST

KORTAST (ÅR)

SEON

OK

1.1 Sammanfattning av spelets funktion

Två spelare kan lägga ner "spelpjäser" med målet att få fyra i rad. Spelarna turas om att lägga på en 8x8 spelplan. De väljer plats och konfirmerar valet med hjälp av varsin styrspak. När ett val gjorts hörs ett pip från högtalaren. När en spelare sedan lyckats bygga en oavbruten linje av fyra spelpjäser, antingen vågrätt, lodrätt eller diagonalt, så rensas spelplanen och vinnarens poäng ökas med ett på vinnarens poäng-display.

BILD HÄR

DET FINNS EN HÖGTALARE

1.2 Syfte med projektet

Syftet med projektet var framförallt att vi skulle få en chans att lära oss mer om hur mikrodatorer kan kommunicera med varandra och annan yttre hårdvara. Men också att skapa ett fungerande elektroniskt spel för två personer. Projektet gav oss också en chans att lära oss mer om assemblerprogrammering av mikrodatorer.

1.3 Kravspecifikation

OK

Forklaring av ORDNING BDA
Vad kravspec

Innan arbetet påbörjades skapades en kravspecifikation över projektet. Dess mål var att rita upp en bild av vad slutprodukten önskas ha för funktionalitet och var på så sätt ett "kontrakt" mellan gruppmedlemmarna och handledaren gällande vad som skulle uppnås. Kravspecifikation bestod av en kort sammanfattning av produkten, ett grundläggande blockschema över komponenterna (se figur 1.1), en lista över grundkrav som måste uppfyllas och en lista utökade krav att implementera om tiden fanns.

SKALE-KRAV

Synd att den inte syns

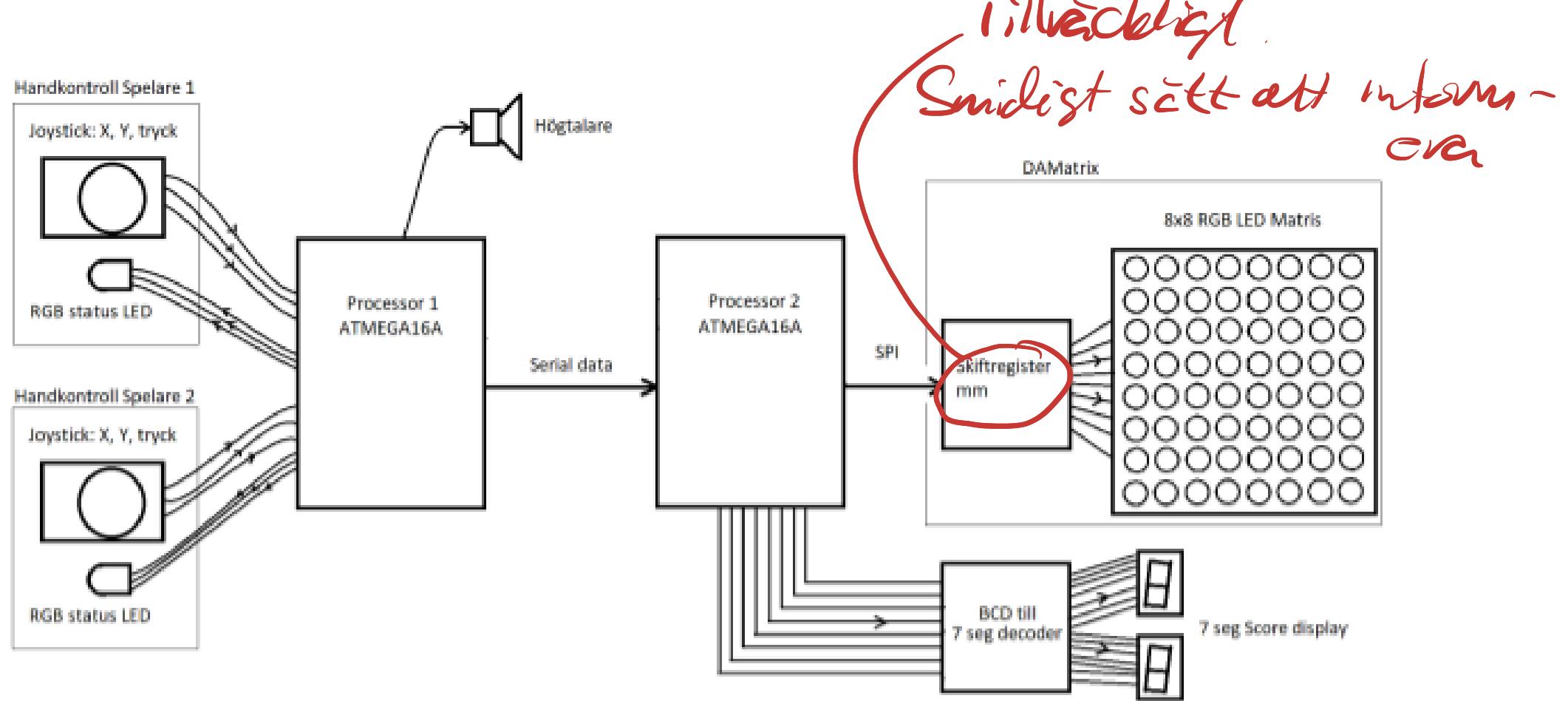
Den önskade grundfunktionaliteten var som följande:

- 1 • Spellogiken ska kunna känna av när spelet är över, dvs när en person fått fyra i rad eller när alla platser är fyllda på spelplanen
- 2 • Kunna spelas av två spelare med varsin styrspak och färg. *← Inte komplett mening*
- 3 • Det ska finnas två stycken poängtavlor bestående av en 7-segment display vardera, alltså en tavla per spelare.

Dessa var enligt överenskommelse absolut grundläggande. Det fanns dessutom tre utökade krav som vi skulle sikta på efter att de grundkraven var uppfyllda:

- Lysdioder som visade vilken spelares tur det var
- En högtalare som signalerade då ett val har gjorts av någon spelare.
- 3D-printade skal åt styrspakarna.

Skal



Figur 1.1. En tidig visuell representation av hur projektet skulle se ut. För det mesta oförändrad jämförd med faktisk slutprodukt. **Beskriv**

Vad vi ser

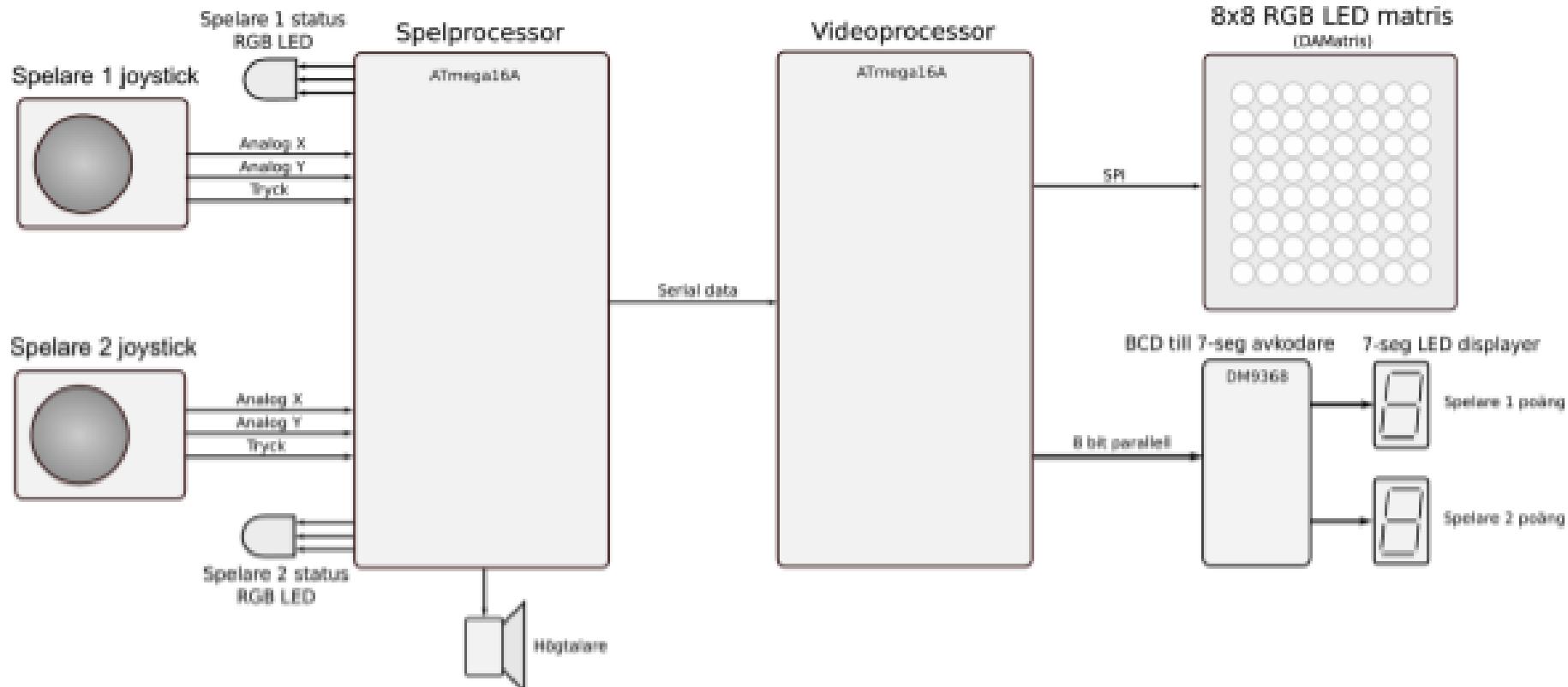
Gott om plats
f att förklara
blockschema o

2 Projektet

Text

2.1 Blockschema

Spelet är uppbyggt kring två stycken ATmega16A mikroprocessorer[1]. spel-processorn har hand om spellogiken. Den läser in data från det två styrspakarna och har en representation av spelplanen i minnet. Denna processor kontrollerar hur spelarna placerar sina spelpjäser samt gör en kontroll om någon vinner för varje spelpjäs som placeras. spel-processorn har också en liten högtalare samt två RGB lysdioder för återkoppling till spelaren.
video-processorns uppgift är att visa vad som händer på displayen. video-processorn visar också aktuella poäng för spelarna på 7-segmentsdisplayer. Se figur 2.1.

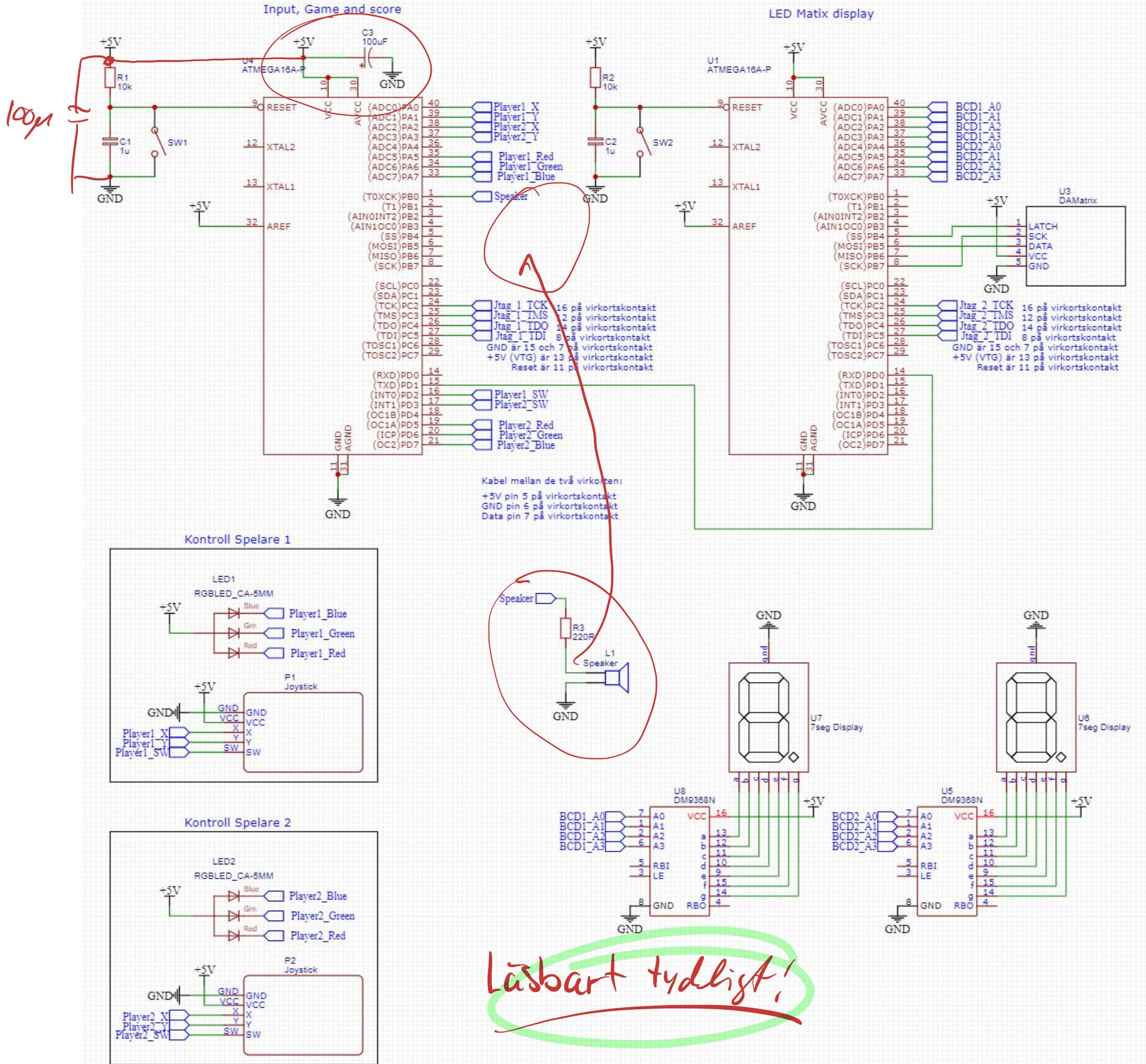


Figur 2.1. Blockschema över implementation av komponenter. Definierade vad de olika mikrokontrollernas uppgifter skulle vara.

Fig 1.1 har samma info(nästan)
Högtalaren har halkat ner sen sät

2.2 Kopplingsschema

Ett kopplingsschema över hårdvaran utformades, se figur 2.2. Detta klargjorde exakt vilka komponenter som skulle användas och hur de skulle kopplas samman. På kopplingsschemat syns också lite anteckningar om hur programmerings-interfacen (JTAG) skulle anslutas till vir-kortens kontakter. Kopplingsschemat ritades i online-verktyget EasyEDA.



Figur 2.2. Fullständigt kopplingsschema över projektet. *Mer förklarande text*

Vill man veta hur många dom var?

2.3 Arbetsfördelning

Gruppmedlemmarna delades in i två mindre grupper. Felix Bergström och Linus Nygård fokuserade på video-processorn och att få displayen att fungera medan Axel Brinkeby och Linus Classon jobbade med spel-processorn som skulle läsa in analoga värden från styrspakarna och köra själva spellogiken. För att effektivisera arbetet byggdes elektroniken upp på två ~~st~~ separata virkort, ett för spel-processorn och ett för video-processorn. Det var då en naturlig uppdelning att sätta två personer på varje processor. De två separata korten gjorde det möjligt för grupperna att samtidigt bygga upp elektroniken, och även programmera varsitt kort vid olika datorer samtidigt.

2.4 Projektets delar

TEXT

2.4.1 Display

En Rektangel Med Text ...
Lättlätt upp

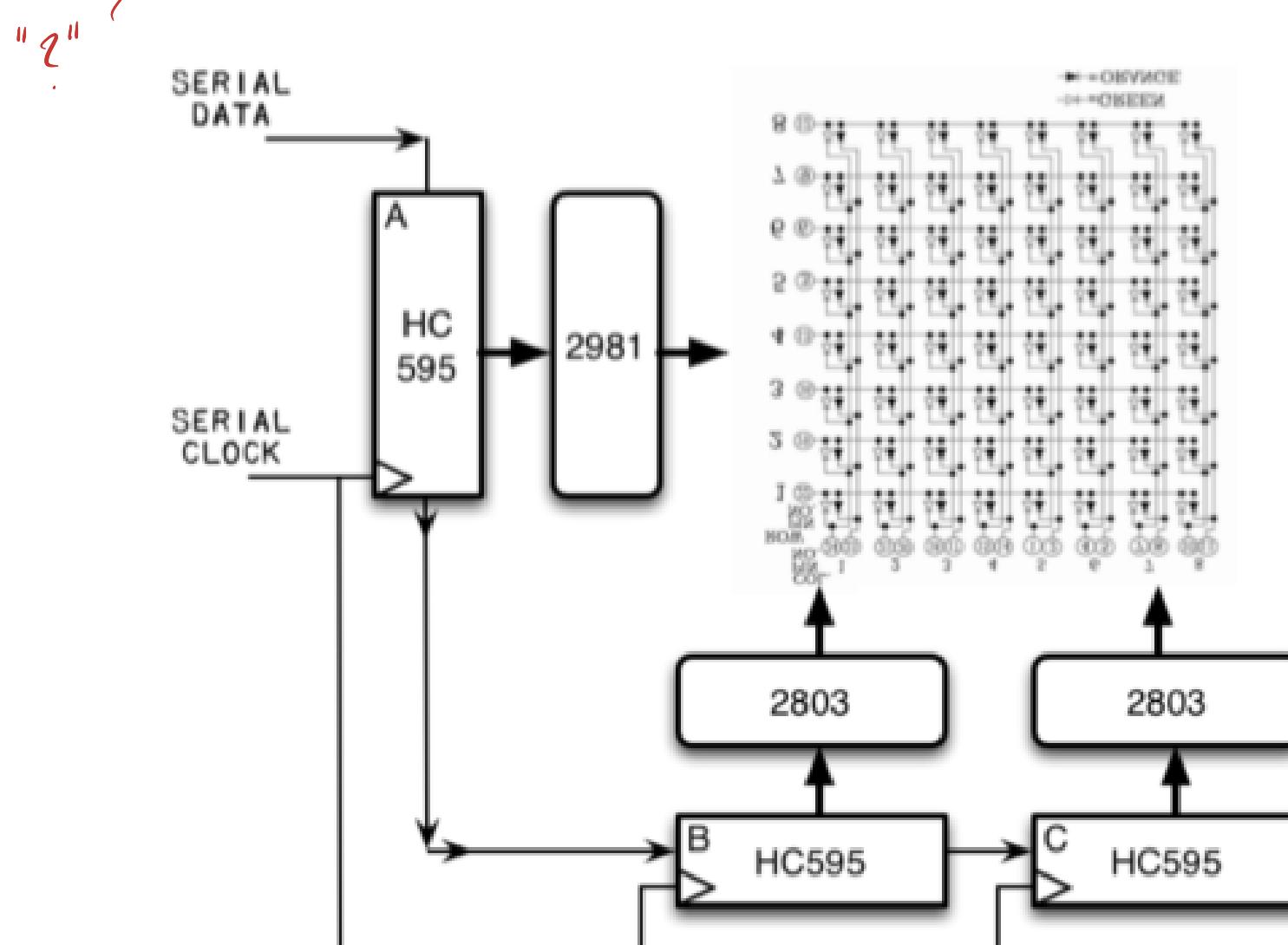
Displayen som användes var en 8x8 *DAMatrix RGB lysdiodmatris*[2]. Den agerade som spelets spelplan. Den ansågs vara det bästa valet för denna applikation av flera anledningar.

En första
anledning ...

Först och främst har displayen tre lysdioder per position, röd, grön och blå, därav RGB.

Detta ger spelet effektivt sex olika färger att utnyttja. På så sätt fanns möjligheten att bland annat tilldela spelarna varsin färg. En annan anledning är "DAMatrix-delen". LED-displayen i sig själv kräver väldigt många portar för att välja vilken diod som ska tändas, för många för att enkelt styra enbart med video-processorns utgångar. Därför har DAMatrixen färdigt ett flertal skiftregister. Detta gjorde det möjligt att skifta in datan som krävdes seriellt. På så sätt kunde displayen styras med, i grund och botten, endast tre portar. En dataport för att skifta in ettorna och nollorna seriellt till respektive register och en klockport för att styra skiftandet. Det finns en extra uppsättning av register som syns i figur 2.3 (2981 och 2x2803). Dessa är till för att kunna "skiffta fram data" så att den faktiskt syns på displayen. Detta möjliggör att ny data kan börja skiftas in utan att displayen behöver släckas först. Dessa register styrs inte av samma gemensamma klocka (SCLK) utan kräver en tredje port, LATCH, för att skiftas.

Signaler



Figur 2.3. Schema över seriell uppkoppling till matris. Notera att schemat är för en tvåfärgad matris. I vårat fall finns en till uppsättning av HC595 och 2803 register.

2.4.1.1 Skrivning till displayen

*Har unte nämnts tidigare
for 4 byte?*

Displayen behöver alltså fyra byte data och en LATCH-puls för att rita ut något. SPI (*Serial Peripheral Interface*) valdes därför för drivning av matrisen. SPI är ett protokoll för seriell och synkron datakommunikation som stöds av Atmega16A. Seriell kommunikation innehåller att *slå ihop* datan skickas seriellt i databussen, alltså direkt efter varandra. Det vill säga genom en ledare istället för flera. Detta innebär att endast en dataport krävdes för kommunikation. Synkron kommunikation innehåller att slavenheten och masterenheten använder sig av en gemensam klockhastighet. Denna bestäms av masterenheten. *”*

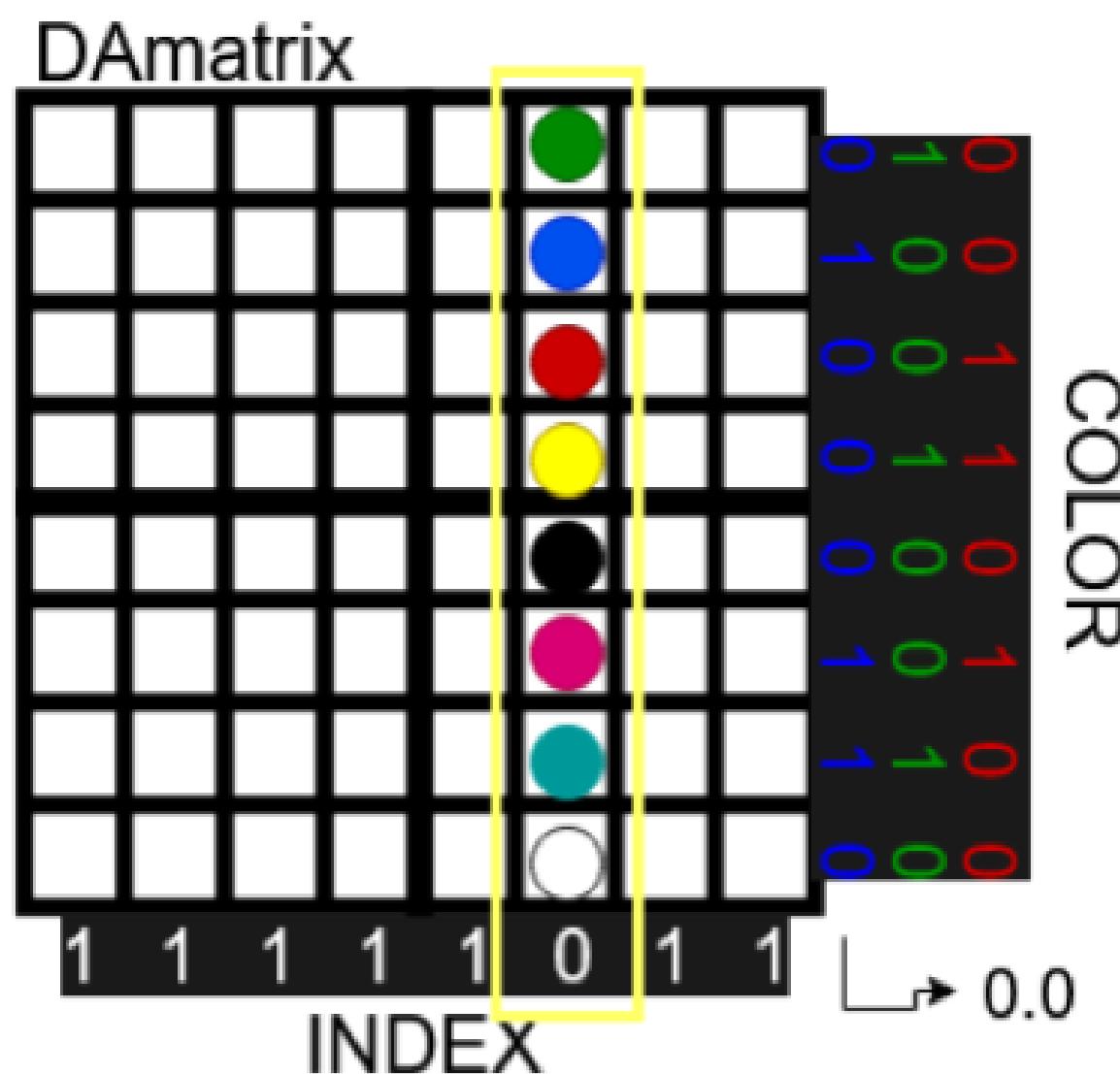
? till en mening

Videoprocessor kunde då användas som *master* och låtas skifta in bytes till en *slave*, displayen i det här fallet. Processorn behövde inte hantera handskakningar eller ”krockar” då slavenhetens jobb endast var att ta emot datan och inte att skicka något. Kommunikationen var enkelriktad. I praktiken skiftades tre bytes åt gången för respektive färg: röd, grön och blå (se figur 2.4). Där en etta i till exempel den röda byten ledde till att motsvarande rads röda dioder kunde tändas. Observera *kunde* tändas. Det krävdes också en nolla på någon position i *index*-byten för att tända en kolumn.

Vad är det?

Det blev då fördelaktigt att multiplexa displayen varje gång de fyra byten skickas. Detta uppnås genom att skifta en nolla i INDEX mellan varje kallelse av SEND rutinen som går igenom mer detaljerat senare i rapporten.

Program Courier



Men detta är en beskrivning av alla DATOR dvs HW

Svårt att blanda in kod här också

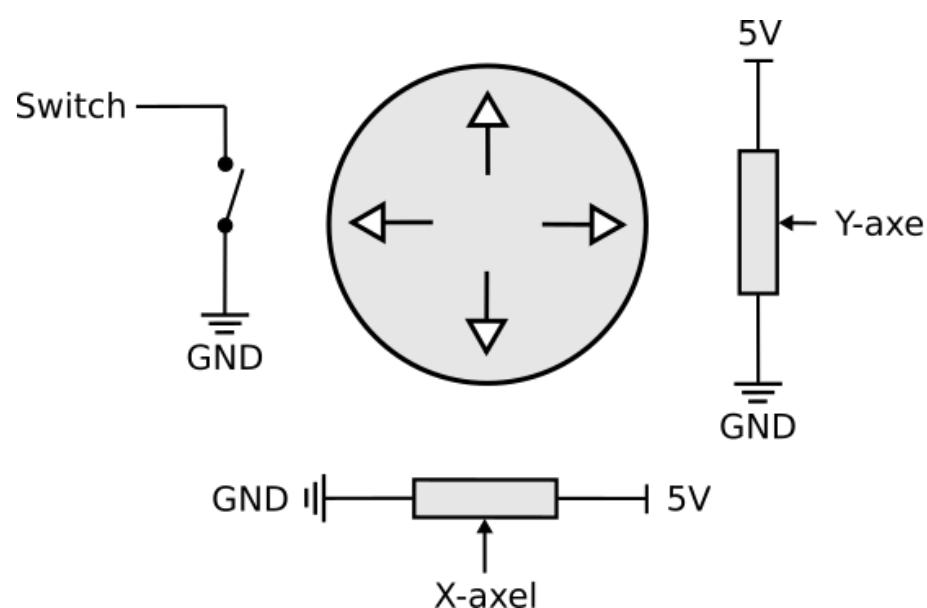
Figur 2.4. Visualisering av hur datan driver displayen.

Berätta Vad vi ser

LAGOM TEXT!

2.4.2 Analoga styrspakar

De analoga styrspakarna som används består av två potentiometrar, en för X-riktningen och en för Y-riktningen, samt en tryckknapp som aktiveras när spaken trycks ner. När en spänning på 5 V läggs över potentiometrarna kan de användas som en ställbar spänningsdelare bestående av två resistorer. En varierande spänning proportionell mot spakens position mellan 0 V och 5 V kan då avläsas från mitten-utgången. Denna spänningen är beroende av spakens position. De analoga spänningarna från spakarna läses av med hjälp av processorns A/D omvandlare (Analog till Digital omvandlare).



Figur 2.4.2: Kopplingsschema för en analog styrspak

A/D omvandlingen görs med enbart 2 bitars upplösning. Detta medför att varje axel delas in i 4 intervall. Den inkommende data får ett värde som är 3 (11 binärt) från A/D omvandlingen när spaken är i det övre intervallet, och 0 (00 binärt) om spaken är i det nedre intervallet. I dessa lägena flyttas spelarens markör åt respektive håll. Om värdet blir 1 eller 2 (01 eller 10 binärt) så är spaken i något av de två mitten-intervallen och då gör inte programmet något.

2.4.3 Högtalare

BILD

Summer
på svenska

Högtalaren används som återkoppling för att bekräfta att spelaren har placerat sin markör. Det var inte nödvändigt att högtalaren skulle kunna göra ljud med olika frekvenser. Enbart ett kort pip i en frekvens räckte. Av denna anledning användes en färdig "buzzer" med inbyggd oscillatorkrets för att göra programmeringen enkel. Denna piper hela tiden med en konstant ton så länge den matas med 5 V. Buzzern kopplades direkt till en utgång på processorn. En kort subrutin i programmet sätter utgången hög en tid för att göra ett kort pip, sedan sätts utgången låg igen.

BRA FÖRDE, FUNKA MED SAMMANBLANDNINGEN

2.4.4 Seriell datakommunikation mellan processorerna

Kommunikationen mellan spel-processorn och video-processorn är enkelriktad. Data skickas från spel-processorn till video-processorn. Detta sker via asynkron seriell dataöverföring. Processorernas USART-hårdvara (Universal Synchronous/Asynchronous Receiver/Transmitter) används för att skicka och ta emot data. spel-processorn skickar data när spelare placerar sina markörer och när spelaren flyttar sin markör. video-processorn läser in denna data med hjälp av avbrottshantering (Interrupts).

2.4.4.1 Dataprotokoll *Kompakt men isekvans o läsbart* *Bra!*

Data skickas med 9600bps baudrate, åtta databitar och en stopbit. Dataprotokollet som används är gjort för att vara så enkelt som möjligt. Ett meddelande består av fyra bytes. Den första byten består alltid av ettor (~~0xFF~~ hexadecimalt). Detta markerar starten på meddelandet och gör det möjligt för video-processorn att trigga på detta och nollställa en räknare för att sedan räkna varje nästkommande byte och på så sätt veta vad det representerar. Detta fungerar eftersom vi förutsätter att ~~0xFF~~ aldrig skickas som data, det förekommer aldrig någon annanstans än som start-byte. Den andra byten är en instruktion om vad det är för data som skickas. Instruktionen kan vara ett värde mellan 0 och 4. De två sista databyten är själva datat som skickas. Tabellen nedan visar vad som kan skickas.

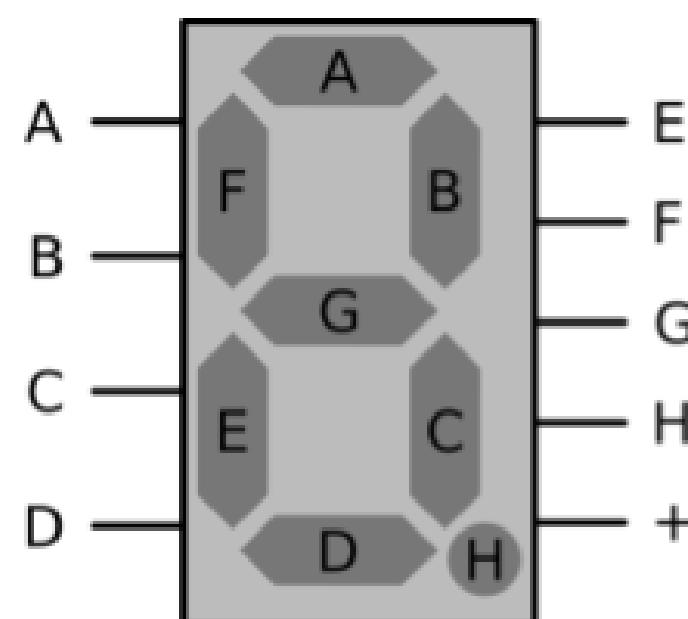
Instruktions nr.	Beskrivning	Data 1	Data 2
0	Spelare 1 markerar här	x-position	y-position
1	Spelare 2 markerar här	x-position	y-position
2	Spelare 1 placerar här	x-position	y-position
3	Spelare 2 placerar här	x-position	y-position
4	Skicka poäng och rensa skärmen	Spelare 1 poäng	Spelare 2 poäng

Tabell 1. Tabell över vad dataprotokollet skickar.

2.4.5 7-segmentsdisplay

För att visuellt demonstrera hur många poäng varje spelare har, dvs. hur många gånger de lagt fyra i rad, användes två 7-segmentsdisplayer med varsin BCD-omvandlare. Detta lät spelet visa upp till 15 (F uttryckt i 7-segment) poäng per spelare. Det är dock mer användarvänligt att låta spelets maximala poänggräns vara nio då de följande sex poängen skulle skildras i bokstäver (A-F). Detta eftersom en 7-segmentsdisplay inte kan uttrycka tiotal decimalt. 7-segmentets utseende och segmenten med dess tilldelade namn syns i figur 2.5.

Vad blev det?



Figur 2.5. En 7-segments tavla med dess motsvarande standardnamn för respektive segment.

För enkelhetens skull styrdes poängtavlorna av varsin BCD-omvandlare (Fairchild Semiconductor (1998), *DM9368 7-segment decoder[3]*). Detta gjorde de möjligt att endast behöva fyra pinnar per display på video-processorn. Totalt endast en port, PORTA, på Atmega16A behövdes då gentemot $7 \cdot 2 = 14$ st. pinnar (exklusive punkten (H), då den ej var nödvändig). Det krävdes också på så sätt mindre kod och processortid för att visa poängen, då det inte krävdes någon översättning från binärt till BCD. Omvandlarna krävde endast fyra bitar data var bestående av att binärt tal. video-processorn kunde då bara skicka ut poängen överförd från spel-processorn på den höga respektive låga delen av PORTA.

Vet läsaren vad ni

pratar om? PORTA?

Har den blivit ordentligt
presenterad än?

Gott om plats

Språk alldeles utmärkt men bättre med bilder.

2.5 Projektets kod

För Tasanen

Både spel-processorn och video-processorn programmerades i AVR Assembler.
Utvecklingsmiljön Atmel Studio användes.

2.5.1 Programmering ~~ut~~ ^{text} spel-processorn

2.5.1.1 Spelplanen

Spelplanen är representerad i spel-processorns sram. En position på planen representeras som en byte i SRAM. Detta innebär att spelplanen tar upp 64 bytes totalt i SRAM. Varje byte visar statusen på sin respektive position på spelplanen. Om en position är omärkande ~~utav~~ en spelare är det en nolla och när en spelare väljer att markera en position placeras en siffra motsvarande den spelaren på den platsen i sram:et. Bilden nedan visar upplägget på hur sram:et är uppdelat.



Figur 2.6 Bilden visar upp minnesstrukturen i Sram:et som används för lagra all information spelet behöver för att kunna köras.

OGENDOMLÖSNINGENS TEXTMASSA

FORTSÄTTA MED JSP

2.5.1.2 Flytta markör

Spelarnas markör flyttas med hjälp ~~av~~ av analoga joysticks som efter den analoga till digitala omvandlingen (a/d omvandling) i processorn tolkas beroende på vilken data man fick in. Därefter ökar eller minskar den spelarens x och y position med 1 motsvarande hur spelaren rör joysticken.

KOORDINAT X,Y?

Spelarnas markörposition sparas i sram:et i separata minnesplatser, så spelare 1 har en plats i minnet för X-koordinaten och en annan plats i minnet för Y-koordinaten som sparar positionen för spelare 1:s markör. *Helt engt i förra bilden*

Här är bra tillfälle att berätta om detta. Detta är tidigare.

De tolkningar som programmet gör från A/D omvandlingen är baserat på vilket värde som kommer in. Detta värde ligger mellan 0 till 3 ~~då bara 2 bitars information används~~. Om värdet är ~~en~~ 3 innehåller det att spelaren tar ett steg "fram" eller till "höger". Om värdet är 0 innehåller det att spelaren vill gå "bakåt" eller till "vänster". Värdena 1 och 2 innehåller att spaken i mittläget och att spelarens markör inte ska flyttas. *Bild*

2.5.1.3 Markera en plats

Tidigare markeras med joysticks. *Detta med position?* När en spelare vill markera positionen som den är på klickar de med styrspaken. Spakarnas tryckströmställare är kopplade så de triggar ett varsitt externt avbrott (Interrupt) då de trycks ned. Först kallas om det är den aktuella spelarens tur, är det inte det så händer inget. Annars går koden vidare till att kolla om positionen spelaren markerat är ledig i spelplanen i SRAM, är den inte ledig så händer inget. Är dock positionen ledig placeras spelarens markör där. Därefter körs en subrutin som skickar iväg instruktioner till video-processorn om att en spelare har placerat en markör och var markören ligger. Om en markör placerades körs sedan en subrutin som gör en koll om spelet är slut. *JSP är fullt rimligt här*

2.5.1.4 Koll om spelet är slut

Det är här den riktiga spel-logiken utförs. Utan denna subrutin kan ingen vinna vilket innehåller att det inte skulle vara ett spel. I denna subrutin sköts allt om för att se till att en spelare kan vinna och att det kan bli oavgjort.

Först kallas om spelplanen är full, är den det så är spelet över och det blir oavgjort och körs sedan en subrutin för att rensa spelplanen och skicka antalet poäng vardera spelare har. Annars börjar koden med att först kolla den horisontella axeln från den punkten där spelaren senast placerade ut sin markör. Om det visar sig att det ligger 3 utav spelarens markörer utöver den som precis blivit placerad har spelaren vunnit. Programmet hoppar då direkt till koden som sköter om vad som sker efter att en spelare vunnit. Denna kontrollen sker för varje riktning, först horisontellt, sedan vertikalt och sist de två diagonala riktningarna.

Efter att alla riktningar kontrollerats eller om en vinst har hittats går koden vidare. Om en spelare vunnit kommer spelplanen i videominnet rensas och spelaren som vann kommer få sitt spelpoäng ökas med 1 och sedan skickas ett kommando till video-processorn med spelarnas poäng och kommandot för att rensa spelplanen. Om ingen vinst hittades fortsätter spelet som vanligt. Det blir då den andra spelarens tur att flytta runt och placera sin markör.

JSP är fullt rimligt här

2.5.2 Programmering av video-processorn

2.5.2.1 Struktur

Balanserat

Det första som sker i video-processorns kod är initialiseringen. Först och främst stackpekaren och sedan hårdvaran i form av att bestämma vilka portar som ska dedikeras till vad och om de ska skicka eller ta emot data. Sedan görs förberedelserna för kommunikation med UART-protokollet genom att sätta rätt flaggor för att matcha spel-processorns hastighet (se bilaga 3, s.31). Här aktiveras dessutom avbrott för processorn. Eftersom videominnet ligger i SRAM nollställs dessa värden i minnesinitialiseringen. Detta för att värdena är okända.

```
57 ;//////////  
58  
59 MAIN:  
60     rcall MEMORY_READ  
61     rcall SEND  
62     rcall INDEX_SHIFT  
63     rcall CHECK_NEXT_INS  
64     rjmp MAIN  
65  
66 ;//////////
```

Figur 2.7. Upplägget av videoprocessorns main.

Programmets main-loop (figur 2.7) lägger störst fokus på läsningen av minnet och överföring till displayen genom SPI. Displayen är multiplexad så överföringen sker kolumnvis per iteration av main-loopen. Efter att överföringen av data till aktuell kolumn på displayen skett blir index färdigt skiftat inför nästa runda i loopen. Sist men inte minst så görs en check för att bestämma om en ny instruktion från spel-processorn finns redo. Om inte; börja om loopen. Om ett nytt kommando finns så tolkas den och skrivas till videominnet.

2.5.2.2 Videominnet

För att på enklaste möjliga sätt styra vad som skulle skrivas till displayen användes ett "videominne" i SRAM. då kunde pekare användas för att smidigt läsa och skriva i minnet. Mycket tid gick åt att fastställa organiseringen av videominnet. Prioriteringen var att använda minnet på ett så effektivt sätt som möjligt, det menas att organisera minnet på ett sådant sätt att det tog upp lite plats och inte krävde någon form av "översättning" varje gång data skulle skickas till displayen. Det beslutades att minnet skulle ställas upp på följande sätt: 24 byte seriellt i SRAM, där varje byte motsvarar ett färgregister. Detta ger alltså seriellt röd, grön och blå byte följt av nästa uppsättning av tre färg-bytes. $24/3 = 8$ trios färg-data, där trions plats i videominnet representerar dess X-position på matrisen, det vill säga vilken kolumn de tre byten tillhör.

X ≠ X

På detta sätt krävs det endast att ladda in tre byte åt gången i en send-buffer och sedan skifta INDEX, processorns "mux-register", för att tända rätt punkter på en viss rad.

Processorn har en subrutin, MEMORY_READ, som löser inladdningen av dessa tre bytes.

KOMPÄKT

LOAD-COURIER

Den pekar ut den första trion och laddar in den i bufferten (**SEND_BYTE** i SRAM), redo för skrivning till displayen. Det krävs också att rutinen ökar och sparar sin position i minnet varje gång så att nästa tre bytes kan hämtas inför tändningen av nästa kolumn. Följande trio läses sedan in och index-byten skiftas för att lysa upp nästkommande rad. Detta leder till en relativt enkel och snabb multiplexad display. Som dessutom inte är så värt minnes-hungrig, då den endast tar upp 25 byte (inklusive INDEX).

2.5.2.3 Skrivning till Videominnet

För att skriva till videominnet finns en subrutin döpt till **MEMORY_WRITE**. Vad denna gör är att ta in en koordinat i form av en X- och Y-position, ett färgvärde och en parameter som berättar om dioden ska tändas eller släckas (se figur 2.8).

```

38      NEW_X_CORD: .byte 1
39      NEW_Y_CORD: .byte 1
40      ON_OFF: .byte 1 ;1=ON, 0=OFF
41      COLOR: .byte 1 ; 0=blue..2=red
42      NEW_Y_CORD_CONV: .byte 1 ;Y cord after CONVERT_CORDS call

```

Figur 2.8. Delen av SRAM dedikerad till parametrarna för subrutinen **MEMORY_WRITE**

Positionen anges som två decimala värden mellan 0-7. Till exempel \$02 och \$04 är alltså position (3,5) på displayen. Det är också på detta format koordinaterna skickas från spel-processorn, ingen extra översättning krävs alltså innan kallelse på write-rutinen. Färg-parametern består av ett decimalt tal mellan 0-2, där noll motsvarar blå och två är röd. Vad den betyder i praktiken är vilken position i en trio av färg-bytes man syftar på, då den första byten är den blåa byten och den sista är röd. ON/OFF-parametern är hyfsat självförlarande, den bestämmer om subrutinen ska tända eller släcka en position på displayen. Alltså skriva en etta eller nolla på avsedd bit. X-koordinaten kräver ingen översättning, allt den säger subrutinen är hur långt den ska stega genom videominnet för att peka på rätt uppsättning, trio, av färgbytes. Y-koordinaten behöver dessvärre lite processering. "Y-positionen" i SRAM är vilken bit i en färgbyte som syftas på. För att komma åt den decimala positionen (0-7) användes en *lookup-tabell* i cseg, SRAM. Se figur 2.7.

```

47  .cseg
48  LOOKUP: .db 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x00 ;For converting y-pos value for DAmatrix use
49

```

Figur 2.9. Lookup-tabell för att konvertera ett decimalt Y-värde till ett "Y-värde" som kan skrivas till displayen.

En subrutin, **CONVERT_CORDS**, stegar genom tabellen Y antal gånger och ger på så sätt en byte med en etta positionerad så att den motsvarar Y-koordinatens rad på displayen. Se figur 2.4 i kapitel 2.4.1 om displayen för en representation. Notera att displayen roterats jämfört med databladet för DAMatrix (Josefsson M. *Drivning av LED-matrisen DAMatrix*), där färgregisterna ligger under displayen och representerar kolumner istället för rader.

Önskad Y-position...	...Motsvarar
\$02	—————→ 00000100

2.5.2.4 Instruktionscheck och utförande

Processorn har alltså möjlighet att skriva och läsa data för att sedan styra displayen så det sista som behövde implementeras var användningen av dessa rutiner för att utföra kommandon från spel-processorn. Som tidigare nämnts används en enkelriktad seriell kommunikation mellan processorer. Se kapitel 2.4.4 om datakommunikation. Eftersom ATmega16 har inbyggt stöd för UART-protokollet ger det möjlighet till att använda avbrott vid mottagande av spel-processorns instruktioner. Processorn kan på detta sätt kalla på ett avbrott varje gång den mottagit data i UART-bufferten.

Detta avbrott läser sedan in datan och spara den under NEXT_INSTRUCTION i SRAM som består av tre byte. Den läser in datan sekventiellt i de tre byten efter att en startbyte, hexadecimalt \$FF, lästs in. Efter varje start-byte börjar den om att fylla NEXT_INSTRUCTION. Se bilaga 3 för avbrottets struktur. Det är sedan i processorns main-loop som CHECK_NEXT_INS körs. Denna rutin kollar med hjälp av en variabel i SRAM ifall tre stycken, på varandra följande, bytes lyckats läsas in. Den kan på så vis härleda att nästa kommando nu finns redo att avfärdas.

Först och främst måste den första instruktion byten testas för att kunna bestämma vilken typ av kommando det är frågan om. Värdet på byten säger ifall spelare N ska flytta markören, placera markören eller om hela spelplanen ska rensas och poängen visas på 7-segmenten. Den kan alltså förekomma som ett värde mellan \$00 och \$04, fem möjliga utfall. Detta bestämmer dessutom hur de två kommande byten data ska tolkas. Se tabellen i kapitel 2.4.4.1. Videominnet påverkas således.

3 Resultat

Det slutliga resultatet uppfyller alla grundkrav som bestämdes och de flesta utökade kraven. Projektet har alltså två fungerande joysticks för två spelare, spelarens tur indikeras av RGB-lysdioder som lyser i den nuvarande spelarens färg, när spelaren gör sitt val så piper en buzzer. Poäng visas på två enskilda 7-segments displayar, en för varje spelare. Spelplanen i resultatet består av en 8x8 RGB matris och kommunicerar via SPI kommunikation, allting utifrån specifikations kraven. Konstruktionen av 3D-utskrivna skal åt joysticken blev dock inte av eftersom det inte fanns tid till det.

Spelet är fullt spelbart och fungerar bra. Efter finjusteringar av spelhastigheten så fick vi en rörelse av markörerna som både kändes bra och reagerade lagom fort på användarens inmatningar från joysticken. Efter ett antal testspel så märktes att kombinationen av att markören kan placeras överallt på spelplanen och att det är fyra i rad resulterade i en strategi som tillät den startande spelaren att alltid vinna. En fördelaktligen flexibel kod tillät ökning av antalet i rad som krävs för en vinst. Detta lät oss testa spelet med även fem i rad som vinst tillstånd. Vilket de facto löste problemet med den ensidiga strategin, men resulterade istället i, möjligtvis, för långa runder.

4 Avslutande tankar

4.1 Misstag

Vid projektets början användes fel pin-layout för mikrodatorn Atmega16. Pin-nummer från den ytmonterade kapseln med 44 pinnar används istället för den rätta hålmonterade varianten med 40 pinnar. Detta resulterade i flera felkopplingar som potentiellt kunde skada olika komponenter. Den felaktiga pin-layouten skapade också en försening av projektet då det tog lång tid att få igång jTAG interfacet som ansvarar för programmeringen av mikrokontrollern. Efter en extra jämförelse mellan databladet och kretsschemat vid virning av kortet med video-processorn framkom problemet och kretsschemat ändrades omgående. Vi bytte också till en ny Atmega16A för spel-processorn då det verkade som att mikrokontrollerns A/D-omvandlare inte längre fungerade till fullo. Efter detta fungerade mikrokontrollern och jTAG-interfacet som förväntat.

Ett annat problem uppstod vid implementationen av BCD-omvandlarna och 7-segmentsdisplayerna. Vi förmodade att det fanns ett fel då vi upplevde att strömmen för kretsen var alldeles för hög, ~200 mA i vissa fall, då vi poängtavlorna var tända. Att BCD-omvandlarna avgav mycket värme ökade våra misstankar om att vi hade defekta komponenter. Det visade sig senare under projektets gång att denna strömförsörjning nämligen var inom specifikationerna och förväntat av våra 7-segmentsdisplayer då de var av en lite äldre modell. Återigen visade sig en extra läsning av databladet vara lösningen till problemet. *Ja dom blev överraskande varma!*

4.2 Möjliga förbättringar och avslutande diskussion

Något som uppstod vid flera tester av spelet mellan samma två spelare var en viss partiskhet till spelaren med första tur. Medan en förändring av ~~vinst tillståndet till fem~~ i rad löste problemet upplevdes spelet mera långtråkigt. En möjlig lösning skulle kunna vara ett mer "traditionellt" upplägg där spelpjäserna släpptes ner från toppen av skärmen och fylla tomma utrymmen på det sättet. Som figur 4.1 visar.



4.1. En mer traditionell version av fyra i rad

+text

Andra förbättringar skulle ha kunnat vara fler funktioner och större utbyggnad på spelet. Till exempel då vi använde oss av en RGB-matris skulle det ha varit mycket möjligt att låta spelarna välja en egen färg i början av spelet.

Vi implementerade också en buzzer för ljudeffekter när en spelare gör en placering på spelplanen. Det skulle ha varit mycket möjligt att bygga vidare på detta med flera ljudeffekter, till exempel vid vinst eller vid spelets början.

De skulle också vara en bra förbättring om spelet markerade på något sätt hur en spelare vinner. Att raden blinkar lite på displayen eller liknande.

Om man nu istället vill ändra spelet till fem i rad som vi testade så skulle det vara en stor fördel att ha en större spelplan än 8x8.

Om tiden funnits kunde dessutom utseendet av slutprodukten förbättras väsentligt. Vi hade funderingar kring 3D-utskrifter, ett av de utökade kraven var hållare för styrspakarna till exempel. Vi valde att stapla processorernas virningskort ovanpå varandra men om mer tid hade funnit till hade vi kanske kunnat 3D-printa en låda för dem istället. Detta kunde ha medfört ett snyggare och mer välarbetat utseende för slutprodukten.

Koden hade säkert kunnat optimeras på många olika sätt också.

Let's see...

5 Referenser

- [1] ATmega16A 8-bit Microcontroller
<https://www.microchip.com/wwwproducts/en/ATmega16A>
- [2] Josefsson M. Drivning av LED-matrisen DAMatrix
<https://docs.isy.liu.se/pub/VanHeden/DataSheets/ledmatris.pdf>
- [3] Fairchild Semiconductor (1998), DM9368 7-segment decoder
<https://docs.isy.liu.se/pub/VanHeden/DataSheets/9368.pdf>

6 Bilagor

Bilaga 1: Programkod för spel-processorn

```
/*
 * Spelprocessor.asm
 *
 * Created: 2020-02-04 09:29:40
 * Author: lincl896
 */

// Main filen

.include "Joystick_driver.asm"
.include "game_logic.asm"
. equ VMEM_SIZE = 64
. equ DELAY_HIGH = 150 ;100 blir bra
. equ DELAY_LOW = 0
. equ BEEP_LENGTH_H = $0f
. equ BEEP_LENGTH_L = $00

.dseg
.org SRAM_START
P1X: .byte 1; Player 1 X position
P1Y: .byte 1; Player 1 Y position
P2X: .byte 1; Player 2 X position
P2Y: .byte 1; Player 2 Y position

Start[Byte]: .byte 1
Command[Byte]: .byte 1
Argument1[Byte]: .byte 1
Argument2[Byte]: .byte 1

Player1_Score: .byte 1
Player2_Score: .byte 1

Win: .byte 1 ; Player1 = 1, Player2 = 2

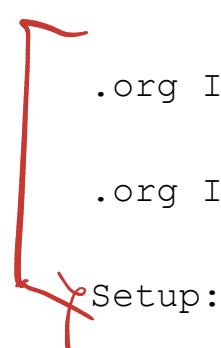
DEBUG: .byte 5

VMEM: .byte VMEM_SIZE

.cseg
.org $0000
    rjmp Setup
.org INT0addr
    rjmp Interrupt0
.org INT1addr
    rjmp Interrupt1
Setup:
    ldi r16, HIGH(RAMEND)
    out SPH, r16
```

BRA FÖN
O. RADAVSTÅND

// Snyggt
Strukturerat
kod



```

ldi r16, LOW(RAMEND)
out SPL,r16

Hardware_Init:
    rcall Joystick_Init
    ldi r16, (1<<ISC11) | (0<<ISC10) | (1<<ISC01) | (0<<ISC00)
    out MCUCR,r16
    ldi r16, (1<<INT0) | (1<<INT1)
    out GICR,r16

    ldi r16, 0b11100000
    out DDRD,r16
    ldi r16, 0b00011111
    out PORTD,r16

    ldi r16, 0b00000001
    out DDRB,r16

Usart_Init:
    ; Se sida 143 i databladet

    ; Set baud rate
    clr r17
    ldi r16,$0c
    out UBRRH,r17
    out UBRRL,r16

    ; Enable receiver and transmitter
    ldi r16, (1<<TXEN)
    out UCSRB,r16

    ldi r16, (1<<U2X)
    out UCSRA,r16

    ; Set frame format: 8data, 1stop bit
    ldi r16, (1<<URSEL) | (3<<UCSZ0)
    out UCSRC,r16

    sei

SRAM_Init:
    ; start byte config
    ldi r16,$ff
    sts Start_Byt,e,r16
    clr r16
    sts Player1_Score,r16
    sts Player2_Score,r16
    sts Win,r16
    ; Clear board
    rcall Clear_Board

Main:
    rcall Player_Input;
    rcall Delay
    rjmp Main

```

Namngivna
 för att under -
 lätta för läsaren

```

SendByte:
    ; Se sida 144 i databladet

    ; Wait for empty transmit buffer
    sbis UCSRA,UDRE
    rjmp SendByte

    ; Put data (r16) into buffer, sends the data
    out UDR,r16

    ret

Send_Data:
    push r16
    lds r16,Start_Byt
    rcall SendByte          ; Start of message

    lds r16,Command_Byt
    rcall SendByte          ; message type: 0 if player selects, 1 if player 1
selects

    lds r16,Argument1_Byt
    rcall SendByte          ; X-value

    lds r16,Argument2_Byt
    rcall SendByte          ; Y-value
    pop r16
    ret

//Player 1 = t 0, Player 2 = t 1

Player_Input:
    cli
    push r17
    brts Player2

Player1:
    ldi r16,0b11000000
    out PORTA,r16
    ldi r16,0b11111111
    out PORTD,r16

    rcall Input_P1
    ldi r17,$00
    rjmp Input_done

Player2:
    ldi r16,0b01111111
    out PORTD,r16
    ldi r16,0b11100000
    out PORTA,r16

    rcall Input_P2
    ldi r17,$01

Input_done:
    rcall Send_Player_Choice
    rcall Send_Data
    pop r17
    sei
    ret

```

```

    ret

    } DELAY:
        ldi r25,DELAY_HIGH
        ldi r24,DELAY_LOW

    DELAY2:
        sbiw r24,1
        brne DELAY2
        ret

    } Load_X_Pointer:
        ldi XH,HIGH(VMEM)
        ldi XL,LOW(VMEM)
        ret

    } Calculate_Pos:
        push r16
        push r17
        add XL,r16
        lsl r17
        lsl r17
        lsl r17
        adc XL,r17
        pop r17
        pop r16
        ret

//Player 1 joystick push button interrupt
Interrupt0:                                SREG??
    push r16
    push r17

    brts P1_DONE
    lds r16,P1X
    lds r17,P1Y

    rcall Load_X_Pointer
    rcall Calculate_Pos

    ld r17,X
    cpi r17,1
    breq P1_DONE
    cpi r17,2
    breq P1_DONE
    rcall BEEP
    ldi r17,1
    st X,r17
    ldi r17,2
    rcall Send_Player_Choice
    rcall Send_Data
    rcall Check_for_end_of_game
    set

P1_DONE:

```

```
    pop r17  
    pop r16  
    reti
```

```
//Player 2 joystick push button interrupt  
Interrupt1:
```

```
    push r16  
    push r17  
    brtc P2_DONE  
    lds r16, P2X  
    lds r17, P2Y  
  
    rcall Load_X_Pointer  
    rcall CALCULATE_POS  
  
    ld r17, X  
    cpi r17, 1  
    breq P2_DONE  
    cpi r17, 2  
    breq P2_DONE  
    rcall BEEP  
    ldi r17, 2  
    st X, r17  
    ldi r17, 3  
    rcall Send_Player_Choice  
    rcall Send_Data  
    rcall Check_for_end_of_game  
    clt
```

```
P2_DONE:
```

```
    pop r17  
    pop r16  
    reti
```

```
Send_Player_Choice:
```

```
    sts Command_Byt, r17  
    brts Send_Player_2
```

```
Send_Player_1:
```

```
    lds r17, P1X  
    sts Argument1_Byt, r17  
    lds r17, P1Y  
    sts Argument2_Byt, r17  
    rjmp Send_Done
```

```
Send_Player_2:
```

```
    lds r17, P2X  
    sts Argument1_Byt, r17  
    lds r17, P2Y  
    sts Argument2_Byt, r17
```

```
Send_Done:
```

```
    ret
```

```
BEEP:  
    push r16  
    push r17  
    in r17, SREG  
  
    ldi r16, 0b00000001  
    out PORTB, r16  
    push r24  
    push r25  
    ldi r24, BEEP_LENGTH_L  
    ldi r25, BEEP_LENGTH_H  
    rcall DELAY2  
    pop r25  
    pop r24  
  
BEEP_DONE:  
    ldi r16, 0b00000000  
    out PORTB, r16  
    out SREG, r17  
    pop r17  
    pop r16  
    ret
```

Behörs info om hur alt är uppdelat

Bilaga 2: Spellogik (spel-processorn)

.org \$600



VARFOR \$600?

.equ Checks_Needed_for_a_win = \$04

Check_for_end_of_game:

```
    push r16
    rcall Check_for_full_map
    pop r16
    clr r0
    cpi r16,1 ; om banan är full
    breq Clear_Board
```

Check_Win:

```
    rcall Check_Horizontal
    lds r16,Win
    cpi r16,1
    breq Check_Winner
```

```
    rcall Check_Vertical
    lds r16,Win
    cpi r16,1
    breq Check_Winner
```

```
    rcall Right_Diagonal_Check
    lds r16,Win
    cpi r16,1
    breq Check_Winner
```

```
    rcall Left_Diagonal_Check
    lds r16,Win
    cpi r16,1
    breq Check_Winner
```

```
    rcall Check_Vertical
    lds r16,Win
    cpi r16,1
    breq Check_Winner
```

```
    lds r16,Win
    cpi r16,1
    breq Check_Winner
    rjmp Check_Done
```

Check_Winner:

```
    brts Player2_Wins
```

Player1_Wins:

```
    lds r16,Player1_Score
    inc r16
```

```

sts Player1_Score,r16
rjmp Clear_Board

Player2_Wins:
    lds r16,Player2_Score
    inc r16
    sts Player2_Score,r16

Clear_Board:
    ldi r17,$04

    sts Command_Byt,e,r17
    lds r17,Player1_Score
    sts Argument1_Byt,e,r17
    lds r17,Player2_Score
    sts Argument2_Byt,e,r17

    clr r16
    sts P1X,r16
    sts P1Y,r16
    sts P2X,r16
    sts P2Y,r16

    sts Win,r16

    clr r17
    rcall Load_X_Pointer
Clear_Loop:
    st X+,r16
    inc r17
    cpi r17,VMEM_SIZE
    brne Clear_Loop

    rcall Send_Data

Check_Done:
    ret

Check_for_full_map:
    push r16
    push r18 ;loop count
    in ZH,SPH
    in ZL,SPL
    rcall Load_X_Pointer
    clr r18
    ; 1 if the map is full, 0 if there is still space.

Loop:
    ld r16,X+
    cpi r16,0
    breq Not_full
    cpi r18,VMEM_SIZE
    inc r18
    brne Loop
    ldi r16,1
    rjmp Loop_done

```

Det var
en hiskeligt
lang rutin

Det var
en hiskeligt
lang rutin

```

Not_full:
    clr r16
Loop_done:
    adiw Z,5
    st Z,r16
    pop r18
    pop r16

    ret

Check_Horizontal:
    push r16
    push r17
    push r18
    push r20
    push r21
    clr r21
    rcall Load_X_Pointer
    brts Horizontal_Player_2
Horizontal_Player_1:
    lds r16,P1X
    lds r17,P1Y
    ldi r18,1
    rcall CALCULATE_POS

    lds r20,P1Y
    rjmp Right
Horizontal_Player_2:
    lds r16,P2X
    lds r17,P2Y
    ldi r18,2
    rcall CALCULATE_POS

Right:
    cpi r16,7
    breq Check_Left
    inc r16
    inc XL
    ld r19,X
    cp r19,r18
    brne Check_Left
    inc r21

    cpi r21,Checks_Needed_for_a_win
    brne Right

    rjmp Horizontal_Check_Done

Check_Left:
    brts Horizontal_Player_2_2

Horizontal_Player1_2:
    lds r16,P1X
    lds r17,P1Y
    rcall Load_X_Pointer
    rcall CALCULATE_POS

```

Det var
en hiskelligt
lang rutin

```

rjmp Left

Horizontal_Player_2_2:
    lds r16, P2X
    lds r17, P2Y
    rcall Load_X_Pointer
    rcall CALCULATE_POS

Left:
    cpi r16, 0
    breq Horizontal_No_Win
    dec r16

    dec XL
    ld r19, X
    cp r19, r18
    brne Horizontal_No_Win
    inc r21

    cpi r21, Checks_Needed_for_a_win
    brne Left
    rjmp Horizontal_Check_Done

Horizontal_Check_Done:
    ldi r16, $01
    rjmp Horizontal_Done

Horizontal_No_Win:
    clr r16

Horizontal_Done:
    sts Win, r16
    pop r21
    pop r20
    pop r18
    pop r17
    pop r16
    ret

Check_Vertical:
    push r16
    push r17
    push r18
    push r20
    push r21
    clr r21
    rcall Load_X_Pointer
    brts Vertical_Player_2

Vertical_Player_1:
    lds r16, P1X
    lds r17, P1Y
    ldi r18, 1
    rcall CALCULATE_POS
    lds r20, P1Y
    rjmp Up

Vertical_Player_2:

```

Det var
 en hansklig
 lang rutn

```

lds r16,P2X
lds r20,P2Y
ldi r18,2
rcall CALCULATE_POS
lds r20,P2Y

```

Up:

```

cpi r20,0
breq Check_Down
dec r20

ldi r19,8
sub XL,r19
sbc XH,r0
ld r19,X
cp r19,r18
brne Check_Down
inc r21

cpi r21,Checks_Needed_for_a_win
brne Up
rjmp Vertical_Check_Done

```

Check_Down:

```
brts Vertical_Player_2_2
```

Vertical_Player1_2:

```

lds r16,P1X
lds r17,P1Y
rcall Load_X_Pointer
rcall CALCULATE_POS
rjmp Down

```

Vertical_Player_2_2:

```

lds r16,P2X
lds r17,P2Y
rcall Load_X_Pointer
rcall CALCULATE_POS

```

Down:

```

cpi r17,7
breq Vertical_No_Win
inc r17

```

```

ldi r16,8
add XL,r16
adc XH,r0
ld r16,X
cp r16,r18
brne Vertical_No_Win
inc r21

```

```

cpi r21,Checks_Needed_for_a_win
brne Down
rjmp Vertical_Check_Done

```

```

Vertical_Check_Done:
    ldi r16,$01
    rjmp Vertical_Done

Vertical_No_Win:
    clr r16

Vertical_Done:
    sts Win,r16
    pop r21
    pop r20
    pop r18
    pop r17
    pop r16
    ret

Right_Diagonal_Check:
    push r16
    push r17
    push r18
    push r20
    push r21
    push r22
    clr r21
    rcall Load_X_Pointer
    brts Right_Diagonal_Player_2

Right_Diagonal_Player_1:
    lds r16,P1X
    lds r17,P1Y
    ldi r18,1
    rcall CALCULATE_POS
    rjmp Up_Right

Right_Diagonal_Player_2:
    lds r16,P2X
    lds r17,P2Y
    ldi r18,2
    rcall CALCULATE_POS

    rjmp Up_Right

Up_Right:
    cpi r17,0
    breq Check_Down_Left
    cpi r16,7
    breq Check_Down_Left
    dec r17
    inc r16

    ldi r20,8
    sub XL,r20
    sbc XH,r0
    inc XL
    ld r20,X
    cp r20,r18
    brne Check_Down_Left
    inc r21

```

```

cpi r21,Checks_Needed_for_a_win
brne Up_Right
rjmp Right_Diagonal_Check_Done

Check_Down_Left:
    brts Right_Diagonal_Player_2_2

Right_Diagonal_Player1_2:
    lds r16,P1X
    lds r17,P1Y
    rcall Load_X_Pointer
    rcall CALCULATE_POS
    rjmp Down_Left

Right_Diagonal_Player_2_2:
    lds r16,P2X
    lds r17,P2Y
    rcall Load_X_Pointer
    rcall CALCULATE_POS

Down_Left:
    cpi r17,7
    breq Right_Diagonal_No_Win
    cpi r16,0
    breq Right_Diagonal_No_Win
    inc r17
    dec r16

    breq Right_Diagonal_No_Win
    ldi r20,8
    add XL,r20
    adc XH,r0
    dec XL
    ld r20,X
    cp r20,r18
    brne Right_Diagonal_No_Win
    inc r21

    cpi r21,Checks_Needed_for_a_win
    brne Down_Left

Right_Diagonal_Check_Done:
    ldi r16,$01
    rjmp Right_Diagonal_Done

Right_Diagonal_No_Win:
    clr r16

Right_Diagonal_Done:
    sts Win,r16
    pop r22
    pop r21
    pop r20
    pop r18
    pop r17

```

```

pop r16
ret

Left_Diagonal_Check:
    push r16
    push r17
    push r18
    push r20
    push r21
    clr r21
    rcall Load_X_Pointer
    brts Left_Diagonal_Player_2
Left_Diagonal_Player_1:
    lds r16, P1X
    lds r17, P1Y
    ldi r18, 1
    rcall CALCULATE_POS
    rjmp Up_Left
Left_Diagonal_Player_2:
    lds r16, P2X
    lds r17, P2Y
    ldi r18, 2
    rcall CALCULATE_POS

Up_Left:
    cpi r17, 0
    breq Check_Down_Right
    cpi r16, 0
    breq Check_Down_Right
    dec r17
    dec r16

    ldi r20, 8
    sub XL, r20
    sbc XH, r0
    dec XL
    ld r20, X
    cp r20, r18
    brne Check_Down_Right
    inc r21

    cpi r21, Checks_Needed_for_a_win
    brne Up_Left
    rjmp Left_Diagonal_Check_Done

Check_Down_Right:
    brts Left_Diagonal_Player_2_2

Left_Diagonal_Player1_2:
    lds r16, P1X
    lds r17, P1Y
    rcall Load_X_Pointer
    rcall CALCULATE_POS
    rjmp Down_Right

```

```

Left_Diagonal_Player_2_2:
    lds r16, P2X
    lds r17, P2Y
    rcall Load_X_Pointer
    rcall CALCULATE_POS

Down_Right:
    cpi r17, 7
    breq Left_Diagonal_No_Win
    cpi r16, 7
    breq Left_Diagonal_No_Win
    inc r17
    inc r16

    ldi r20, 8
    add XL, r20
    adc XH, r0
    inc XL
    ld r20, X
    cp r20, r18
    brne Left_Diagonal_No_Win
    inc r21

    cpi r21, Checks_Needed_for_a_win
    brne Down_Right

Left_Diagonal_Check_Done:
    ldi r16, $01
    rjmp Left_Diagonal_Done

Left_Diagonal_No_Win:
    clr r16

Left_Diagonal_Done:
    sts Win, r16
    pop r21
    pop r20
    pop r18
    pop r17
    pop r16
    ret

```

Ny sida

Bilaga 3: Joystick driver (spel-processorn)

```

.equ Channel_P1_X = $01
.equ Channel_P1_Y = $00
.equ Channel_P2_X = $03
.equ Channel_P2_Y = $02

.equ Y_UP = 3

```

```

.equ Y_DOWN = 0
.equ X_LEFT = 0
.equ X_RIGHT = 3
.equ Neutral = 2

.macro INCSRAM      ; inc byte in SRAM
    lds r16,@0
    inc r16
    sts @0,r16
.endmacro

.macro DECSRAM      ; dec byte in SRAM
    lds r16,@0
    dec r16
    sts @0,r16
.endmacro

```

.org \$500

Joystick_Init:

```

ldi r16, 0b11100000
out DDRA,r16
ret

```

Input:

```

out ADMUX,r16

```

```

ldi r16, (1<<ADSC) | (1<<ADEN)
out ADCSRA,r16

```

Wait:

```

sbic ADCSRA,ADSC
rjmp Wait

```

```

in r16,ADCH

```

```

ret

```

Input_P1:

```

push r16

```

```

ldi r16, (0<<REFS1) | (0<<REFS0) | (0<<ADLAR) | (Channel_P1_X)
rcall Input
rcall Check_X1

```

```

ldi r16, (0<<REFS1) | (0<<REFS0) | (0<<ADLAR) | (Channel_P1_Y)
rcall Input
rcall Check_Y1

```

```

pop r16

```

```

ret

```

```

Input_P2:
    push r16

    ldi r16, (0<<REFS1) | (0<<REFS0) | (0<<ADLAR) | (Channel_P2_X)
    rcall Input
    rcall Check_X2

    ldi r16, (0<<REFS1) | (0<<REFS0) | (0<<ADLAR) | (Channel_P2_Y)
    rcall Input
    rcall Check_Y2

    pop r16

    ret

Check_X1:
    //Forward X
    cpi r16, X_LEFT
    breq X1_INC
    cpi r16, X_RIGHT
    breq X1_DEC

    rjmp X1_DONE

X1_INC:
    INCSRAM P1X
    // se till att X är mindre än 8 här
    lds r16, P1X
    cpi r16, 8
    breq X1_DEC //minskar P1X

    rjmp X1_DONE

X1_DEC:
    DECSRAM P1X
    // se till att X är större än -1 här
    lds r16, P1X
    cpi r16, 255
    breq X1_INC //öka P1X

    rjmp X1_DONE

X1_DONE:
    ret

Check_X2:
    //Forward X
    cpi r16, X_LEFT
    breq X2_INC
    cpi r16, X_RIGHT
    breq X2_DEC

```

```

rjmp X2_DONE

X2_INC:
    INCSRAM P2X
    // se till att X är mindre än 8 här
    lds r16, P2X
    cpi r16, 8
    breq X2_DEC//minska P1X

    rjmp X2_DONE

X2_DEC:
    DECSRAM P2X
    // se till att X är större än -1 här
    lds r16, P2X
    cpi r16, 255
    breq X2_INC//öka P1X

    rjmp X2_DONE

```

```

X2_DONE:
    ret

```



```

Check_Y1:
    cpi r16, Y_UP
    breq Y1_INC
    cpi r16, Y_DOWN
    breq Y1_DEC

    rjmp Y1_DONE

Y1_INC:
    INCSRAM P1Y
    // se till att X är mindre än 8 här
    lds r16, P1Y
    cpi r16, 8
    breq Y1_DEC//minska P1X

    rjmp Y1_DONE

Y1_DEC:
    DECSRAM P1Y
    // se till att X är större än -1 här
    lds r16, P1Y
    cpi r16, 255
    breq Y1_INC//öka P1X

    rjmp Y1_DONE

Y1_DONE:
    ret

```

Check_Y2:

```
cpi r16, Y_UP  
breq Y2_INC  
cpi r16, Y_DOWN  
breq Y2_DEC  
  
rjmp Y2_DONE
```

Y2_INC:

```
INCSRAM P2Y  
// se till att X är mindre än 8 här  
lds r16, P2Y  
cpi r16, 8  
breq Y2_DEC //minskar P1X
```

```
rjmp Y2_DONE
```

Y2_DEC:

```
DECSRAM P2Y  
// se till att X är större än -1 här  
lds r16, P2Y  
cpi r16, 255  
breq Y2_INC //ökar P1X
```

```
rjmp Y2_DONE
```

Y2_DONE:

```
ret
```

Kan
denna kan
göras kortare

Bilaga 4: Programkod för video-processorn

```
.include "UART.asm"  
  
.equ RED = 0x02  
.equ GREEN = 0x01  
.equ BLUE = 0x00  
  
.org 0x00  
    rjmp START  
  
.org 0x016  
    rjmp RECEIVE  
  
.dseg  
    .org 0x300  
  
SEND_BYTE: .byte 4  
SEND_BUFF: .byte 1  
LOOP: .byte 1
```

```

ROWS: .byte 24 ;rgb, rgb, rgb...
INDEX: .byte 1 ;register with shifting 0
ROW_POS: .byte 1 ;ROW_POS*3 = next row of rgb

NEXT_INSTRUCTION: .byte 3 ;UART bytes
CURR_INS_BYTE: .byte 1 ;keeps track of bytes in instruction

OLD_X_CORD: .byte 1
OLD_Y_CORD: .byte 1

NEW_X_CORD: .byte 1
NEW_Y_CORD: .byte 1
ON_OFF: .byte 1 ;1=ON, 0=OFF
COLOR: .byte 1 ; 0=blue..2=red
NEW_Y_CORD_CONV: .byte 1 ;Y cord after CONVERT_CORDS call

ROWS_BACKUP: .byte 24 ;rgb, rgb, rgb...

```

```

.cseg
LOOKUP: .db 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x00
;For converting y-pos value for DAmatrix use

```

```

START:
    ldi r16,high(RAMEND)
    out SPH,r16

    ldi r16,low(RAMEND)
    out SPL,r16

    rcall INIT
    rcall UART_INIT
    rcall MEMORY_INIT

```

```
;//////////
```

```

MAIN:
    rcall MEMORY_READ
    rcall SEND
    rcall INDEX_SHIFT
    rcall CHECK_NEXT_INS
    rjmp MAIN

```

```
;//////////
```

```

SEND:
;//Sends byte to spi and rcalls LOAD_DATA 4x, then resets SEND_BYT pointer
    ldi ZH, high(SEND_BYT)
    ldi ZL, low(SEND_BYT)

```

```

SEND_LOOP:
    lds r16, LOOP
    cpi r16, 0x04
    breq RESET_PTR
    rcall LOAD_DATA
    lds r16, SEND_BUFF
    out SPDR, r16 ;SEND DATA

```

```

WAIT:
;//Checks if shifting of byte to display is done
    sbis SPSR,SPIF
    rjmp WAIT
    rjmp SEND_LOOP

```

SĀ HĀR!

```

RESET_PTR:
    rcall PULL_LATCH
    ldi ZH, high(SEND_BYTE)
    ldi ZL, low(SEND_BYTE)
    clr r16
    sts LOOP, r16

    ret

```

```

MEMORY_READ:
; //Reads ROWS in sram and stores in right order in SEND_BYTE
    rcall Load_Rows

```

```

    lds r16, ROW_POS
    cpi r16, 24
    brne ADD_POS
    clr r16

```

```

ADD_POS:
    add XL, r16

```

```

NOT_END:
    ld r18, X++
    sts SEND_BYTE, r18

```

```

    ld r18, X++
    sts SEND_BYTE+1, r18

```

```

    ld r18, X
    sts SEND_BYTE+2, r18

```

```

    ldi r17, 0x03
    add r16, r17
    sts ROW_POS, r16

```

```

    ret

```

OVANLIG
 INDENTERING
 Dock...

```

INDEX_SHIFT:
; Shifts the zero-bit in INDEX
    lds r16, INDEX
    cpi r16, 0x7f
    breq NO_SET_CARRY
    sec

```

```

NO_SET_CARRY:
    rol r16
    sts INDEX, r16
    sts SEND_BYTE + 3, r16

```

```

    ret

```

```

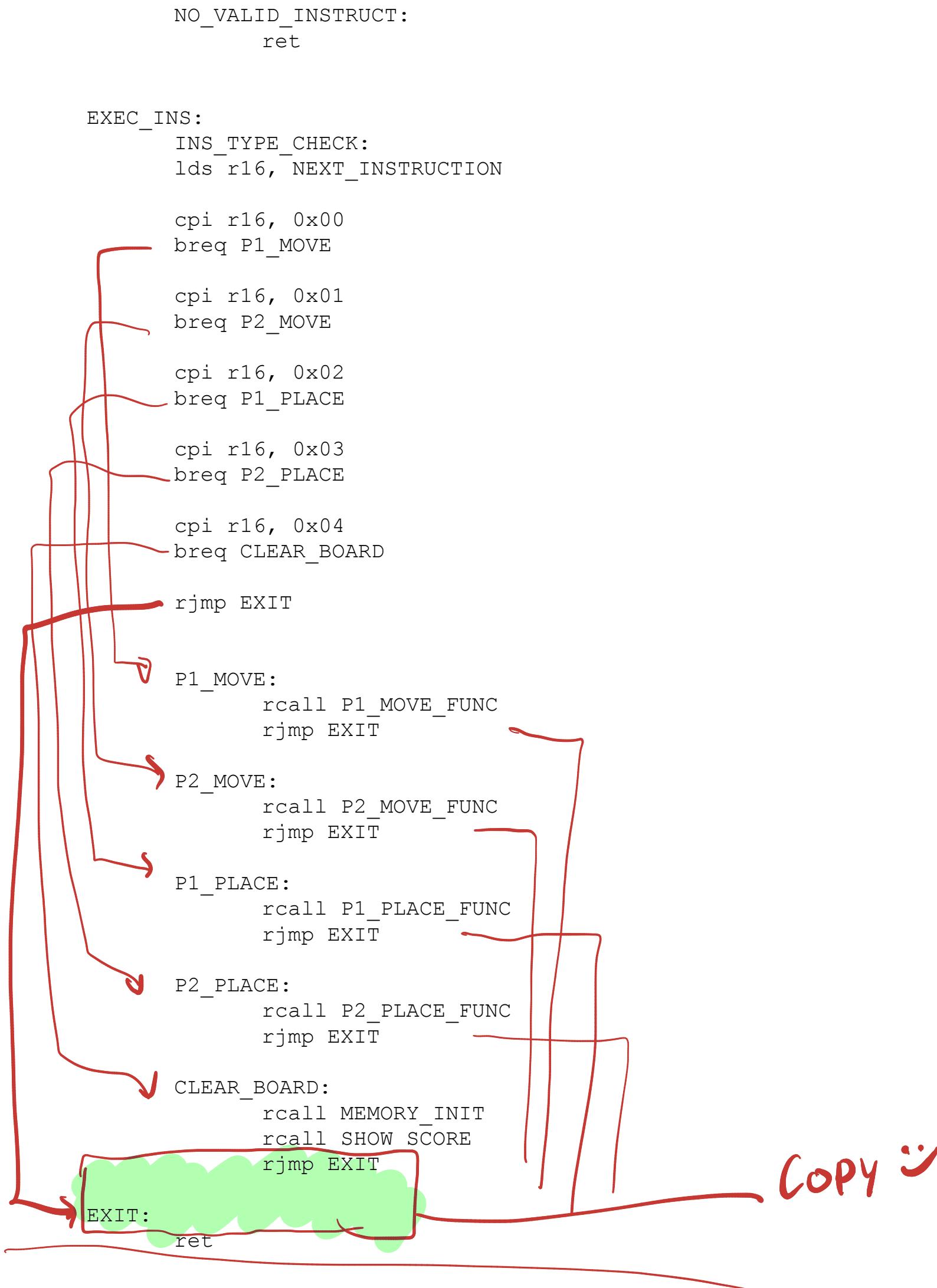
CHECK_NEXT_INS:
; Check if a full, 3 byte, valid instruction has been read.
; If so, execute instruction
    lds r16, CURR_INS_BYTE
    cpi r16, 0x03
    brne NO_VALID_INSTRUCT

```

```

EXECUTE_INSTRUCTION:
    rcall EXEC_INS

```



```

P1_MOVE_FUNC:
; Moves player 1 marker
OLD_P1_OFF:
    rcall RESTORE_ROW

NEW_P1_ON:
    rcall STORE_ROW
    lds r16, NEXT_INSTRUCTION+1
    sts NEW_X_CORD, r16
    sts OLD_X_CORD, r16

    lds r16, NEXT_INSTRUCTION+2
    sts NEW_Y_CORD, r16
    sts OLD_Y_CORD, r16

```

```
ldi r16, RED
sts COLOR, r16
ldi r16, 0x01
sts ON_OFF, r16
rcall MEMORY_WRITE

ldi r16, BLUE
sts COLOR, r16
ldi r16, 0x01
sts ON_OFF, r16
rcall MEMORY_WRITE

ret
```



```
P2_MOVE_FUNC:
;Moves player 2 marker
OLD_P2_OFF:
    rcall RESTORE_ROW
```

```
NEW_P2_ON:
    lds r16, NEXT_INSTRUCTION+1
    sts NEW_X_CORD, r16

    lds r16, NEXT_INSTRUCTION+2
    sts NEW_Y_CORD, r16

    rcall STORE_ROW

    ldi r16, RED
    sts COLOR, r16
    ldi r16, 0x01
    sts ON_OFF, r16
    rcall MEMORY_WRITE

    ldi r16, BLUE
    sts COLOR, r16
    ldi r16, 0x01
    sts ON_OFF, r16
    rcall MEMORY_WRITE
```

```
ret
```



```
P2_PLACE_FUNC:
;Place player 2 marker
    rcall RESTORE_ROW

    lds r16, NEXT_INSTRUCTION+1
    sts NEW_X_CORD, r16

    lds r16, NEXT_INSTRUCTION+2
    sts NEW_Y_CORD, r16

    ldi r16, BLUE
    sts COLOR, r16
    ldi r16, 0x01
    sts ON_OFF, r16

    rcall MEMORY_WRITE
    rcall STORE_ROW
```

```
ret
```



```

P1_PLACE_FUNC:
;Place player 1 marker
    rcall RESTORE_ROW

    lds r16, NEXT_INSTRUCTION+1
    sts NEW_X_CORD, r16

    lds r16, NEXT_INSTRUCTION+2
    sts NEW_Y_CORD, r16

    ldi r16, RED
    sts COLOR, r16
    ldi r16, 0x01
    sts ON_OFF, r16

    rcall MEMORY_WRITE
    rcall STORE_ROW

    ret

SHOW_SCORE:
;Sends new player scores to 7-segments
    lds r16, NEXT_INSTRUCTION+1
    lds r17, NEXT_INSTRUCTION+2
    clr r18
    swap r17
    or r16, r17
    out PORTA, r16

    ret

LOAD_DATA:
;//Loads next byte in SEND_BYTES into send buffer (SEND_BUFF)
    ld r16, Z++
    sts SEND_BUFF, r16

    lds r16, LOOP
    inc r16
    sts LOOP, r16

    ret

PULL_LATCH:
;//Update display after byte shifting done
    sbi PORTB, 4
    nop
    lpm ← SAY WHAT?
    lpm
    cbi PORTB, 4
    ret

Calculate_Position:
;Calculates where to point in rows based on x-coordinate given
    push r18
    push r17
    push r16
    clr r16
    lds r18, NEW_X_CORD
    ldi r17, 0x03
    cpi r18, 0x00
    breq Calc_Done

```

```

Calc_Loop:
    inc r16
    add XL,r17
    cp r16,r18
    brne Calc_Loop

Calc_Done:
    pop r16
    pop r17
    pop r18
    ret

Calculate_Backup_Position:
    push r18
    push r17
    push r16
    clr r16
    lds r18, NEW_X_CORD
    ldi r17,0x03
    cpi r18, 0x00
    breq Calc_Done
Calc_Backup_Loop:
    inc r16
    add YL,r17
    cp r16,r18
    brne Calc_Backup_Loop

Calc_Backup_Done:
    pop r16
    pop r17
    pop r18
    ret

```



```

MEMORY_WRITE:
; //Writes to ROWS in sram
; Parameters: ON_OFF, COLOR and X- and Y-coordinates

    rcall CONVERT_CORDS

    rcall Load_Rows
    rcall Calculate_Position

    ON_OFF_CHECK:
        lds r16, ON_OFF
        cpi r16, 0x00
        breq TURN_OFF

    TURN_ON:
        lds r17, COLOR
        add XL, r17

        ld r16, X
        lds r17, NEW_Y_CORD_CONV

        or r16, r17
        st X, r16
        rjmp Memory_Write_Done

    TURN_OFF:
        lds r17, COLOR

```

```

        add XL, r17

        ld r16, X
        lds r17, NEW_Y_CORD_CONV

        rcall OFF_CHECK
        brcc MEMORY_WRITE_DONE

        eor r16, r17
        st X, r16

MEMORY_WRITE_DONE:
    ret

CONVERT_CORDS:
; Converts unformatted Y-coordinate to a true bit in a byte
    lds r16, NEW_Y_CORD

    ldi ZH, high(LOOKUP*2)
    ldi ZL, low(LOOKUP*2)

    add ZL, r16
    lpm r16, Z

    sts NEW_Y_CORD_CONV, r16
    ret

OFF_CHECK:
; Checks whether position is already lit
    push r16
    push r17

    lds r17, NEW_Y_CORD

SHIFT_TO_CARRY:
    ror r16
    cpi r17, 0x00
    breq OFF_CHECK_DONE
    dec r17
    rjmp SHIFT_TO_CARRY

OFF_CHECK_DONE:
    pop r17
    pop r16
    ret

Load_Rows:
    ldi XH, HIGH(ROWS)
    ldi XL, LOW(ROWS)
    ret

Load_Rows_Protect:
    ldi YH, high(ROWS_BACKUP)
    ldi YL, low(ROWS_BACKUP)
    ret

RESTORE_ROW:
; //WRITES TO ROWS
    push XH

```

```

push XL
push YH
push YL

rcall Load_Rows
rcall Load_Rows_Protect
rcall Calculate_Position
rcall Calculate_Backup_Position

ld r16, Y+
st X+, r16
ld r16, Y+
st X+, r16
ld r16, Y
st X, r16

pop YL
pop YH
pop XL
pop XH
ret

```

```

STORE_ROW:
; //WRITES TO ROWS_BACKUP
push XH
push XL
push YH
push YL

rcall Load_Rows
rcall Load_Rows_Protect
rcall Calculate_Position
rcall Calculate_Backup_Position

ld r16, X+
st Y+, r16
ld r16, X+
st Y+, r16
ld r16, X
st Y, r16

pop YL
pop YH
pop XL
pop XH
ret

```

```

INIT:
; //SPI initalization
ldi r16, 0xB0
out DDRB, r16

ldi r16, (1<<SPE | 1<<MSTR | 0<<SPIE | 0<<SPR0)
out SPCR, r16

ldi r16, 0xff
out DDRA, r16

clr r16
out PORTA, r16

ret

```

```
MEMORY_INIT:  
;Memory initialization  
    ldi r16, 0b11111110  
    sts INDEX, r16  
  
    clr r16  
    sts ROW_POS, r16  
    sts ON_OFF, r16  
  
CLEAR_MEM:  
    clr r16  
    clr r17  
    rcall Load_Rows  
    CLR_LOOP:  
        inc r17  
        st X+,r16  
        cpi r17, 24  
        brne CLR_LOOP  
  
CLEAR_MEM_BACKUP:  
    clr r16  
    clr r17  
    rcall Load_Rows_Protect  
    CLR_LOOP_BACKUP:  
        inc r17  
        st Y+,r16  
        cpi r17, 24  
        brne CLR_LOOP_BACKUP  
  
ret
```

Bilaga 4: UART implementation (video-processor)

```
.org 0x500

UART_INIT:
    ldi r17, 0x00
    ldi r16, 0x0c

; Set baud rate
    out UBRRH, r17
    out UBRRL, r16

; Receiver and receiver-interrupt enable
    ldi r16, (1<<RXEN | 1<<RXCIE)
    out UCSRB, r16

; Set frame format: 8data, 1stop bit
    ldi r16, (1<<URSEL | 3<<UCSZ0)
    out UCSRC, r16

; Double asynchronous speed
    ldi r16, (1<<U2X)
    out UCSRA, r16

    clr r16
    sts CURR_INS_BYTE, r16

    sei
    ret

RECEIVE:
    push ZH
    push ZL
    push r19
    push r18
    push r17
    push r16
    in r16, SREG
    push r16

READ:
;READ UART BUFFERS
    in r16, UCSRA
    in r17, UCSRB
    in r18, UDR

;ERROR CHECK
    andi r16, (1<<FE | 1<<DOR | 1<<PE)
    breq NO_ERROR
    rjmp DATA_RECEIVED

NO_ERROR:
;FILTER STOP-BIT
    lsr r17
    andi r17, 0x01

STORE:
;CHECK START BYTE AND STORE
```

```

lds r19,CURR_INS_BYTE
cpi r19,0x03
brne NO_RESET
clr r19
sts CURR_INS_BYTE,r19

NO_RESET:
cpi r18, 0xff
breq DATA_RECEIVED
STORE_INSTRUCT:
    ldi ZH, high(NEXT_INSTRUCTION)
    ldi ZL, low(NEXT_INSTRUCTION)
    lds r19, CURR_INS_BYTE
    add ZL, r19
    st Z, r18
    inc r19
    sts CURR_INS_BYTE, r19

DATA_RECEIVED:
;ALL BYTES READ, EXIT
    pop r16
    out SREG, r16
    pop r16
    pop r17
    pop r18
    pop r19
    pop ZL
    pop ZH
    reti

```