Specifikation och programmeringsguide för LedJoy

Detta dokument innehåller en specifikation av plattformen LedJoy som används för laboration 2 (Snake) i kursen Maskinnära programmering. I kapitel 1 beskrivs hårdvaran samt kopplingen mellan de specialbyggda komponenterna och Arduino-plattformen. Kapitel 2 innehåller programmeringstips för laborationen.

1. Översikt

Hårdvaran består av en tvåaxlig joystick och en LED-display (plus drivelektronik) kopplade till en Arduino Uno med en ATmega328p-processor. LedJoy är avsedd att hållas med joysticken **under** LED-displayen, det vill säga ungefär som en ursprunglig Game Boy.

1.1 LED-display

LED-displayen är ordnad som en matris med åtta rader och åtta kolumner. Varje rad och varje kolumn är anslutna till var sin utgång på processorn, det vill säga totalt 16 utgångar används för displayen. En given lysdiod tänds genom att strömsätta (aktivera) utgångarna för dess rad respektive kolumn. Om en kolumn och flera rader är aktiverade så lyser kolumnens dioder svagare och svagare för varje rad som aktiveras. Om en rad och flera kolumner är aktiverade så lyser radens dioder med samma intensitet oberoende av antalet aktiva kolumner. För att få en jämn intensitet på lysdioderna är det alltså bara möjligt att ha en rad tänd i taget.

Om programmet itererar genom alla rader med tillräckligt hög hastighet så kommer ögat att se det som att alla rader är tända samtidigt. Eftersom varje lysdiod bara är tänd 1/8 av tiden så är dock den upplevda ljusintensiteten lägre än en ständigt aktiv lysdiod.

1.2 Joystick

Joystickens axlar är anslutna till var sin analogingång på processorn. Analogingångarnas upplösning är 10 bitar, vilket innebär att en Joystick i neutralt läge ger ett värde på nära 512. Om joysticken dras så långt som möjligt uppåt eller åt vänster ger avläsningen av Y-respektive X-axeln värden nära 1023, medan nedåt och höger ger värden nära 0.

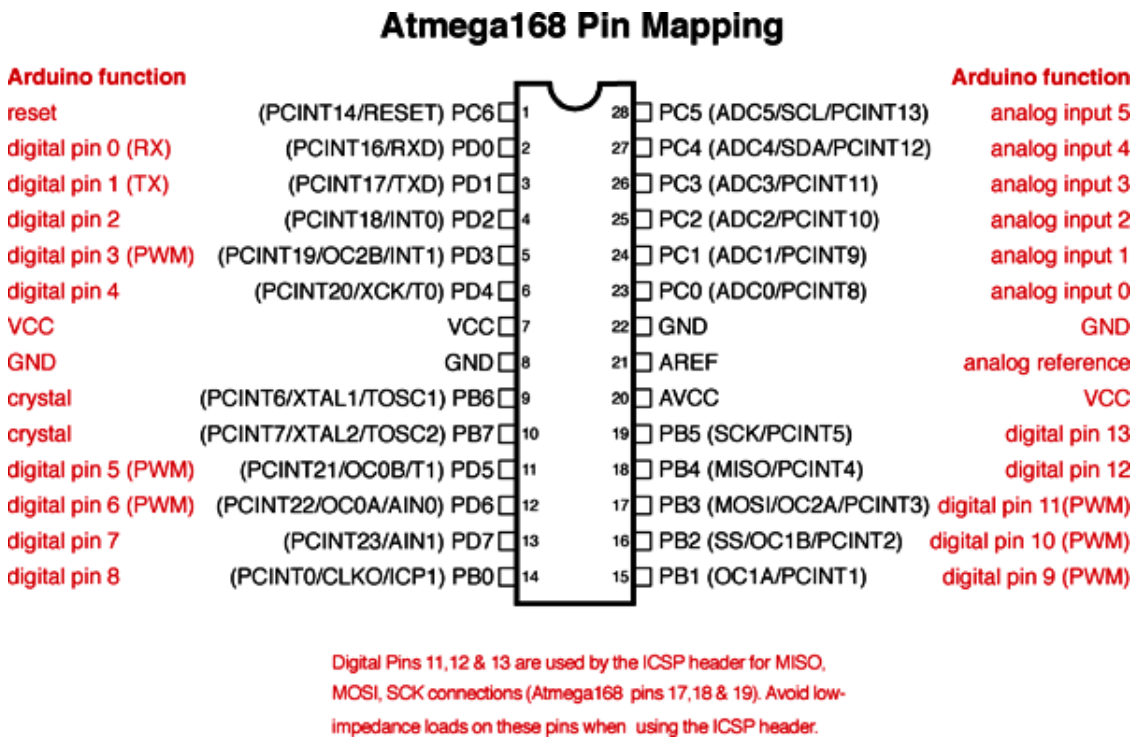
Det är möjligt att ställa om ATmegans A/D-omvandlare till ett 8-bitarsläge för att lättare läsa av de 8 mest signifikanta bitarna (i och med att ATmega har 8-bitarsregister är 8-bitarsvärden något lättare att hantera). Se avsnitt 2.4 för information om hur detta kan göras.

1.3 Koppling till utgångar

LED-displayen och joysticken är kopplade till ATMegas ut/ingångar enligt följande:

LedJoy	ATmega pin	Arduino-funktion
LED-display Rad 0	**PORTC0** (PC0)	Analog input 0 (används som digital)
LED-display Rad 1	**PORTC1** (PC1)	Analog input 1 (används som digital)
LED-display Rad 2	**PORTC2** (PC2)	Analog input 2 (används som digital)
LED-display Rad 3	**PORTC3** (PC3)	Analog input 3 (används som digital)
LED-display Rad 4	**PORTD2** (PD2)	Digital 2
LED-display Rad 5	**PORTD3** (PD3)	Digital 3
LED-display Rad 6	**PORTD4** (PD4)	Digital 4
LED-display Rad 7	**PORTD5** (PD5)	Digital 5
LED-display Kolumn 0	**PORTD6** (PD6)	Digital 6
LED-display Kolumn 1	**PORTD7** (PD7)	Digital 7
LED-display Kolumn 2	**PORTB0** (PB0)	Digital 8
LED-display Kolumn 3	**PORTB1** (PB1)	Digital 9
LED-display Kolumn 4	**PORTB2** (PB2)	Digital 10
LED-display Kolumn 5	**PORTB3** (PB3)	Digital 11
LED-display Kolumn 6	**PORTB4** (PB4)	Digital 12
LED-display Kolumn 7	**PORTB5** (PB5)	Digital 13
Joystick Y-axel	**PORTC4** (PC4)	Analog input 4
Joystick X-axel	**PORTC5** (PC5)	Analog input 5

Kopplingen mellan portnamnen och den faktiska hårdvaran framgår av följande bild (tagen från www.arduino.cc, gäller både ATmega 168 och ATmega 328p):



2. Programmeringstips för I/O

Vid implementationen av Snake-uppgiften behöver ett antal hårdvaruspecifika problem lösas. Detta kapitel beskriver de huvudsakliga problemen och ger tips på hur de kan angripas.

2.1 I/O-programmering

ATmega använder sig av minnesmappad I/O. Mappningen mellan portar och minnesaddresser, samt mappningen av bitar i portarna till specifika in- eller utgångar finns i referensmanualen för ATmega (se filsamlingen i scio under ATmega – material), kapitel 30 (Register Summary). Notera att AVR studio har inbygga alias för portadresserna (t ex PORTD istället för 0x2B). Alias finns i ”Name”-kolumnen i referensmanualens Register Summary-kapitel (även namnen på de enskilda bitarna kan användas som alias). Använd helst dessa för mer lättläst kod.

2.1.1 Indelning av I/O-rymden

I/O-rymden i ATmega består av två delar. Den *primära* I/O-rymden är adresserna 0x0020-0x005F. För I/O-operationer i denna delrymd kan instruktionerna `in` samt `out` användas. Notera dock att dessa instruktioner använder sig av adresserna 0x0000-0x003F. En offset på 0x0020 adderas alltså automatiskt till den adress som anges till `in` och `out`. Detta görs för att ”hoppa förbi” den del av I/O-rymden som refererar till de minnesmappade processorregistren. Genom att använda alias istället för explicita adresser behöver man inte tänka på detta.

Adresserna 0x0060-0x00FF utgör den *utökade* I/O-rymden. Dessa adresser mappar också till I/O-portar men kan inte adresseras av `in` och `out`. För att läsa eller skriva till denna minnesrymd måste istället instruktionerna `lds` och `sts` (`ldd` / `std`) användas. Ingen offset adderas när dessa instruktioner används utan den ”riktiga” minnesadressen skall anges.

2.1.2 Bitmanipulation

När enskilda bitar skall sättas i ett I/O-register är det viktigt att övriga bitar förblir oförändrade. Om till exempel rad 4 ska aktiveras i LED-displayen (bit 2 i PORTD sätts till 1) så får inte bit 6 och 7 förändras eftersom dessa är associerade med kolumner. De har satts till 1 eller 0 för att visa om dioderna på motsvarande position skall lysa när rad 4 aktiveras.

För de I/O-portar som ligger i den primära I/O-rymden kan instruktionerna `sbi` samt `cbi` användas för att sätta enskilda bitar till 1 respektive 0. Detta fungerar dock inte för den utökade I/O-rymden. Processen för att ändra en enskild bit i detta utrymme blir istället

1) Läs in föregående värde
2) Modifiera enskilda bitar genom maskning
3) Skriv det modifierade värdet

Till exempel kan följande kod användas för att sätta bit TOIE0 i register TIMSK0 till 1:

```
ldi  rTemp, (1<<TOIE0)
lds  rTemp2, TIMSK0
or   rTemp, rTemp2
sts  TIMSK0, rTemp
```

2.2 Initialisering av program och avbrottsvektorer

Först i kodsegmentet skall en tabell med alla avbrottsvektorer ligga. Man behöver egentligen bara tänka på att definiera vektorer för de avbrott som man ämnar använda. Vektorn som ligger först i kodsegmentet är reset-vektorn, vilken är den som körs när processorn startas eller startas om. Här bör ett hopp till den kod som programmet skall inledas med läggas. Sådan initialisering bör vara konfiguration av stackpekaren, initialisering av variabler, konfiguration av I/O-register etc.

Ett annat avbrott som är användbart för uppgiften är TIM0_OVF, som aktiveras varje gång Timer0 får overflow på sin räknare (se avsnitt 2.6). Vektorn för detta avbrott ska ligga på position 0x0020. En lämplig inledning av kodsegmentet är därför

```
.CSEG                    // Code segment
.ORG 0x0000
     jmp  init           // Reset vector
     nop
.ORG 0x0020
     jmp  isr_timerOF    // Timer 0 overflow vector
     nop

.ORG INT_VECTORS_SIZE
// … resten av koden
```

`isr_timerOF` är den subrutin som innehåller koden som skall köras i timer-avbrottet. INT_VECTORS_SIZE är definierad som storleken på tabellen med avbrottsvektorer.

2.2.1 Initialisering av stackpekaren

För att kunna använda `push` och `pop`-instruktionerna måste först stackpekaren initialiseras. `push` och `pop` är användbara för att till exempel temporärt spara undan innehållet i register om registret behöver användas till något annat, exempelvis vid subrutinanrop eller avbrottshantering. Stackpekaren implementeras som två minnesmappade register: SPH (Stack Pointer High) samt SPL (Stack Pointer Low). SPH skall sättas till den högre byten i den högsta tillgängliga minnesadressen och SPL till den lägre byten. Nedanstående kod kan användas till detta:

```
ldi  rTemp, HIGH(RAMEND)
out  SPH, rTemp
ldi  rTemp, LOW(RAMEND)
out  SPL, rTemp
```

`rTemp` är i exemplet ett `.DEF`-alias för något processorregister. Koden bör köras allra först i initialiseringen, innan några avbrott aktiveras.

2.3 Val av input/output

För att kunna göra läsning från eller skrivning till en I/O-port måste den tillhörande in-/utgången konfigureras för input eller output. För varje portregister (t ex PORTD, 0x2B) finns ett så kallat *Data Direction Register (DDR)*. Bitarna i det sistnämnda styr hur bitarna i det förstnämnda fungerar. En etta i DDR visar att motsvarande bit i det associerade portregistret är output, en nolla visar input. För att konfigurera bit 0 i PORTC (där rad 0 i LedJoy är inkopplad) som en output-bit måste alltså bit 0 i DDRC sättas till 1. Samtliga in- och utgångar där LED-displayen är inkopplad måste sättas till output.

I AVR studio finns inbyggda alias för DDR-registren. DDR-registret för PORTB heter DDRB, det för PORTC heter DDRC och det för PORTD heter DDRD.

2.4 Uppdatering av LED-displayen

Som beskrivet i avsnitt 1.1 måste raderna i displayen aktiveras en efter en. En lämplig metod för displayuppdatering är:

1. Avaktivera aktuell rad
2. Aktivera och avaktivera kolumner för nästa rad
3. Aktivera nästa rad

Ju oftare detta görs desto mindre flimmer blir det på displayen.

För att aktivera en rad eller kolumn skrivs en etta till motsvarande utgång (in/utgången måste vara konfigurerad som output, se avsnitt 2.1). När en rad aktiveras skall alltså först ettor skrivas till de utgångar som motsvarar de kolumner som ska ha tända dioder på den raden, sedan skall en etta skrivas till den utgång som motsvarar den rad som skall aktiveras. För att avaktivera en rad skrivs en nolla till den utgång som motsvarar den rad som skall avaktiveras.

2.5 A/D-omvandling för joystickavläsning

Joystickens axlar är kopplade till var sin analogingång på ATmegan (se tabellen i avsnitt 1.3). Det analoga värde som läses av på ingången måste omvandlas till ett digitalt värde via ATmegans inbyggda A/D-omvandlare (Analog to Digital). För att använda denna till joystick-avläsning behövs fyra minnesmappade register: ADMUX, ADCSRA, ADCH och ADCL.

Register	Användning
ADMUX	Konfigurationsregister. Bit 0 – 3 väljer analogingång. Bit 6 – 7 väljer källa för referensspänning. Bit 5 (ADLAR) väljer mellan 10-bitars och 8-bitarsläge.
ADCSRA	Kontrollregister. Bit 6 (ADSC) används för att påbörja omvandling samt för att avgöra om omvandlingen är klar. Bit 7 (ADEN) aktiverar A/D-omvandlaren. Bit 0-2 styr ”pre-scaling” som avgör klockhastigheten för omvandlaren.
ADCH	De övre 2 resultatbitarna efter A/D-omvandling
ADCL	De lägre 8 resultatbitarna efter A/D-omvandling

Innan A/D-omvandling kan göras måste A/D-omvandlaren konfigureras. Detta behöver bara göras *en* gång, till exempel i en initialiseringsrutin. Det som skall göras är:

1. Sätt bit 6 (REFS0) i ADMUX till 1 och bit 7 (REFS1) till 0.
2. Sätt bit 0 – 2 (ADPS0, ADPS1, ADPS2) samt bit 7 (ADEN) i ADSCRA till 1.

Processen för att göra en enkel A/D-omvandling är som följer:

1. Välj källa (analogingång) genom att sätta de fyra lägsta bitarna i ADMUX till rätt värden. Dessa tolkas som ett fyrabitars binärt tal, så för att till exempel välja analogingång 4 sätts bitarna till 0100.
2. Starta konverteringen genom att sätta ADSC-biten i ADCSRA-registret till 1.
3. Iterera tills ADSC-biten i ADCSRA-registret är 0 (busy-wait är OK). Detta signalerar att omvandlingen är klar. Instruktionen `sbrc` är användbar för detta.
4. Kopiera resultatet från ADCL samt ADCH. **Notera att ADCL måste läsas först, sedan ADCH.** A/D-omvandlaren anser en omvandling vara klar när ADCH-registret är läst.

Detta görs alltså varje gång en A/D-omvandling görs. Tänk på att övriga bitar i ADMUX- samt ADCSRA-registren inte får förändras när skrivning görs i steg 1 och 2. Det är alltså nödvändigt att läsa in registervärdet, modifiera bitar via maskning och sedan skriva tillbaka värdet (skuggregister behöver inte användas).

2.5.1 8-bitarsläget

Det kan vara lite svårt att hantera 10-bitarsvärden eftersom ATmegans register bara är 8 bitar stora. Om 10-bitarsprecision inte behövs kan A/D-omvandlaren konfigureras till att generera 8-bitarsvärden. De två lägsta resultatbitarna från A/D-omvandlingen ignoreras då och de övre åtta bitarna placeras i ADCH. För att ställa in A/D-omvandlaren i 8-bitarsläge skall bit 5 (ADLAR) i ADMUX-registret sättas till 1 (tänk på att inte ändra övriga bitar).

2.6 Slumptalsgenerering

Ett problem med att jobba direkt mot hårdvaran är att det inte finns något standardbibliotek med vanliga funktioner. En sådan funktion som kan vara överraskande krånglig att implementera är en slumptalsgenerator –det finns ju inga slumptalstabeller att

tillgå. För att generera ett slumptal behövs alltså en källa till pseudoslump. En sådan källa kan vara bruset på analogingångarna.

En metod för att generera tal som ger intrycket av att vara slumpmässiga är att vid varje avläsning av analogingång 4 eller 5 (bedöm vilken som verkar fluktuera mest) addera resultatet till en variabel/ett register som tillåts ”slå över” om värdet övergår kapaciteten. När ett koordinatpar skall genereras kan en avläsning av x- respektive y-axelvärdet på joysticken göras. Om dessa körs genom en icke-linjär funktion (till exempel kan värdet upphöjas till 5) och adderas till slumpvariabeln/registret borde resultatet bli två någorlunda slumpmässiga värden. Använd de tre minst signifikanta bitarna i varje värde för att få ett koordinatpar som är tillräckligt slumpmässigt för spelet.

2.7 Timers

För att göra en jämn uppdatering av spelet krävs en klocka som räknar upp oberoende av programkoden. En sådan klocka kan implementeras via de *timers* som finns inbyggda i ATmega-processorn. Den timer som är enklast att använda är Timer0, en 8-bitarstimer som kan ställas in att räkna upp från 0 till 255. När timern har nått 255 slår den över till 0 igen och ett avbrott genereras. Detta avbrott kan användas för att räkna upp en klocka som i sin tur kan användas för att avgöra när nästa uppdatering av spelet ska göras.

Två register används för att kontrollera Timer0. TCCR0B-registret används för att ställa in ”pre-scaling”, dvs hur ofta timern skall räknas upp. Bit 0-2 i TCCR0B konfigureras på följande vis:

CS02 (bit 2)	CS01 (bit 1)	CS00 (bit 0)	Effekt
0	0	0	Timern stannas
0	0	1	Klocka
0	1	0	Klocka / 8
0	1	1	Klocka / 64
1	0	0	Klocka / 256
1	0	1	Klocka / 1024

Genom att sätta bitarna till 101 ökas alltså timern med 1 för var 1024:e klockcykel. Detta är en lämplig konfiguration för Snake-uppgiften.

Det andra registret som används för Timer0 är TIMSK0. Här används bit 0 (TOIE0) för att aktivera ett avbrott som genereras varje gång timern ”slår över” från 255 till 0. Detta avbrott (TIM0_OVF) gör att programmet hoppar till avbrottsvektor 0x0020. På plats 0x0020 i kodsegmentet skall alltså ett hopp till den rutin som hanterar timern placeras.

För att starta timern krävs sammanfattningsvis tre steg:

1. Konfigurera pre-scaling genom att sätta bit 0-2 i TCCR0B
2. Aktivera globala avbrott genom instruktionen `sei`
3. Aktivera overflow-avbrottet för Timer0 genom att sätta bit 0 i TIMSK0 till 1.