Arduinokurs

Del 1 - Vett och etikett

Talsystem

Binärt – ettor och nollor

Det binära (Bin) talsystemet är grundläggande för alla datorer och logiska system. Ett värde kan endast vara sant eller falskt – ett eller noll! Det är därför viktigt att behärska det binära och även det hexadecimala talsystemet. 8-bitar datorns register är uppbyggt med just 8 bilar. De åtta bitarna 10101010 är lika med talet 170 (decimalt). Det decimala talsystemet är det vi använder dagligen.
Principen är samma som för det decimala talsystemet med skillnaden att basen är 2 istället för 10 som vi normalt använder. 2 därför att vi endast har två siffror.
T.ex. 0101 Bin = $0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 0 + 4 + 0 + 1 = 5$ Dec.

Hexadecimalt 0 - F

Det hexadecimala (Hex) talsystemet är till skillnad från det decimala uppbyggt av talen 0 – 9 samt A, B, C, D, E & F. Man kan alltså med ett tecken representera talen 0 – 15 Dec.

Decimalt – binärt och hexadecimalt

Dec	Bin				Hex
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	A
11	1	0	1	1	B
12	1	1	0	0	C
13	1	1	0	1	D
14	1	1	1	0	E
15	1	1	1	1	F

Lär dig tabellen utantill! (eller i alla fall hur du tar fram den)

Ovan nämndes åtta bitar men mittenkolumnen innehåller ju bara 4 bitar. Om vi dubblerar fås åtta bitar. Här kommer finurligheten med det hexadecimala talsystemet. Vi kan ta två ex. med 8-bitar.
Talet 7 Dec skrivs 00000111 Bin och 07 Hex
Talet 58 Dec skrivs 00111010 Bin och 3A Hex

ASCII-tabellen

Bra att känna till är även ASCII-tabellen. Den är grunden för alla skrivbara tecken samt även styrtecken som t.ex vagnretur (Carrige Return, CR)
Varje tecken & kommando i tabellen har sin egen kod och det är den koden son vi använder internt i koden eller det som t.ex. skickas via serieporten nät text och siffror skickas till en terminal.

Ett typiskt användningsområde är om en räknares närde ska skrivas ut på en display. Räknaren kan räkna från 0 – 9. Skulle vi skriva räknarens värde direkt till displayen skulle utskriften bli styrtecknen ’00 – 09’. Det fungerar inte. Om vi däremot adderar 30 Hex och sedan skriver ut får siffrorna 0 – 9. ASCII-koden för 0 är nämligen 30 Hex.
Det här kan vara lite förvirrande men det finns funktioner som tar hand om detta.

Codepage 819 - Latin 1 - ISO 8859-1

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	007F
8-	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F
9-	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
A-	00A0	¡ 00A1	¢ 00A2	£ 00A3	¤ 00A4	¥ 00A5	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	ª 00AA	« 00AB	¬ 00AC	- 00AD	® 00AE	¯ 00AF
B-	° 00B0	± 00B1	2 00B2	3 00B3	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	1 00B9	º 00BA	» 00BB	¼ 00BC	½ 00BD	¾ 00BE	¿ 00BF
C-	À 00C0	Á 00C1	Â 00C2	Ã 00C3	Ä 00C4	Å 00C5	Æ 00C6	Ç 00C7	È 00C8	É 00C9	Ê 00CA	Ë 00CB	Ì 00CC	Í 00CD	Î 00CE	Ï 00CF
D-	Ð 00D0	Ñ 00D1	Ò 00D2	Ó 00D3	Ô 00D4	Õ 00D5	Ö 00D6	× 00D7	Ø 00D8	Ù 00D9	Ú 00DA	Û 00DB	Ü 00DC	Ý 00DD	Þ 00DE	ß 00DF
E-	à 00E0	á 00E1	â 00E2	ã 00E3	ä 00E4	å 00E5	æ 00E6	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ì 00EC	í 00ED	î 00EE	ï 00EF
F-	ð 00F0	ñ 00F1	ò 00F2	ó 00F3	ô 00F4	õ 00F5	ö 00F6	÷ 00F7	ø 00F8	ù 00F9	ú 00FA	û 00FB	ü 00FC	ý 00FD	þ 00FE	ÿ 00FF

Logik

Mycket programmering bygger på logiska samband och det är därför viktigt att kunna grunderna i logik, även kallad boolesk algebra.
Logiska funktioner brukar presenteras i sanningstabeller. De grundläggande är OCH, ELLER och ICKE. AND, OR och NOT

AND		f
0	0	0
0	1	0
1	0	0
1	1	1

OR		f
0	0	0
0	1	1
1	0	1
1	1	1

NOT	f
0	1
1	0

NAND		f
0	0	1
0	1	1
1	0	1
1	1	0

NOR		f
0	0	1
0	1	0
1	0	0
1	1	0

XOR		f
0	0	0
0	1	1
1	0	1
1	1	0

Bitvis och logiska operationer.
Det är stor skillnad på bitvisa och logiska operationer. De bitvisa opererar på bitnivå medan de logiska arbetar på ’hela’ uttrycket.

I C skrivs operanderna för

Bitvis
AND - &
OR - |
NOT - !
XOR - ^

Logiskt
AND - &&
OR - ||

Ex.
Bitvis. C = A | B. Används t. ex. för att ’sätta’ bit 3

A = 0101 (5)
B = 1000 (8)
C = 1101 (13)

Logiskt blir C = A || B = 1 eftersom alla tal utom 0 tolkas som sanna. Mer om det här längre fram.

Logiska signalnivåer

Den logiska nivån för ’0’ är – tro det eller ej – 0V och logisk nivå för ’1’ är 5V (eller 3,3V). Det är mycket viktigt att dessa nivåer följs och att man inte blandar 3,3V och 5V logik hur som helst.
Logisk nolla vid 3,3V och 5V är ca 0 – 0,8V
Logisk etta vid 3,3V är ca 2V – 3,3V och vid 5V ca 2,5V – 5V

Vid minsta tvekan kontrolleras kretsens datablad där även annat av intresse kan hittas.

Del 2 - Grunderna i programmering

Programmeringsspråket C är ett generellt högnivåspråk som är oberoende av plattform. Det finns även många andra språk som i grunden har samma syntax. (Php, Java, Visual Basic m.m.)

Sätta bitar

Ofta behöver information om tillstånd sparas under programkörningen. Det finns bra och dåliga sätt att utföra den saken på.
Vi tar en hiss som exempel.

Vi behöver veta följande.
Har någon tryckt på knappen för att hissen ska komma?
Är dörren öppen eller stängd?
Är hissen på våning 1?

Tre frågor betyder att vi tar tre variabler.
Deklarering: int Button_Pressed = 0, Open = 0, Level = 0;

I minnet tas nu 3 x 2 = 6 byte upp.

Ett bättre alternativ är att utnyttja bitarna i en byte.

```
Deklarering: char Status = 0;
             //Status bits
             #define Button_pressed 1
             #define Open 2
             #define Level 4
```

För att sätta eller resetta bitarna används maskar samt AND och OR.
En bit ettställs med:
Status |= Button_pressed;
Och nollställs med
Status &= ~Button_pressed;

I minnet tas endast en byte upp dessutom fås en fördel på köpet. Man kan nollställa samtliga statusbitar med ett kommando. Status = 0;

5 byte skillnad låter inte mycket men om en liten controller används gäller det att hushålla med minnet.

Här är ett alternativt sätt att utföra samma sak på.

```
//def of Status
typedef union
{
  struct
  {
    unsigned Button_pressd :1; //1 if Button pressed
    unsigned Open          :1; //1 if open
    unsigned Level         :1; //1 if on level 1
  };
```

```
  unsigned _ALL_Status;
}Status;

Deklarering
Status Statusbits;

Statusbits.Button_pressed = 0;

Nollställa alla flagger
Statusbits._ALL_Status = 0;
```

Följande grundläggande funktioner är bra att känna till

if()
if() else
while()
for()

Funktionsanrop

Funktioner används ofta för att förenkla koden genom att ge funktionen en speciell uppgift och för att spara på minnet. En annan stor fördel är att man endast behöver ändra på ett ställe i koden om den ersätter kod på många ställen. Funktionen kan returnera och ta emot parametrar men det är inget krav.

Som exempel kan vi ta ett program där två tal behöver summeras på flera ställen. Istället för att skriva C = A + B på flera ställen anropas funktionen summa på följande sätt.
C = summa(A, B);
När kompilatorn kommer till raden förflyttas exekveringen till funktionen som finns på annan plats i minnet. Instruktionerna utförs och svaret returneras. Exekveringen fortsätter sedan på raden efter funktionsanropet.
Funktionen byggs upp på följande sätt

```
// summa
// summerar heltalen A & B
// Indata A, B
// returdata heltalsumman
int summa(int A, int B)
{
  int C;
  C = A + B;
  return C;
}
```

Funktionen behöver inte returnera eller ta emot data om t.ex. en port D0 ska nollas kan funktionen anropas med
Clear_port_D0();

Funktionen i sig

```
void Clear_port_D()
{
  digitalWrite(1,0);
}
```

Arrayer & pekare

Pekare anses i allmänhet som något svårt och det gäller att hålla ordning på begreppen. Enklast kan en pekare jämföras med en array, men skillnaden att syntaxen skiljer. Pekare används vid relativ adressering vilket betyder att man utgår från en basadress & x-antal byte framåt från denna.

En array kan ses som en tabell med fast storlek och som sparar variabler eller data. Det kan t.ex var temperaturer. Fördelen är att datan får ett index som man kan peka ut datan med. T.ex. värdet på variabeln i kolumn 3.
En array för heltal initieras på följande vis:

```
datatyp namn[storlek];

int tal_array[3];  //skapa en array med 3 kolumner
tal_array[0] = 6;
tal_array[1] = 7;
tal_array[2] = 8;
```

kolumn	1	2	3
index	0	1	2
data	6	7	8

För att t.ex. summera tabellen kan man skriva:

```
int summa, n;

for(n = 0; n < 3;n++)
{
  summa += tal_array[n];
}
```

Samma sak som ovan kan göras med pekare. En pekare är en variabel som innehåller adressen till ett värde. All data i minnet har en unik adress.
Om vi deklarerar
int b = 0;
int a = 5;

sparas ’a’ på en viss adress som bestäms av kompilatorn, t.ex. 0x0010

minne

Adress	Data
0x0010	5
0x0011	6
0x0012	

Samma sak som ovan kan göras mha. pekare.

```
int *ptr;  //ptr är en pekare till int
ptr = &a;  //& betyder adressen av, alltså tilldelas pekaren ptr adressen av a som i det här fallet är 0x0010
```

Vill vi läsa in nästa tal i tabellen ökas pekaren.

*ptr++;
b = *ptr; //tilldelar b talet 6 som finns på adressen 0x0011.

I den här kursen kommer pekare att användas när vi ska omvandla ettor och nollor till CW.

Interrupt

Interrupt är något av det kraftfullaste som finns i kontrollern. Enkelt förklarat betyder det att viss programexekvering sker automatiskt utan att huvudkoden hela tiden behöver kontrollera att t.ex. en räknare eller port har ändrats. Det finns både interna och externa interrupt.

Interrupt är något som kan ske när som helst under programexekveringen. Det som då händer är att pågående program avbryts. Exekveringen flyttas till interrupt rutinen (ISR). Koden i ISR exekveras och slutligen återupptas den 'vanliga' programkörningen.
Det är viktigt att ISR exekveras relativt snabbt så att tidsavbrottet inte blir märkbart i huvudprogrammet. Det är mao. Små funktioner som ska utföras, t.ex. öka en räknare. Det tar endast ett par klockcykler i tid och kan anses försumbart.
Är ett visst kodavsnitt väldigt tidskritiskt kan interrupt stängas av under exekveringen för att sedan åtet tillåtas när den kritiska koden är utförd.

ISR kommer att användas i CW-exemplet.

Modulo

Modulo är resten vid heltalsdivision. Kan även användas för att ta fram bråkdelar, t.ex. fjärdedelar. Tänk dig att du ska markera 'kvartarna' på en klocka mha. en räknare, n, som räknar från 0 - 60.
60 / 4 = 15. Man kan naturligtvis skriva 4 if-satser och villkora dessa mot kvartarna men det är här modulo kommer in.
1 mod 15 = 1, 2 mod 15 = 2,… 15 mod 15 = 0, 16 mod 15 = 1.
Villkoret blir betydligt enklare med modulo
If(!(n%15)) {'markera';}

Del 3 – Hårdvaran

Vad händer i mikrokontrollern?

Vid uppstart sker en hel del i mikrokontrollern.
Typ av oscillator väljs och om den interna används sätts även klockfrekvensen. Ocillatorn bestämmer hur snabbt instruktionerna exekveras. En instruktion exekveras per klockcykel och Arduinon arbetar på 16MHz vilket betyder att en instruktion tar 62,5ns att utföra.
Att räkna från 0 – 100 tar ca 20 us.
Portar initieras som in eller ut. In-portar kan även sättas som analoga ingångar.
Ränkare, uart, och andra pereferienheter initieras och som exempel på hur det sker tas PORTB. PORTB har 8 i/o och portpinnarna har många olika funktioner. Vissa funktioner är som standard aktiverade vi reset men generellt kan man säga att alla portar är ställda som ingångar.
Om porten är en in eller utgång bestäms av värdet som finns i registret DDRX, där X är porten. I det här fallet B. För att sätta PORTB5 till en utgång måste en ’1’ skrivas till bit 5 i DDRB, alltså 00100000 skrivs till registret som ligger på adressen 0x25. För att sedan bestämma om porten ska vara ett eller noll måste data skrivas till själva portpinnen. För att ettställa PORTB5 skrivs en ’1’ till PORTB bit 5, alltså 00100000.

13.4 Register Description

13.4.1 MCUCR – MCU Control Register

Bit	7	6	5	4	3	2	1	0	
0x35 (0x55)	–	BODS	BODSE	PUD	–	–	IVSEL	IVCE	MCUCR
Read/Write	R	R	R	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 4 – PUD: Pull-up Disable**

When this bit is written to one, the pull-ups in the I/O ports are disabled even if the DDxn ar PORTxn Registers are configured to enable the pull-ups ({DDxn, PORTxn} = 0b01). See ”Co figuring the Pin” on page 76 for more details about this feature.

13.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

13.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

13.4.4 PINB – The Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

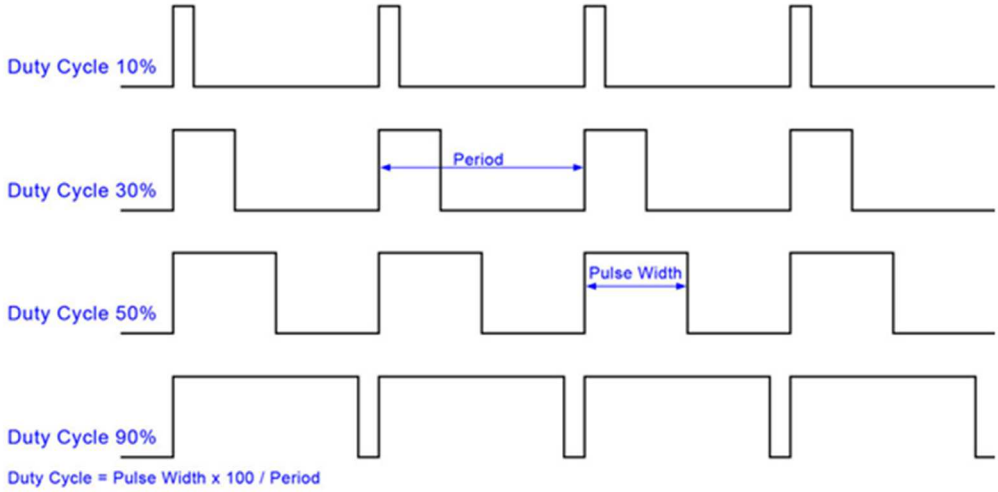
Pulsbreddsmodulering (PWM) – H-bryggan och motorstyrning

PWM är det digitala sättet att skapa en variabel effektmatning.
Principen är enkel. En fyrkantsignal med en hög frekvens är grunden. Frekvensen väljs oftast så att den ligger ovanför hörbart område för att slippa tjut i t.ex. en motor. Tjutet kommer sig av att motorn magnetiseras och vibrerar i takt med den inmatade signalen.
Grundfrekvensen är som sagt konstant men 'till-tiden', då signalen är hög ändras från 0 – 100 % vilket i sin tur gör att medelvärdet på utspänningen hamnar på samma nivå.
Matematiskt ser det ut som följer:

Vi väljer grundfrekvensen 1kHz. Det ger en periodtid (T) på 1ms. Matningen är 5V.
Utspänningen = Matningsspänningen x pulsbredden / periodtiden.

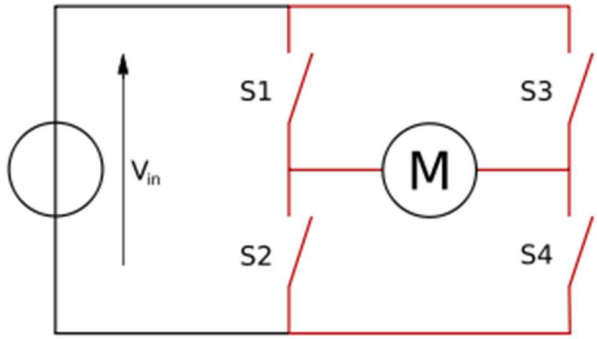
I bilden nedan har vi en pulsbredd på 10% vilket ger 0,1ms puls
Utspänning = 5V x 0,1ms / 1ms = 0,5V

Tar vi 50% får vi på samma sätt 2,5V vilket är lite intressant. Mäter man på en digital klocka med en multimeter fås värdet 2,5V. Nu vet du varför.



En motor kan inte kopplas direkt till en utgång på mikrokontrollern. Någon form av effektsteg behövs. Enklaste lösningen är en H-brygga pga effektivitet och pris.

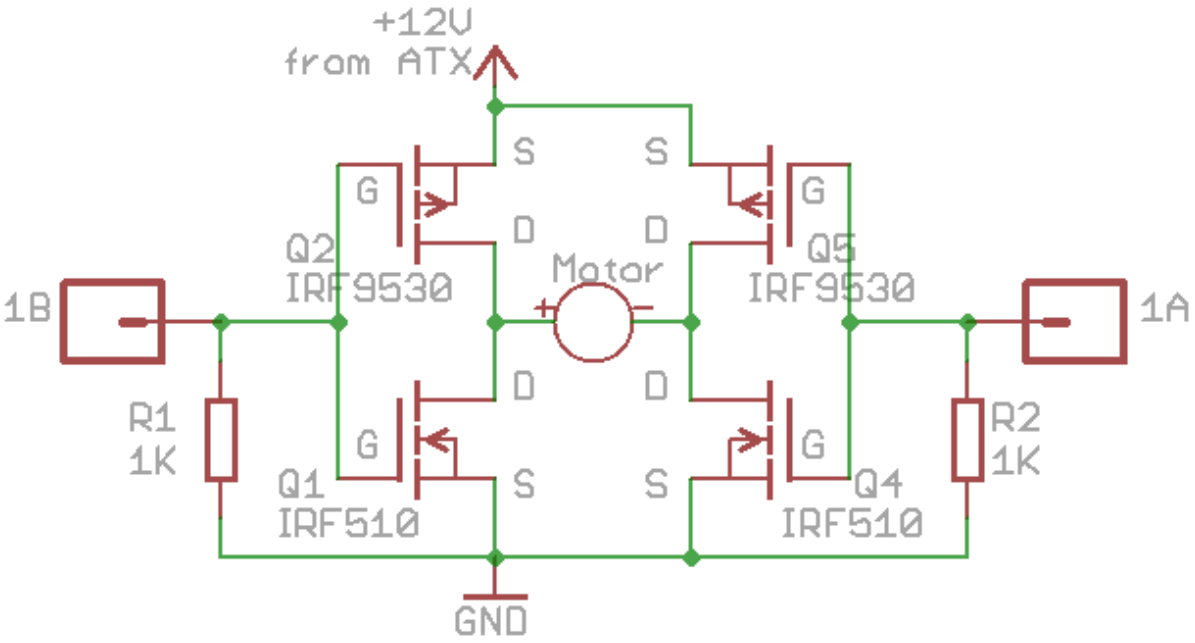
Funktion:
S1 – S4 är transistorer av något slag. Både MOSFET och bipolära förekommer.
Om ingen av S1 – S4 är till står motorn naturligtvis still. Slås S1 och S4 till läggs full maning på och motorn snurrar framåt. Sluts istället S2 och S3 fås omvänd polaritet och motorn snurrar bakåt. Motorn kan alltså köras både framåt och bakåt.



Vi tar nu o kikar på hur det ser ut i verkligheten.
Läggs en ’nolla’ på ingång 1B och en ’etta’ på ingång 1A fås plus och minus enligt bilden.
Gör vi tvärt om fås omvänd polaritet.

Det finns två sätt att koppla in PWM-signalen.

1. PWM på 1B och sedan väljs 1A hög eller låg beroende på vilket håll motorn ska snurra åt. Två portpinnar behövs.
2. En inverterare kopplas mellan 1B och 1A. PWM-signalen kopplas till 1B. En portpinne behövs



Fördelen med alternativ 2 är att fram/back sköts med endast PWM-signalen, styrningen blir dock lite annorlunda.
Full fart framåt fås vis 100% dutu cycle
Stopp fås vid 50% duty cycle
Full back fås vid 0% duty cycle.