Subrutiner generelt

Når en subrutine bliver kaldt vil den næste instruktion adressen blive gemt i stakken. Der efter bliver den placering som den kaldte subrutine ligger på lagt i program counter. Dette vil forårsage at returnering adressen er gemt i stakken og subrutine vil blive udført. Når subrutine er blevet ud ført bliver adressen som blev gemt i stakken lagt i program counter så mikrocontrolleren forsætter i programmet fra efter kaldet.

Subrutinerne er så vidt muligt opbygget så den værdi som skal behandles eller benyttes i subrutine tages fra registret Arg. Hvis der kræves flere parameter til subrutine bliver registrene Temp1 og Temp2 også brugt. Hvis subrutine skal returnere en værdi til vider brug i programmet ligges denne i registret Ret1.

SendSpeed

SendSpeed subrutinen sender et telegram med typen REPLY, kommandoen Start og den aktuelle værdi i OCR2 over seriel kommunikation.

SendSpeed kræver ingen parameter og returnerer ingen værdi.

Det første SendSpeed subrutinen gør er at den lægger den værdi af besked typen REPLY ind i registret Arg hvorefter Send subrutinen kaldes. Denne subrutine sender værdien i Arg over seriel kommunikation. Der efter lægges den værdien for kommandoen Start ind i Arg og den sendes på samme måde. Indholdet i OCR2 bufferen hentes ind i Arg hvorefter den sendes som resten af telegrammet ved brug af Send subrutinen. Se side 565 i \cite{AVR}.

Ref:

AVR side 565

Send

Send subrutinen sender det aktuelle indhold i Arg registret via seriel kommunikation.

Send kræver en parameter, den værdi som skal sendes. Der returneres værdien i Arg over seriel kommunikation.

Send subrutinen starter med at tjekke om UDRE i UCSR er sat hvilket fortæller at ”sende bufferen” TXB er klar til at modtage den nye besked. Se side 407 i \cite{AVR}. UCSR er kontrol register for den seriel kommunikation. Se side 408 i \cite{AVR}. Hvis UDRE ikke er sat vil Send subrutinen vente i et loop ind til UDRE er sat.

Når TXB er klar ligges Arg ud i UDR hvilket gør at beskeden i Arg ligge i TXB hvor den vil blive sendt over seriel kommunikation.

Ref:

AVR side 407/408

SendPrescaler

SendPrescaler subrutinen bruges til at sende den prescaler der aktuelt bruges på timer2 til PWM for motoren.

SendPrescaler kræver ingen parameter og returnerer den aktuelle prescaler over seriel kommunikation.

Til at starte med indhentes TCRR2 hvilket er timer 2 kontrol registret ind i Temp1. Se side 567 i \cite{AVR}. Bit 7 til 3 ud-maskes da det udelukkende er bit 2 til 0 der kontrollerer prescaleren. Derefter lægges værdien for et REPLY ind i registret Arg og sendes med subrutinen Send over seriel kommunikation. Værdien for kommandoen PWM Prescaler lægges i Arg og derefter sendes med Send. Temp1 bliver derefter kopieret over i Arg for derefter at blive sendt som de andre.

Ref

ACR side 567

CalcOCR2

CalcOCR2 subrutinen beregner ud fra en værdi på 0 til og med 100 svarende til hvor mange procent af den totale hastighed motoren skal køre med. Hvilken værdi der skal ligges i OCR2 hvilket kontrollerer PWM til motoren.

CalcOCR2 kræver en parameter fra 0 til og med 100 i Arg svarene til den ønskede hastighed og returnerer værdien til OCR2 i Ret1.

I bogen ”The AVR Microcontroller and Embedded Systems: Usnig Assenmbly and C” gives ligningen for at udregne duty cycle for phase korrekt PWM i AVR. Se side 569 i \cite{AVR}. Se ligning \ref{eq:CaOCR21} (1 CalcOCR2) Da OCR2 og OCR0 begge er 8 bit register er ligningerne ens for dem.

\begin{equation}

Duty Cycle = \frac{OCR0}{255}\cdot 100 (1 CalcOCR2)

\label{eq:CaOCR21}

\end{equation}

Hvis OCR0 som fremover nævnes som OCR2 isoleres fås ligningen \ref{eq:CaOCR22} (2 CalcOCR2).

\begin{equation}

OCR2 = Duty Cycle \cdot 2.55 (2 CalcOCR2)

\label{eq:CaOCR22}

\end{equation}

For at multiplicere med et decimaltal tal kan fixed-point beregninger benyttes. Hvis 2.55 laves om til Q-Format Q2.6 kan man udføre multiplikation med decimaltal i mikrocontrolleren. Format Q2.6 er det format på fixed-point som med højst præcision udtrykker 2.55 på 8 bit.

Værdien som skal udregnes vil være på format Q7 da en værdi der ligger fra 0 til og med 100 kan udtrykkes med 7 bit. Dermed vil en multiplikatioen resutat være i format Q9.6. Se side 44 i \cite{FixP}. Da en multiplikation mellem 2.55 og en værdi fra 0 til og med 100 maksimalt kan give 255 hvilket kan udtrykkes med 8 bit vil den 9’ne bit i resultatet være 0 og dermed ligegyldig. Resultatet vil dermed kunne udtrykkes med format Q8.6.

CalcOCR2 implementere dette ved at gemme 2.55 som Q2.6 format i Temp1 og derefter udføre en multiplikation af Arg og Temp1. Efter multiplikationen ligger resultatet høje Byte i R1 og lave Byte i R0. Da resultatet som nævnt er i format Q8.6 shiftes R1 til højre ind i carry og R0 shiftes mod højre med carry. Begge shift gøres 6 gange for at have resultatet i format Q8 i R0. Til sidst kopieres R0 ind i Ret1 og CalcOCR2 afsluttes.

Ref

AVR side 569

Fixed-Point Signal Processing side 44

SetLED

SetLED subrutinen tager en værdi på fra 0 til og med 63 og lægger denne ud på de udgange som RGB LED’erne er forbundet til.

SetLED kræver en parameter fra 0 til og med 63 i Arg og returner ingen værdi.

SetLED tjekker først LEDTimON flaget i SREG2 for at sikre at der ikke er en timet LED i gang. Hvis der er dette afsluttes SetLED. Hvis der ikke er dette Tjekkes Arg for at være fra 0 til og med 63. Dette gøres da de påsatte RGB LED’er har 6 udgange og der med kan udtrykkes med 6 bit hvilket giver 64 muligheder inklusiv 0. Hvis den ikke er dette afsluttes SetLED. Hvis værdien er inden for det gyldige område shiftes Arg 1 bit til venstre. Nu hentes værdien på PORTA ind i Temp1 hvorefter Temp1’s bit 6 til 1 ud-maskes således at den gamle LED værdi ikke påvirker den nye. Arg indsættes i Temp1’s bit 6 til 1 hvorefter Temp1 sendes til PORTA. Der efter afsluttes SetLED. Denne metode sikrer også at ADC konverteringen på indgang 7 på PORTA ikke påvirkes da dette bit ikke ændres.

SaveInEEPROM

SaveInEEPROM subrutinen gemmer værdien i Arg på en adresse angivet med høje 2 bit i Temp1 og lave Byte i Temp2

SaveInEEPROM kræver værdien der skal gemmes i Arg og adressen i Temp1:Temp2

Til at starte med tjekkes om EEPROM er klar til at modtage en ny handling. Dette gøres ved at tjekke om EEWE flaget i EECR registret ikke er sat. EECR er kontrol registret for EEPROM og EECR flaget et sat mens EEPROM er optaget. Hvis flaget er sat kører SaveInEEPROM i et loop hvor den venter på flaget ikke er sat.

Hvis EECR flaget ikke er sat og EEPROM dermed er klar ligges Temp1 ind i EEARH og Temp2 ind i EEARL. Dette er henholdsvis registret til den høje 2 bit og lave Byte af adressen man vil operer med i EEPROM kaldet EEPROM adresse registret. Herefter lægges Arg ind i EEDR, hvilket er EEPROM data registret. Dette register er en form for mellem station mellem EEPROM og CPU’en. Som det næste stoppes for globale interrupt. EEMWE og EEWE i EECR sættes højt i den nævnte rækkefølge. Derefter sættes globale interrupt til igen. Grunden til den globale interrupt er slukket mens EEMWE og EEWE sættes, skyldes at der maksimalt må gå fire clock cycles fra EEMWE er sat til EEWE skal være sat. Se side 234 i \cite{AVR}. Dette sikrer at microntrolleren sætter de to flag højt lige efter hinanden. Efter dette afsluttes SaveInEEPROM.

Ref:

AVR side 234

LoadFromEEPROM

LoadFromEEPROM subrutinen henter værdien på adressen angivet med høje 2 bit i Temp1 og lave Byte i Temp2 ind i Ret1

LoadFromEEPROM kræver adressen i Temp1:Temp2 og returnerer værdien fra adressen i Ret1

Til at starte med tjekkes om EEPROM er klar til at modtage en ny handling. Dette gøres på samme måde som i SaveInEEPROM. Hvis EEPROM er klar lægges Temp1 og Temp2 ind i EEPROM adresse registret. Herefter sættes EERE i EECR højt da dette vil starte en ”read opreation” på den angivne adresse i EEPROM adresse registret. Se side 234 i \cite{AVR}. Værdien på adresse hentes fra EEDR ind i Ret1. Derefter afsluttes LoadFromEEPROM.

Ref:

AVR side 234

IsType

IsType subrutinen tjekker indholdet af InBesked for at være en Type. Hvis den er dette, gemmes InBesked i data space.

IsType bruger InBesked som input og gemmer typen i data space.

IsType tjekker InBesked for at være de tre typer Types der er gyldig i protokollen. Hvis InBesked ikke er en Type afsluttes IsType. Hvis InBeseked er en type gemmes InBesked i Data Space på adressen under navnet \verb!InType\_DatSpac!.

IsCmd

IsCmd subrutinen tjekker indholdet af InBesked for at være en kommando. Hvid den er dette gemmes InBesked i data space

IsCmd bruger InBesked som input og gemmer kommandoen i data space

IsCmd tjekker InBesked for at være en gyldig kommando. Hvis InBesked er dette gemmes den i Data Space på adressen under navnet \verb!InCmd\_DatSpac!. Hvis InBesked ikke er en kommando afsluttes IsCmd.

---------------------------------------Dette er blevet opdateret-------------------------------------------------------------------

SetSpeed

SetSpeed subrutinen tager den aktuelle værdi fra 0 til og med 100 i registret Arg og sætter bilens hastighed til aktuelle værdi i procent af bilens max hastighed.

SetSpeed kræver en parameter, den ønskede hastighed fra 0 til og med 100 i ligges i registret InBesked. SetSpeed returnerer ingen værdi.

Det første der sker i SetSpeed er at InBesked bliver tjekket for at være 100, hvis den ikke er dette vil InBesked blive kopieret til Arg og subrutinen CalcOCR2 bliver kaldt. CalcOCR2 returnerer den værdi der skal lægges i timer 2’s OCR register kaldet OCR2. Den returnerede værdi i Ret1 lægges i OCR2 bufferen hvor den vil blive lagt i OCR2 registret når timer 2’s optællings register TCNT2 kommer op på 0xFF hvilket er ”toppen” af registret. Se side 565 i \cite{AVR}. Dermed ændres den PWM som påvirker motoren på bilen. Hvis InBesked er 100 bliver OCR2 sat til 255.

Ref:

AVR side 565

PulseLED

PulseLED subrutinen tager en værdi på fra 0 til og med 63 og lægger denne ud på de udgange som RGB LED’erne er forbundet hvor værdien vil forblive i et sekund.

PulseLED kræver en parameter fra 0 til og med 63 i Arg og returner ingen værdi.

PulseLED starter med at tjekkes værdien i Arg for at være fra 0 til 63 af samme grund som i SetLED. Hvis værdien ikke er dette afsluttes PulseLED. Hvis værdien er gyldig bliver flaget LEDTimOn i SREG2 sat højt for at sikre at LED værdien ikke bliver overskredet af subrutinen SetLED i et sekund. Herefter shiftes Arg 1 bit til venstre for at passe med LED’ernes placering på PORTA hvilket er fra pin 6 til 1. PORTA hentes ind i Temp1 og bit 6 til 1 ud-maskes så evt. gamle LED værdier ikke påvirker de nye. LED værdien i Arg indsættes i Temp1 da Arg er blevet shiftet til venstre kommer værdien ind på bit 6 til 1. På denne måde påvirkes ADC konverteringen indgang pin 7 ikke. Herefter sendes Temp1 til PORTA. Timer1’s tæller register nulstilles og timeren startes med compare match mode og en prescaler på 1024. Da timer1’s OCR1A register allerede er opsat til at der skal ske et match efter et sek. Vil der nu gå et sek. før et interrupt vil ske. interrupt service rutine vil slukke for LED’eren og sætte LEDTimOn flaget lavt igen så subrutinen SetLED atter kan ændre LED’eren.

StopCar

StopCar subrutinen bruges til at stoppe bilen.

StopCar kræver ingen parameter og returnerer ingen værdi.

StopCar starter med at lægge 0 ind i registret Temp1. Temp1 lægges herefter ind i OCR2 bufferen hvilket vil forårsage at PWM til motoren vil være på 0 % næste gang TCNT2 kommer op på 0xFF. Se side 565 i \cite{AVR}. Derefter bruges PulsLED for at vise SpopCar er blevet kaldt.

Ref:

AVR side 565

CalcOCR2

CalcOCR2 subrutinen beregner ud fra en værdi på 0 til og med 100 svarende til hvor mange procent af den totale hastighed motoren skal køre med. Hvilken værdi der skal ligges i OCR2 hvilket kontrollerer PWM til motoren.

CalcOCR2 kræver en parameter fra 0 til og med 100 i Arg svarene til den ønskede hastighed og returnerer værdien til OCR2 i Ret1.

I bogen ”The AVR Microcontroller and Embedded Systems: Usnig Assenmbly and C” gives ligningen for at udregne duty cycle for phase korrekt PWM i AVR. Se side 569 i \cite{AVR}. Se ligning \ref{eq:CaOCR21} (1 CalcOCR2) Da OCR2 og OCR0 begge er 8 bit register er ligningerne ens for dem.

\begin{equation}

Duty Cycle = \frac{OCR0}{255}\cdot 100 (1 CalcOCR2)

\label{eq:CaOCR21}

\end{equation}

Hvis OCR0 som fremover nævnes som OCR2 isoleres fås ligningen \ref{eq:CaOCR22} (2 CalcOCR2).

\begin{equation}

OCR2 = Duty Cycle \cdot 2.55 (2 CalcOCR2)

\label{eq:CaOCR22}

\end{equation}

For at multiplicere med et decimaltal tal kan fixed-point beregninger benyttes. Hvis 2.55 laves om til Q-Format Q2.6 kan man udføre multiplikation med decimaltal i mikrocontrolleren. Format Q2.6 er det format på fixed-point som med højst præcision udtrykker 2.55 på 8 bit.

Værdien som skal udregnes vil være på format Q7 da en værdi der ligger fra 0 til og med 100 kan udtrykkes med 7 bit. Dermed vil en multiplikatioen resutat være i format Q9.6. Se side 44 i \cite{FixP}. Da en multiplikation mellem 2.55 og en værdi fra 0 til og med 100 maksimalt kan give 255 hvilket kan udtrykkes med 8 bit vil den 9’ne bit i resultatet være 0 og dermed ligegyldig. Resultatet vil dermed kunne udtrykkes med format Q8.6.

CalcOCR2 implementere dette ved at gemme 2.55 som Q2.6 format i Temp1 og derefter udføre en multiplikation af Arg og Temp1. Efter multiplikationen ligger resultatet høje Byte i R1 og lave Byte i R0. Da resultatet som nævnt er i format Q8.6 shiftes R0 til venstre ind i carry og R1 shiftes mod venstre med carry. Begge shift gøres 2 gange for at have resultatet i format Q8 i R1. Til sidst kopieres R1 ind i Ret1 og CalcOCR2 afsluttes.

Ref

AVR side 569

Fixed-Point Signal Processing side 44

Cleanup

Cleanup subrutinen bruges til at nulstille modtagede typer, kommandoer og data i protokollen samt Arg, Ret1, Temp1 og Temp2.

Cleanup kræver ingen parameter og der returneres ingen værdi.

0 ligges ind i registret Arg der derefter gemmes i data Space på placeringerne \verb!InType\_DatSpace! og \verb!InCmd\_DatSpace!. Dermed nulstiller den modtaget type og kommando. Registrene InBesked, Temp1, Temp2 og Ret1 nulstilles også ved at der gemmes 0 i dem.

SetPrescaler

SetPrescaler kræver en parameter i InBesked som er den prescaller man vil have sat. Værdien skal være fra 1 til og med 7 hvilket vælger mellem de 7 prescaller på timer 2. SetPrescaler returnerer ingen værdi.

SetPrescaler starter med at tjekke om InBesked ligger fra 1 til og med 7 hvis den ikke gør dette vil den afslutte uden at påvirke prescalleren. Hvis værdien ligger mellem 1 og 7 hentes TCCR2 hvilket er timer 2 kontrol registret ind i Temp1. Se side 567 i \cite{AVR}. Bit 2 til 0 ud maskes i Temp1 for at fjerne den tidligere prescalleres værdi. Herefter ligges InBesked ind i Temp1 da InBesked allerede er blevet tjekket for at være fra 1 til og med 7 vil den være maks. 3 bit stor og vil dermed ikke påvirke de andre indstillinger i TCCR2. Temp1 lægges til sidst ind i TCCR2 igen så den nye prescaller indstilles.

Ref

ACR side 567

------------------------------------------------------Dette er nyt \/ \/-------------------------------------------------------------------

ChangeState

ChangeState subrutinen gemmer det bane stykke bilen forlader. Subrutinen opdaterer også den angivet state i SREG2 samt angiver den nye state med RGB LED’eren.

ChangeState kræver LastState som input og gemmer det sidste bane stykke i data space samt opdatere state.

ChangeState starter med at kalde subrutinen StoreTrack hvilket vil gemme det bane stykke bilen forlader og nulstille distance tælleren. Herefter opdateres State med værdigen i LastState og der med opdatere hvilket State bilen vil køre efter.

StoreTrack

StoreTrack subrutinen tager den opmålte distance samt hvilket state den er opmålt under og gemmer den i data space.

StoreTrack kræver distance liggende i DistH med høje Byte og DistL med lave DistL. State i bit 7 og 6 i SREG2. StoreTrack gemmer DistH med State i bit 7 og 6 samt DistL i data space.

StoreTrack starter med at ligge state fra SREG2 ind i bit 7 til 6 i DistL. Herefter gemmes DistH på Z’s plaseringen og Z incremeteres. DistL gemmes nu på på Z’s plaseringen og Z incremeteres igen så Z placering er ved den næste ledige placering til næste gang StoreTrack kaldes. Nu nulstilles DistH og DistL samt StoreTrack afsluttes.

StateLED

StateLED subrutinen ændre RGB LED’eren efter hvilket state bilen køre in.

StateLED kræver State og retunere ved hjælp af RGB LED’eren hvilket State bilen mener den køre i.

StateLED starter med at tjekke hvilket State bilen køre i ved brug af bit 7 og 6 i SREG2. Herefter ændres LED’eren til Sving1\_LED, Lige\_LED eller Sving2\_Led at efter hvilket state bilen er i. Dettet gøres ved hjælp af subrutinen LEDSet.

SendTrack

SendTrack subrutinen stopper bilen og sender den opmålte bane.

SendTrack kræver banen i data space og retunere banen over seriel kommunikation.

SendTrack starter med at kalde subrutinen StopCar hvilket stopper bilen. Herefter slukkes for globale interupt. For at vide hvor slutningen på opmålingen er i dataspace gemmes nu Z i Temp1 (høje Byte) og Temp2 (lave Byte). Nu gemmes Z med værdien i ZStart. Herefter kommer SendTrack ind i et loop hvor den vil sende alle track’ne. Dette gøres ved den i loopet indhenter værdien på adressen som Z angiver og post incrematere z. Der efter sendes værdien. Dette gøres mens Z’s værdi er mindre end Z’s værdi da SendTrack blev kaldt. Når værdien er den samme tændes der for global interupt igen og SendTrack afsluttes.

StateVel

StateVel opdatere hastigheden og slukker for elektromagneten.

StateVel kræver ingen parameter. StateVel opdatere bilen hastighed og elektromagneten.

StateVel starter med at slukke elektromagneten og hastigheden bliver sat til værdien Velocity\_Straightellers. Her efter afsluttes StateVel.

CalcOffset

CalcOffset udregner offset af distancen der skal køres for der skal ændres hastighed og elektromagnet status.

CalcOffset kræver den nye state i Arg og retunere den offsatte high byte i Temp1 og lave byte i Temp2.

CalcOffset starter med at gemme Temp1 med StoredDistH og Temp2 med StoredDistL. Herefter udregnes 1/4 af den totale distanse om StoredDistH og StoredDistL angiver, dette resultat gemmes i Ret1. Temp2 subtrahere med Ret1, og Temp1 med carry. Hvis state var en af svingen ville den afslutte med det samme.

AvgAcc

AvgAcc tager 256 AccData og udregner gennemsnittet af dem.

AvgAcc kræver acclometer konvaseringen resutatet til at ligge i AccData. Efter gennemsnittet er udregnet ligger det i Ret1.

AvgAcc Starter med at sætte T flaget i SREG til 0. Dette gøres for at angive den nuværende værdi i AccData er blevet brugt. Her efter adderes AccData i AccSumL og AccSumH med carry. På denne måde adderes AccData’eren ind i et 16 bit register. Et 16 bit register kan indeholde en maksimal værdi på 65535. Hvis der adderes 256 gange den maksimale værdi fra 8 bit registret AccData hvilket er 255. Vil værdien være på 65280 så et 16 bit register er tilstrækkelig til dette. Derefter bliver DivCounter tjekker for at være kommet op på 255. Hvis den ikke er dette adderes DivCounter med 1. Hvis den er 255 bliver AccSumH kopiret over i Ret1. Ved at tage de 8 høje bit divideres der med 256. Her efter afsluttes AvgAcc. AvgAcc må kun kaldes når T flaget er højt dette sikre at der ikke benyttes den samme AccData værdi flere gange da T flaget sættes højt når der er en ny værdi fra ADC konverteringen. For at vide hvornår AvgAcc har en færdig værdi skal DivCounter efter AvgAcc er kaldet tjekkes da DivCounter kun er 0 hvis der er opnået et gennemsnti efter AvgAcc er kaldet.