PROFIBUS DP

Implementering av Profibus DP i en AVR-mikrokontroller

Andreas Junell

Examensarbete för ingenjör (YH)-examen

Utbildningsprogrammet för elektroteknik

Vasa 2012

EXAMENSARBETE

Författare:	Andreas Junell
Utbildningsprogram och ort:	Elektroteknik, Vasa
Inriktningsalternativ/Fördjupning:	Automationsingenjör
Handledare:	Dag Björklund

Titel: *Implementering av Profibus i en AVR-mikrokontroller*

Datum 22.05.2012 Sidantal 25 Bilagor 5

Sammanfattning

Detta examensarbete handlar om att implementera Profibus DP i en Atmega644P mikrokontroller, och på så sätt få en Profibus DP-slav som kan användas i utbildningssyfte vid Yrkeshögskolan Novia. Detta examensarbete har inneburit att tillverka ett tilläggskort till det befintliga mikrokontrollerkortet för att kunna koppla in det till en fältbuss som använder sig av RS485-kommunikation. Det har även tillverkats en programkod för Profibus DP-kommunikation till själva mikrokontrollern för att den skall kunna fungera som slav i en Profibus DP-fältbuss. Detta examensarbete har innefattat kretskortsdesign och portande av C-kod.

Språk: svenska Nyckelord: profibus, fältbuss, AVR, mikrokontroller

---

BACHELOR'S THESIS

Author:	Andreas Junell
Degree Programme:	Electrical Engineering
Specialization:	Automation Engineering
Supervisor:	Dag Björklund

Title: *Implementation of Profibus DP in an AVR microcontroller*

Date 22.05.2012 Number of pages 25 Appendices 5

Summary

This Bachelor's thesis is about implementing Profibus DP on an Atmega644P microcontroller, so that it can be used in the education at Novia University of Applied Sciences in Vaasa. The first part of this thesis was to make an expansion card for the microcontroller that is used in education so it can be connected to an RS485 fieldbus. The second part of this thesis was to port a program code for the microcontroller that was originally implemented for another microcontroller to get it to work with the Atmega664P microcontroller.

Language:Swedish Key words: profibus, fieldbus, AVR, microcontroller

---

Innehållsförteckning

Bilagor

1. Inledning

Detta examensarbete har utförts åt Yrkeshögskolan Novia i Vasa, enheten för teknik. Syftet med detta arbetet är att utveckla hårdvara och mjukvara för undervisningen.

1.1. Beskrivning av uppgiften

Uppgiften är att utveckla hårdvara, mjukvara samt kunskap som stöd för undervisningen i datakommunikation och/eller mikroprocessortillämpningar.

Uppgiften består av en undersökning av Profibus fältbussen, den existerande utrustningen för Profibus-kommunikation samt utveckling av en Profibus-slav på de existerande mikrokontrollerkort som kan kommunicera med övrig Profibus-utrustning.

Jag valde att dela upp arbetet i ett par huvudfaser: Bygg RS485-tilläggskort till mikrokontrollerkorten och implementering av Profibus högre protokollager som mjukvara i AVR.

1.2. Uppgiftsspecifikation

Följande uppgiftsbeskrivning fick jag initialt:

1. Bygg tilläggskort. Dokumentera och beskriv vad som händer elektriskt. Tri-state logik, pull up resitorer?
2. Undersök och beskriv Profibus lager II. Läs samtidigt, och försök förstå, Profibus-koden för AVR-kontrollern.
3. Porta koden till ATmega644P och gärna PC (Linux). Identifiera det mikrokontrollerspecifika i koden och bryt ut det i skild fil. I alla fall allt som har att göra med seriekommunikation är kontrollerspecifikt. Försök göra detta ”HW abstraktion layer” så tunt som möjligt.

1.3. Uppdragsgivare

Yrkeshögskolan Novia har fungerat som uppdragsgivare för detta examensarbete. Dag Björklund vid Yrkeshögskolan Novia har fungerat som handledare. Yrkeshögskolan Novia har ca. 4000 studerande och personalen består av ca. 400 personer. Det är den största svenskspråkiga yrkeshögskolan i Finland.

2. Teori

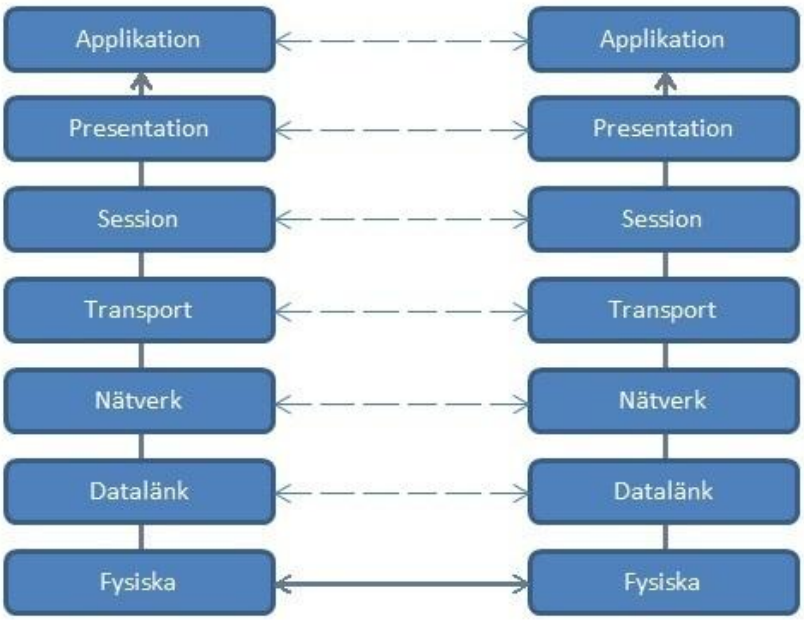
Detta kapitel kommer att behandla nätverksteori samt teori om Profibus.

2.1. Nätverksteori

I detta kapitel berättas det om OSI-modellen som är en referensmodell för hur datakommunikation skall utföras. Olika nätverkstopologer kommer även att behandlas.

2.1.1. OSI-Modellen

OSI-modellen fungerar som en referens för hur datakommunikation skall utföras. Denna modell fungerar som en röd tråd i hur kommunikationen skall byggas upp mellan olika tillverkares system och applikationer. För att två enheter skall kunna kommunicera så behövs ett antal tekniker som definieras i olika protokoll. Dessa protokoll definierar hur dataöverföringen skall gå till. Enligt OSI-modellen är kommunikationen uppbyggd i olika lager. De olika lagren kan delas upp i två kategorier, lager 1–4 är nätverksorienterade, medan lager 5–7 är användarorienterade. Om man t.ex. sänder data via det sjunde lagret byggs meddelandet på med ”headers” från de övriga lagren, ända ner till det andra lagret där meddelandet blir komplett. Därefter går data vidare till det första lagret varifrån bitströmmen med ettor och nollor överförs till mottagaren. (*4*)



Figur 1. *OSI-modellen*

OSI-modellen beskriver de sju olika lagrens uppgifter enligt följande:

- Det första lagret är själva överföringsmediet och den elektriska signaleringen. Det kan vara en radiolänk eller en kabel.
- Det andra lagret, datalänklagret, är en länk mellan det fysiska lagret och nätverkslagret. Detta lager består av två underlager, MAC och LLC. De sköter om bl.a. adressering, att sända och ta emot dataramar och feldetektering.
- Det tredje lagret, nätverkslagret, beskriver vilken standard som används för upp- och nedkoppling av förbindelser, adressering och vidaresändning av data. Det kan t.ex. stöda den logiska adresshantering då flera olika nätverk skall kommunicera.
- Det fjärde lagret, transportlagret, upprättar en förbindelse mellan processerna både i sändande och mottagande riktning. Transportlagret säkerställer kommunikationen mellan användarna genom att identifiera användarna och ta hand om fel som inte hanteras av de lägre lagren (lager 1–3).
- Det femte lagret, sessionslagret, beskriver standarden för den logiska kopplingen, dataflödet, synkronisering och buffring av data mellan de olika enheterna och ändprocesserna. Lagret fungerar också som ett gränssnitt mellan t.ex. nätverksprogramvaran och program högre upp i hierarkin.
- Det sjätte lagret, presentationslagret, beskriver kryptering och komprimering av data, samt kod- och formatkonvertering. Detta lager hjälper det sjunde lagret att tyda data.
- Det sjunde lagret, applikationslagret, beskriver standarden för gränssnittet mellan avsändaren och mottagaren. Detta är alltså själva användargränssnittet, som t.ex. kan vara en webbläsare eller ett HMI.

(*4*)

Det finns tre olika varianter av Profibus: DP, PA och FMS. Profibus DP och Profibus PA använder sig endast av de två lägsta lagren från figur 1 ovan, medan Profibus FMS använder förutom de två lägsta även det lager som är högst upp i modellen. *(3)*

2.1.2 Nätverkstopologier

Ett nätverk kan fysiskt byggas upp på olika sätt. Dessa olika sätt kallas nätverktopologier. Själva kabeldragningen kan se ut på bl.a. följande olika sätt:

- Stjärnnät
- Ringnät
- Bussnät

Det finns både positiva och negativa sidor med de olika alternativen.

Stjärnnätet har fått sitt namn från att nätet ser ut som en stjärna med en central nod i mitten, vilken kan vara t.ex. en hubb eller en switch. Denna nätverkstyp är vanlig i hemnätverk där man har kopplat datorer till en router. Fördelarna med ett stjärnnät är att det är enkelt att bygga ut så länge det finns lediga portar på centralnoden. Om det skulle uppstå ett fel på en av noderna som är kopplade till ett stjärnnät påverkar detta inte de övriga noderna. Men om centralnoden skulle få ett fel påverkas hela nätverket. En annan nackdel med stjärnnätet är att det går åt mycket mera kabel än i ett bussnät eller ett ringnät. *(4)*

Ringnätet är uppbyggt likt en ring med alla noder efter varandra. Varje nod måste ha två anslutningar, för att kunna ta emot och skicka vidare data. När någonting sänds i ett ringnät går det via varje nod tills det kommer fram dit det har adresserats. Enligt boken *Datakommunikation i Praktiken* blir det således en fördröjning på ca 3–5 bitar i varje nod. Fördelerna med ett ringnät är att det är enkelt uppbyggt. Nackdelarna med ringnät är att det är störningskänsligt och svårt att bygga ut. En annan nackdel är om ringen bryts (t.ex. en felaktig nod) kan hela nätet slås ut. *(4)*

Bussnätet består av en lång kabel till vilken alla noder är anslutna. Alla noderna tar emot data samtidigt, men endast en nod kan sända data. Om flera noder försöker sända samtidigt uppstår kollisioner och data går kan förloras. Från figur 2 ser man ett exempel på ett bussnät. I kabelns ändar måste det finnas terminatormotstånd som avslutar bussen. Det är enkelt att bygga ut bussen för fler noder. Eftersom alla noder tar emot all data är det mycket enkelt att göra ”broadcasting”. Nackdelarna med bussnät är att om flera noder försöker sända samtidigt uppstår det kollisioner som slöar ner bussen och kan leda till förlorad data. Om kabeln går sönder kan samtliga noder slås ut. Det kan även vara svårt att felsöka större bussnät.

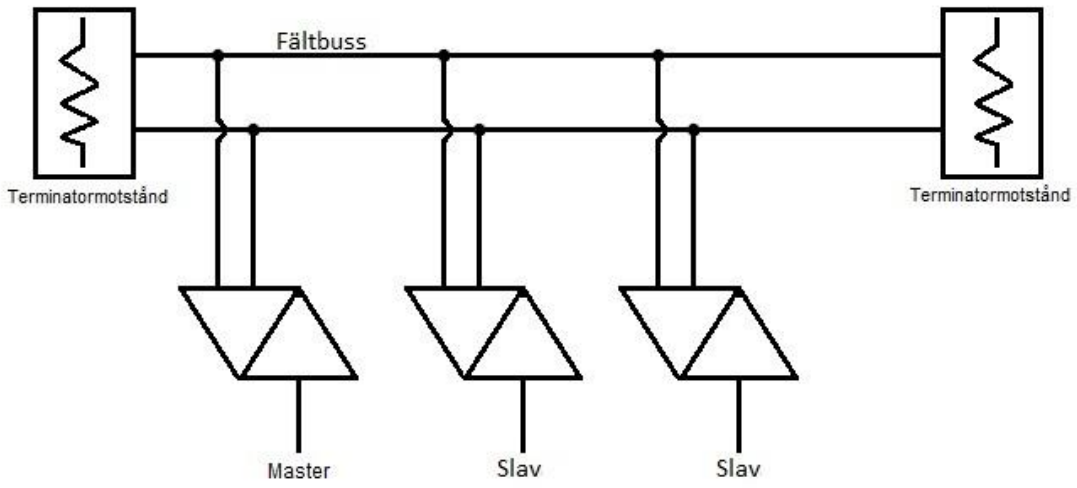
2.2. Profibus

I detta kapitel behandlas teori för Profibus. Först kommer det lite allmän teori om Profibus, de olika lagren som används enligt OSI-modellen samt profiler och konfigurering.

2.2.1. Allmänt

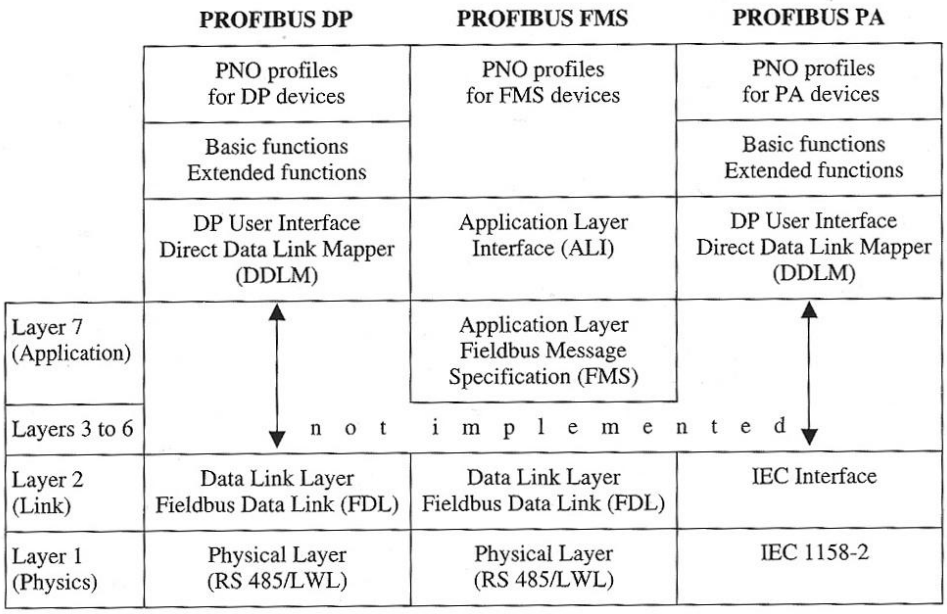
Profibus (PROcess FIeld BUS) är ett öppet fältbussystem för automationsapplikationer. Utvecklingen av denna standard startade 1987 i Tyskland när 21 företag kom överens om att utveckla en fältbuss för seriekommunikation baserad på de enklaste kraven för fältutrustningens gränssnitt. Målet med detta var att utveckla en fältbuss som fungerar med seriell kommunikation. Först kom det komplexa protokollet Profibus FMS (Fieldbus Message Specification), som är anpassat för krävande kommunikationsuppgifter. 1993 kom det ett betydligt enklare och snabbare protokoll kallat Profibus DP (Decentralized Peripherals). Profibus FMS används för kommunikation mellan olika Profibus master enheter, medan Profibus DP används för kommunikation mellan masterenheter och mellan masters och deras I/O-slavar.(*3*)

I figur 2. ser man hur en fältbuss kan se ut, med en master och två slavar, samt terminatormotstånd i båda ändarna.



Figur 2. *Exempel på en fältbuss.*

Profibus i allmänhet använder sig endast av tre lager i OSI-modellen, det första, andra och sjunde lagret. I Profibus DP används endast det första och det andra lagret. Det sjunde lagret används endast vid kommunikation med Profibus FMS som är ett lite mera komplext protokoll.



Figur 3. *Protokollarkitekturen för Profibus. (3)*

I figur 3 ser man hur de olika Profibus-protokollen anpassar sig till OSI-modellens hierarki.

I det första lagret som är det fysiska lagret för själva överföringen används RS485, optisk och MBP (Manchester Bus Powered) kommunikation. Utöver de två lagren som används i Profibus DP finns det även olika användarprofiler, eftersom Profibus DP-protokollet endast definierar hur data blir överfört från en nod till en annan över fältbussen. Profilerna gör att det blir enkelt att blanda Profibus DP-komponenter från olika tillverkare. Profilerna är applikationsspecifika, de specifierar hur en Profibus DP-enhet skall kommunicera med t.ex. en frekvensomriktare. Det finns några specifierade protokoll i dags dato, de kommer att tas upp i ett senare kapitel. *(3, 5)*

Härnäst följer en genomgång av Profibus DP-protokollet, lager för lager.

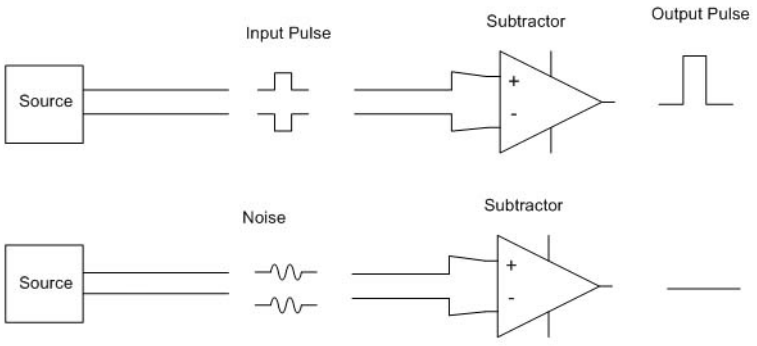
2.2.2. Lager 1, Fysiska lagret

Som fysiskt lager i Profibus DP används RS485. Standarden för RS485 ger endast rekommendationer om kablage, men Profibus preciserar det till tvinnade parkablar med

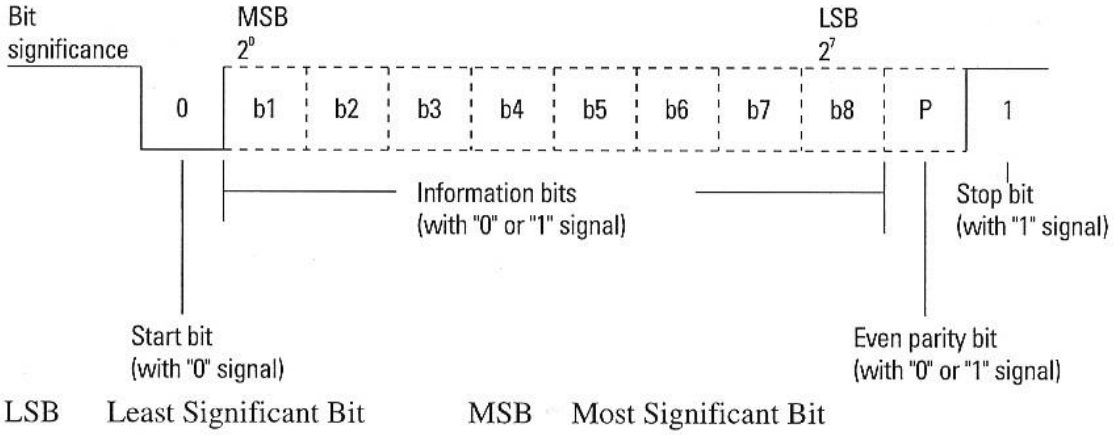
impedansen 150 ohm. Med denna teknik så kan överföringshastigheten vara från 9,6 kbit/s och upp till 12 Mbit/s. Kabellängderna mellan repeatrarna kan vara mellan 100 m och 1200 m beroende på överföringshastigheten. Spänningsnivåerna som används inom RS485-kommunikation är -5V och +5V. Med detta kommunikationssätt kan det anslutas högst 32 noder per segment, dvs. mellan två repeatrar. Detta på grund av att signalen i fältbussen annars skulle bli för svag. En repeater fungerar med andra ord som en linjeförstärkare. RS485 är en standard som används för seriell kommunikation där det behövs höga dataöverföringshastigheter, eller långa avstånd. (*4*)

Differentiell signalering motverkar yttre störningar som kan absorberas av själva kabeln, och även sådana störningar som uppkommer när potentialen för jorden i de olika noderna varierar. Vid användning av halv-duplexkommunikation räcker det med endast en skärmad parkabel för att skicka och ta emot data. Ena ledaren är för den positiva signalen och den andra är för den negativa signalen. När det sänds en binär ”1” så blir den positiva linjen en spänning på +5 V och den negativa linjen en spänning på -5 V. Denna överföringsmetod gör systemet säkert för yttre störningar. I figur 4 ser man att när data tas emot subtraheras den negativa signalen med den positiva. På detta sätt får man fram den slutgiltiga signalen. Om det skulle förekomma störningar på linjen, är dessa lika på den positiva och den negativa tråden, och när dessa sedan subtraheras med varandra tar störningarna ut varandra och försvinner.

Överföringsmetoden som används i lager ett är baserad på en halv-duplex, asynkron, gapfri synkronisering. Data överförs i 11-bitars kodord, eller tecken, enligt figur 5, med NRZ-kodning (Non Return to Zero).



Figur 4. *Differentiell signalering. (6)*



Figur 5. *Exempel på ett kodord. (3)*

I lager 1 sänds och tas dataramar, eller kodord emot, vilka består av en startbit, en paritetsbit, en stoppbit samt åtta bitar data (figur 5), Startbiten är alltid en 0:a och stoppbiten är alltid en 1:a, medan paritetsbiten kan vara en 0:a eller en 1:a beroende på databitarna. Inom Profibus används jämn paritetsbit, om antalet ettor i databitarna är udda så blir paritetsbiten en etta, om antalet ettor är jämn blir då paritetsbiten en nolla.

Om endast det första lagret, dvs. RS485, används så leder detta till att informationen från en nod sänds ut till alla noder. Ingenting förhindrar heller att två noder börjar sända samtidigt, vilket leder till kollisioner och förlorad data. I lager 1 finns det ingen felkontroll eller korrigering, förutom paritetskontroll, utan dessa återfinns i lager 2.

2.2.3. Lager 2, Datalänklagret

Det andra lagret enligt OSI-modellen är datalänklagret, där definieras de olika ramarna som används vid överföring av data. Vidare sköter lager 2 felhantering och multipel access även kallat media access (adressering, kollisionshantering osv.). I Profibus DP kallas det andra lagret FDL (Fieldbus Data Layer). Lager 2 kan (som i OSI-modellen), delas upp i två underlager (sublayers) det ena är MAC (Media Access Control) sublayer och det andra är LLC (Logical Link Control) sublayer.

LLC-sublagret

LLC-sublagret sköter om feldetektering och eventuell återsändning. Även själva ramstrukturen definieras här. I Profibus DP talar man inte om ramar i lager 2 (som i OSI-modellen), utan om telegram.

Feldetekteringen sköts med en ”Frame Check Sequence” som är en byte i slutet av varje telegram, informationen till denna byte fås fram genom att AND:a ihop ramens innehåll. Om det data som har mottagits varit korrekt sänds en kvittering tillbaka till mastern. Om datat är korrupt krävs en återsändning. Med Profibus DP kan man sända data med kvittering och minst två återsändningar. Det går att konfigurera upp till åtta återsändningar. Inom Profibus används det fem olika typer av telegram vid överföring:

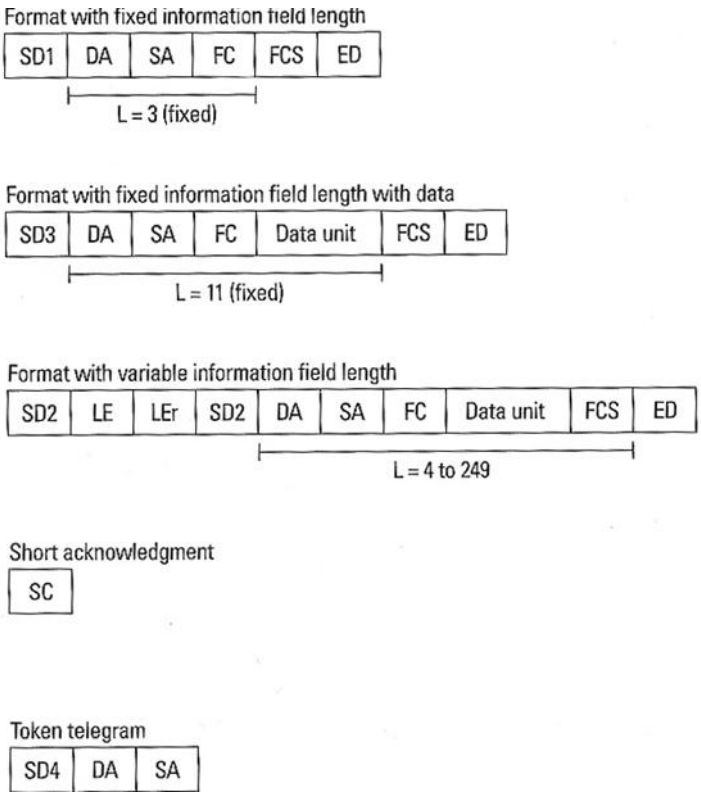
SD1 innehåller destinations- och ursprungsadresser och kontrollbyte, samt start- och stoppbyte och även checksumma.

SD2 används när mängden data i telegrammet varierar. Det kan innehålla upp till 245 bytes med data. Utöver start-, stoppbyte och checksumma innehåller detta telegram även 2 längdbytes som indikerar längden på telegrammet.

SD3 är i övrigt samma som SD2 men man har en fast datalängd på 8 byte.

SD4 är ett så kallat ”token” telegram. Detta används för att signalera åt en master att bussen är ledig.

SC används för att bekräfta att data har mottagits rätt.



Figur 6. *Beskrivning av de olika telegrammen som används i Profibus. (3)*

Förklaringar på de olika förkortningarna:

L = Längden på data.

SD1 – SD4 = Start byte, väljer formatet på telegrammet.

LE/LEr = Längd byte, indikerar längden på informationsfältet i telegrammet med variabel datalängd.

DA = Destinationsadress, indikerar vart telegrammet skall skickas.

SA = Källans adress, indikerar adressen på källan som skickat telegrammet.

FC = Kontrollbyte, innehåller information om vilka tjänster som använts i telegrammet samt prioriteten för meddelandet.

Data = Innehåller telegrammets data.

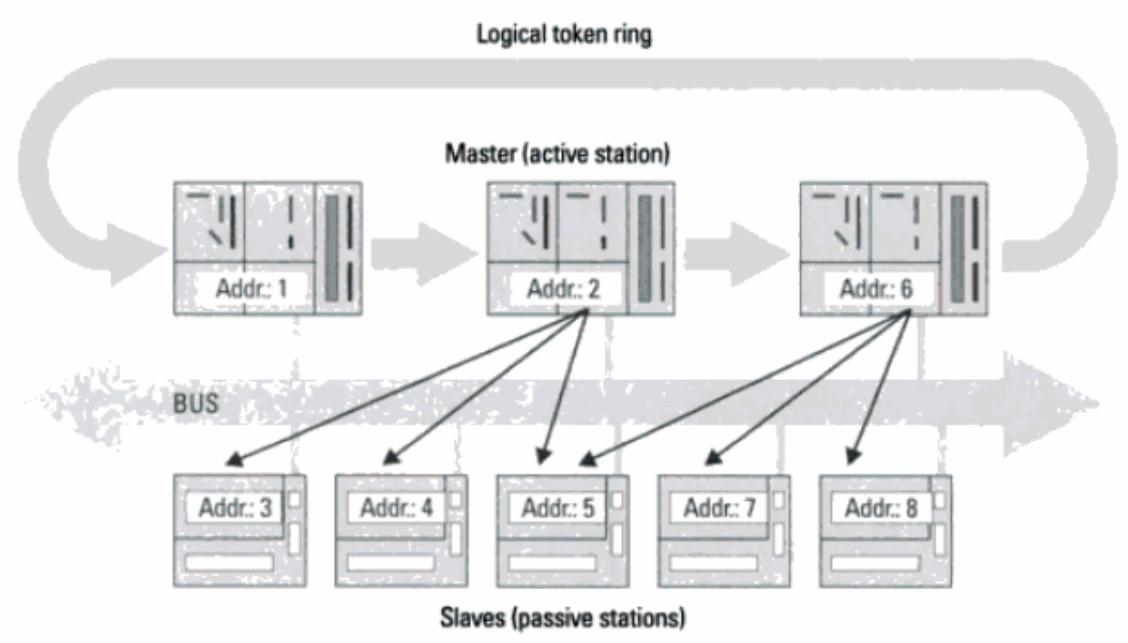
FCS = Innehåller telegrammets checksumma

ED = Stoppbyte, indikerar slutet på telegrammet.

(3)

Man använder sig av två olika MAC- (Medium Access Control) tekniker inom Profibus DP. Den ena tekniken är för kommunikationen mellan en aktiv och en passiv nod, även kallat master/slave-tekniken. Denna teknik används när det endast finns en aktiv nod (master) och simpla passiva noder (slave). Dessa passiva noder kan vara t.ex. ett relä, en givare eller en annan simpel I/O-enhet. En passiv nod kan aldrig initiera kommunikation med andra slavar eller mastern, det är alltid mastern som initierar kommunikationen med en slav som sedan en slav kan svara på. Detta gör att två noder inte kan tala i munnen på varandra, på det här sättet undviker man kollisioner i trafiken.

Den andra tekniken som används är då man använder sig av flera aktiva noder, denna teknik kallas ”Token bus procedure”. Detta används för att alla de aktiva noderna skall få lika stor möjlighet att sända på fältbussen. Det finns en polett (”token”) som skickas mellan de aktiva noderna, vilket gör att de i tur och ordning kan kommunicera på fältbussen. Den master som har poletten är den som får inleda kommunikationen på bussen. Då ingen annan än den som har poletten får kommunicera på bussen resulerar detta i att kollisioner inte kan inträffa. *(3, 4)* I figuren här nedan ser man hur detta kan se ut.



Figur 7. *Visualisering av ”Token bus procedure”. (3)*

2.2.4. Profibus-profiler

Det finns några olika användarprofiler specifierade för olika scenarion som Profibus DP kan tänkas användas vid.

Profil för NC/RC (3.052)

Denna profil beskriver hantering och installation av robotar kontrollerade via Profibus DP. Det finns sekvensdiagram som beskriver rörelse- och programkontroll sett från en högre automationsstations synvinkel.

Profil för avkodning (3.062)

Denna profil beskriver hur axelgivare, axelvinkelgivare och linjära avkodare med envarvig eller flervarvig upplösning kopplas till Profibus. Två olika klasser av apparater definierar både enkla och avancerade funktioner såsom skalning, alarmhantering och diagnostisering.

Profil för frekvensomvandlare (3.072)

Flera olika ledande tillverkare av frekvensomvandlare har slagit sig samman för att utveckla PROFIDRIVE-profilen. Profilen anger hur olika parametrar skall ställas in, samt även hur är-värden och bör-värden skall överföras. Denna profil innehåller specifikationer som krävs för att köra frekvensomvandlaren i lägena för ”hastighetskontrollering” och ”positionering”. Det finns angivet de enklaste funktionerna för frekvensomvandlare, medan det ändå lämnas frihet för programspecifika tillägg och vidare utveckling.

Profil för kontroll och övervakning, HMI (3.082)

Detta är en profil som definierar hur olika användargränssnitt skall kopplas in till en Profibus DP-fältbuss. Denna profil kan även användas för datakommunikation över fältbussen.

Profil för felsäker dataöverföring via Profibus DP (3.092)

Denna profil definierar olika säkerhetsmekanismer för kommunikation med olika säkerhetsapparater, t.ex. nödstopp. Säkerhetsmekanismerna för denna profil har blivit godkända av tyska TÜV och amerikanska BIA.

Dessa profiler baserar sig på en programnivå som är högre än det hos applikationslagret enligt OSI-modellen. *(3)*

2.2.5. Konfigurering av enheter.

För konfigurering av fältenheter använder man sig av en så kallad GSD-fil, som är en konfigurationsfil som beskriver kommunikationsegenskaperna för en Profibus-enhet. Filen beskriver produktindenfikation, parametrar och datatyper. Styrenheten måste ha tillgång till alla fältenheters GSD-filer för att kommunikationen skall kunna fungera med respektive enhet. *(3)*

2.3. Mikrokontrollern

ATmega644P är en 8-bit mikrokontroller tillverkad av Atmel. Några av dess funktioner är att den har 32 binära I/O-portar och 8-kanals 10-bits analog till digital omvandlare. Den har även två programmerbara USART-portar samt två 8-bitars timers och en 16-bits timer. Den har en intern oscillator som går på klockfrekvensen 8 MHz.

Den kan matas med spänningarna 2,7–5,0 V och med en extern oscillator kan den arbeta med klockfrekvenser på upp till 20 MHz, men då krävs en matningsspänning på minst 4,5 V.

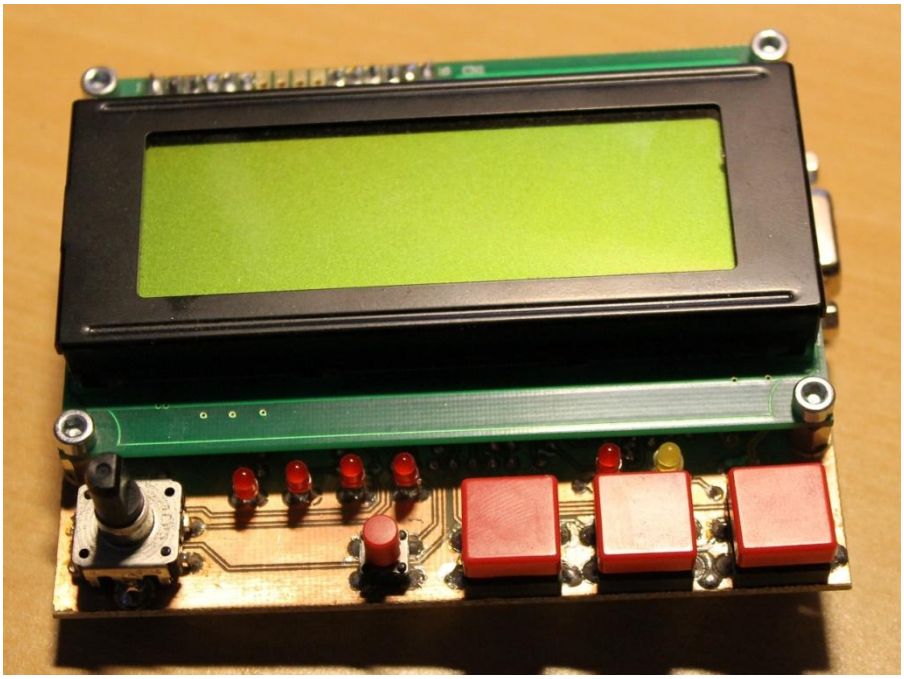
Programmeringsmöjligheterna för mikrokontrollern är via JTAG- och SPI-gränssnitt. Mikrokontrollern har flera funktioner för de flesta av pinnarna.

Mikrokontrollerns binära ingångar tolkar en spänning mellan +3 V och +5,5 V som en binär etta och en spänning mellan –0,5 och +1,5 V som en binär nolla. Dessa värden gäller vid en matningsspänning på 5,0 V.

Mikrokontrollern består av olika register som är uppbyggda av åtta bitar. Bland dessa register finns ett *Control Register* som styr mikrokontrollerns kringutrustning, där man kan ändra t.ex. *baud rate* för en USART, ändra hur snabbt en timer räknar samt när eller hur ett externt avbrott inträffar.

En USART är en krets som styr seriekommunikationen med mikrokontrollern. Det finns två av dessa i ATmega644P. En USART kan fungera både asynkront (synkroniserar med startbit) eller synkront (synkroniserar med klocksignal).

Till en USART:s uppgift hör att skicka och ta emot data seriellt. Mikrokontrollern bearbetar alltid data parallellt så en USART omvandlar all parallell data till seriell data vid sändning och tvärtom vid mottagning. Den ser till så att de seriella bitarna skickas med önskad baudrate. Till uppgifterna hör även att tillföra de nödvändiga start- och stoppbitarna, samt även eventuella paritetsbitar till bitströmmen. (*1*)



Figur 5. *Mikrokontrollerkortet.*

3. Utförande

Detta kapitel handlar om hur examensarbetet utfördes. I det första avsnittet handlar det om tilläggskortet som fungerar som lager 1 i Profibus DP, i det andra kapitlet om koden som fungerar som lager 2.

3.1. Tilläggskortet

Huvudkomponenten på tilläggskortet är en RS485-omvandlare som både kan sända och ta emot signaler. Den består av två skilda delar för de båda funktionerna, den ena för att skicka data och den andra för att ta emot data. Omvandlaren använder en aktiveringsingång, där man väljer vilken funktion som skall användas. Här nedan visas funktionstabellerna på de olika ingångarna respektive utgångarna.

Tabell 1. *Funktionstabell på differentialutgångarna*

Ingång D	Aktivera DE	Utgång A	Utgång B
H	H	H	H
L	H	L	L
X	L	Z	Z

Från tabellen ovan ser man hur utgångarna A (positiva) och B (negativa) fungerar när aktiveringsingången (DE) är hög (H). När DE blir låg (L) går utgångarna till högimpedansläget (Z), och påverkas ej av signal på ingången D. Denna tabell förklarar således hur omvandlaren sänder på fältbussen.

Tabell 2. *Funktionstabell på differentialingångarna*

Differential Ingång A-B	Aktivera $\underline{RE}$	Utgång R
$V_{ID} \geq 0.2$ V	L	H
-0.2 V < V_{ID} < 0. 2V	L	?
$V_{ID} \leq$ -0.2 V	L	L
X	H	Z

Tabell 2 visar hur omvandlaren tar emot data från fältbussen med en differentialingång som subtraherar den negativa signalen från den positiva. Denna del av omvandlaren fungerar när den inverterade aktiveringsingången är låg. När denna är hög går utgången in i högimpedansläget.

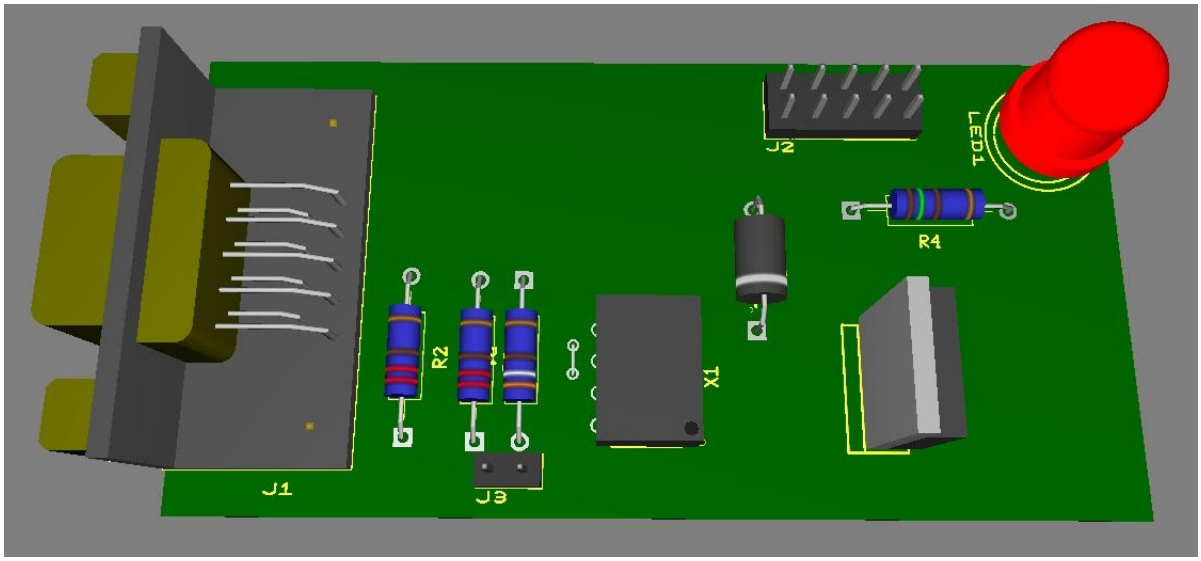
När utgångarna är i högimpendansläget är de flytande och påverkar inte resten av fältbussen. (2)

När data skall sändas sätts de aktiverande ingångarna till höga. Detta utföres så att dessa ingångar kopplades ihop och till den sändande ingången. De aktiverande ingångarna kopplades även via en diod till jord så att de är låga när ingenting sänds, då är mottagardelen aktiverad. Dioden är kopplad så att den spärrar spänningen från att ledas till jord.

På kortet finns ett så kallat terminatormotstånd, motståndets uppgift är att fungera som ett avslut på fältbussen. Med en jumper (J3 på kretskortet) kan man välja om man vill använda terminatormotståndet eller ej. Värdet på detta motstånd har definierats enligt standarderna för Profibus DP från boken *Decentralization with PROFIBUS DP/DPV1*. Från samma bok har även värdena fåtts till ”pull-up” och ”pull-down” resistorerna. Dessa gör att de positiva och negativa linjerna blir och hålls höga när ingen data sänds.

Själva omvandlaren är en SN75168 som är designad för balanserade differentiala signalöverföringslinjer och motsvarar kraven för *ANSI Standard EIA/TIA-422-B* (Amerikanska standardinstitutet) och *ITU Recommendation V.11* (Internationella Telekommunikationsinstitutet). SN75168 använder sig av endast +5 V för att driva kretsen.

För att stabilisera spänningen på +5 V har jag använt en AN7805-spänningregulator, som säkerställer att spänningen hålls på endast +5 V. Den klarar av inspänningar på upp till +35 V.

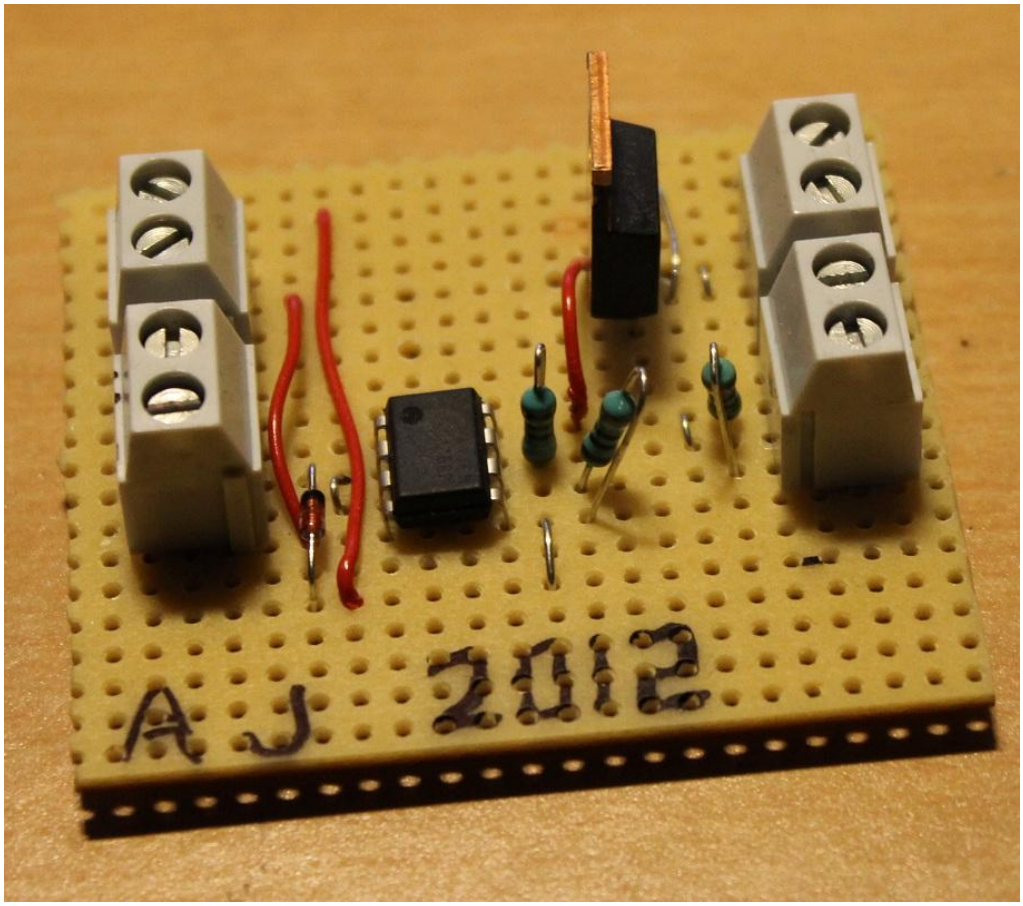


Figur 6. *3D-modell av tilläggskortet.*

Från figuren ovan ser man att jag har valt att sätta en DB9-kontakt på tilläggskortet. Detta var ett önskemål av beställaren av kortet. Till DB9-kontakten skall man ansluta själva fältbussen. Det finns befintliga Profibus DP-kablar vid skolan som använder sig av detta kontaktdon.

Mikrokontrollern ansluter man till J2 på kretskortet.

Det har tillverkats endast en prototyp av kretsen, vilken jag byggde upp på ett så kallat ”*breadboard*” kretskort. Orsaken till att jag använde mig av denna lösning för prototypen var att man enkelt skulle kunna testa om kretsen överhuvudtaget fungerade. Man kunde enkelt koppla in trådarna från fältbussen och den kommunicerande enheten till kretsen via skruvplintar. Kretsen fungerade bra när den testades i en fältbuss med fabrikstillverkade omvandlare. Fältbussen testades genom att man kopplade in omvandlarna till en dators usb-portar med usb→RS232→RS485 omvandlare och konfigurerade dem som skilda enheter och körde simpel kommunikation via programmet Hyperterminal. Om man använder endast denna typ av kommunikation kan alla noder sända på samma gång och alla tar emot allting. Detta kan leda till många kollisioner inom överföringen, och då går mycket data förlorat. Så därför behövs det även mjukvara som sköter om själva kommunikationen. Ritningar och materiallista för tilläggskortet finns som bilaga. (*2*, *4*)



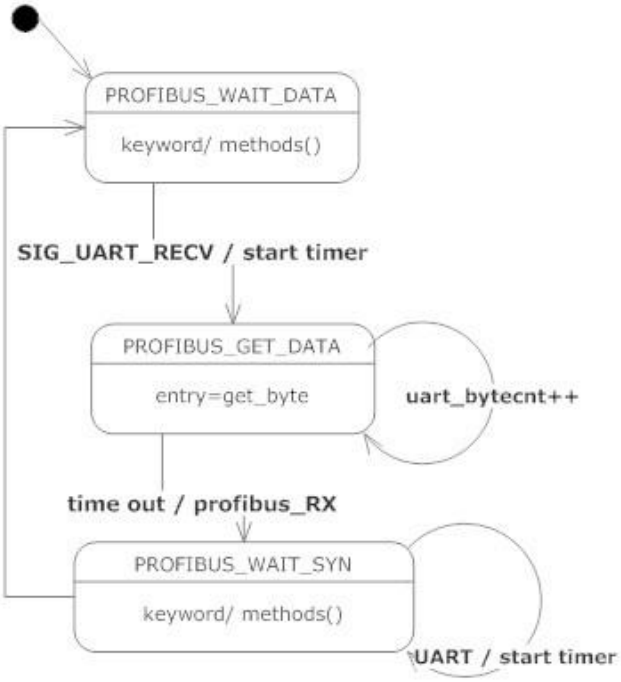
Figur 70. *Prototyp av tilläggskortet på ”breadboard”.*

3.2. Koden

Som programkod använde jag en färdig kod som var skriven för en annan mikrokontroller, som sedan portades om för att fungera på Atmega644P, som är den mikrokontroller som användes i detta lärdomsprov. För att utföra detta så användes programmet AVR-studio som har tillverkats av Atmel. Programmet är avsett för deras mikrokontroller och har inbyggd simulator som gör att man kan provköra koden före man kör in den i mikrokontrollern.

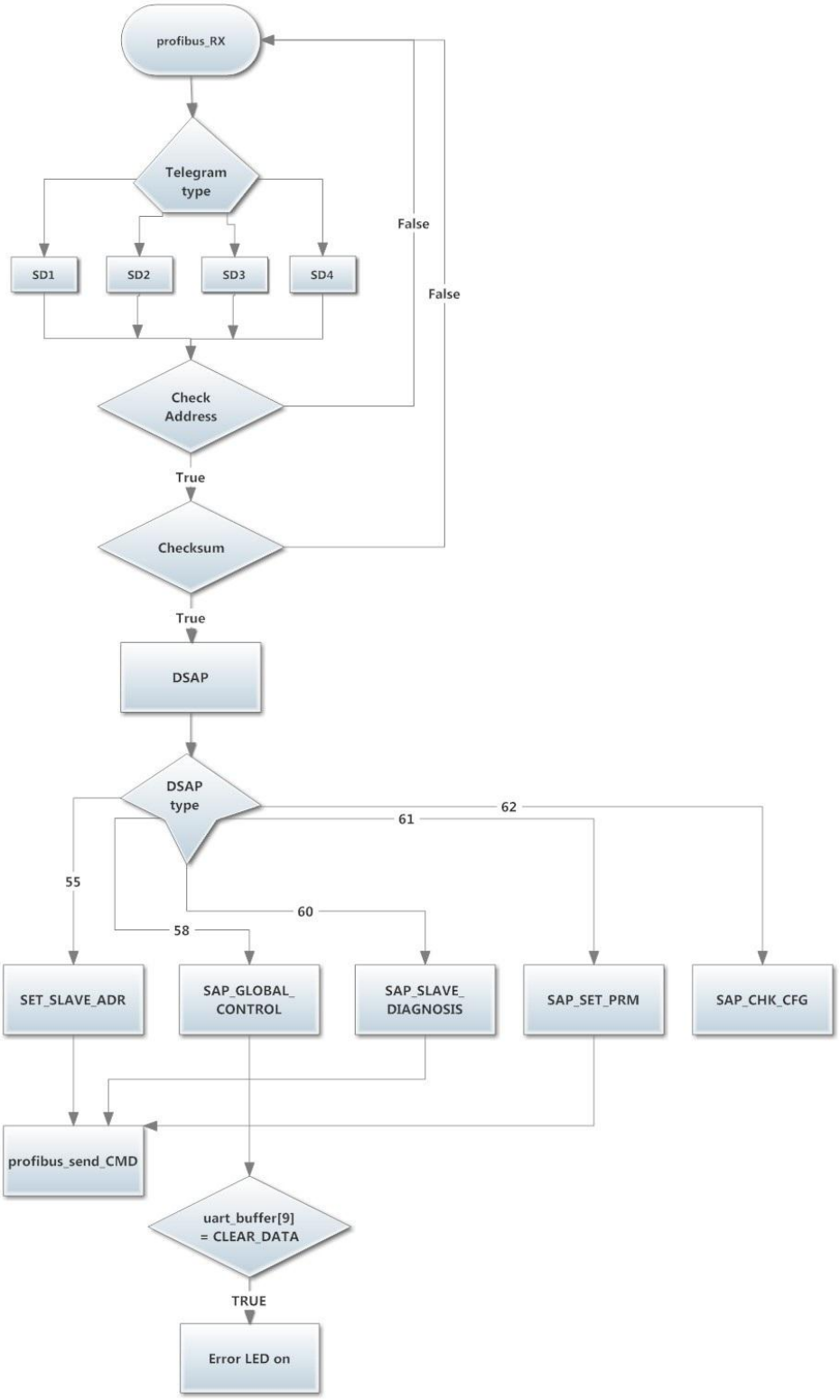
Själva portandet av koden utförde jag inledningsvis enligt ”*Trial and Error*”-metoden. Det gick till så att jag först provade att kompilera den gamla koden för att se hur många fel det fanns i den, sedan var det bara att börja beta av felen i ordningsföljd. När sedan alla felen som uppstod var åtgärdade prövade jag att köra koden i en simulator som finns i AVR-studioprogrammet. I simulatorn gick det att köra koden steg för steg, där kunde man se hur programmet hoppade in i de olika funktionerna.

För att bygga upp förståelse för koden har jag illustrerat den med olika tillståndsmaskiner och flödesscheman. Där ser man hur koden är uppbyggd.



Figur 81. *Tillståndsmaskin: Mottagning av data.*

Från figur 12. ser man att programmet först är i ett vänteläge där det väntar på att det skall komma data till USART:en. När det sedan kommer in data (det första kodordet i ett telegram) startas en timer som räknar ner, samtidigt som de följande kodorden i telegrammet tas emot. Variabeln `uart_bytecnt` håller reda på hur många tecken (kodord) som mottagits. När timern når noll och det inte kommer mera går vi över till följande tillstånd (`PROFIBUS_WAIT_SYN`), där det identifieras vilken typ av telegram det gäller. Funktionen `profibus_RX` anropas vid övergången till detta tillstånd. Denna funktion illustreras som ett flödesdiagram i figur 13.



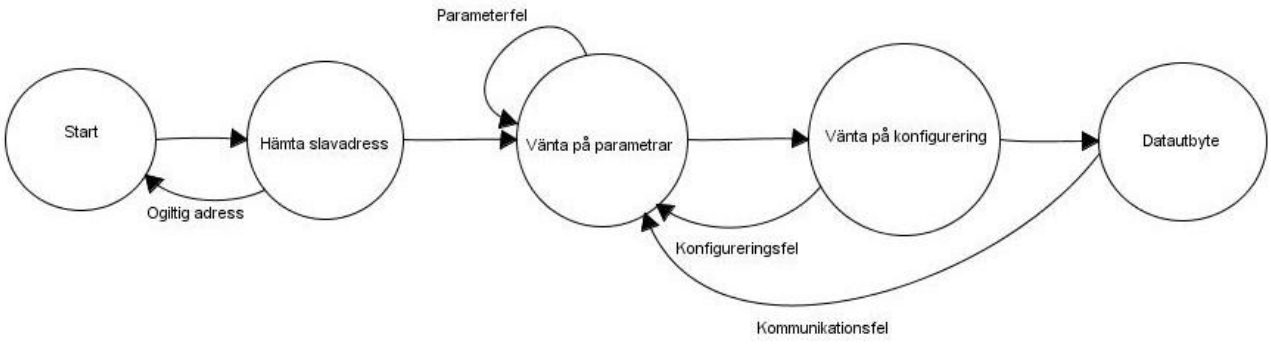
Figur 9. *Identifiering av telegramtyp, kontroll av adress, checksumma samt parametrar*.

I `profibus_RX`-funktionen kontrolleras den första byten i datapaketet, där definieras det vilken typ av telegram det är (se avsnitt 2.2.3). Sedan kontrolleras det om det om telegrammet har kommit till rätt adress. Detta kontrolleras på så sätt att DA byten jämförs mot slavens adress. Om telegrammets adress stämmer överens så kontrolleras

checksumman. Dessa saker kan endast kontrolleras efter att man har identifierat telegramtypen, för deras information är placerade i olika bytes i de olika telegrammen. Om någon av kontrollerna går fel så avbryts behandlingen av datapaketet och programmet återgår till att vänta på data. När detta är avklarat så börjar programmet behandla DSAP (Destination Service Access Point).

I DSAP behandlas olika kommandon som kontrollerar att slaven är i skick och sedan returnerar ett svar till mastern. I kommandot SET_SLAVE_ADR ställer man in slavadressen och returnerar den till mastern. SAP_GLOBAL_CONTROL är ett kontrollkommando för slaven som kontrollerar om ”clear_data” flaggan är hög, i så fall tänds en varnings-LED.

Från figur 14. ser man en sammanställning av hur ett Profibus-telegram hanteras då det tas emot av slaven. Här ser man hur hantereringen börjar från startbyten och sedan fortsätter till adressbyten där adressen kontrolleras att den stämmer överens med slavens adress, om den är felaktig så skickas hanteringen tillbaka till start. Om adressen är korrekt så går hanteringen vidare till att vänta på parametrarna. När de är kontrollerade och korrekta så går programmet vidare till att vänta på konfigureringsinformation från mastern. När konfigureringen av slaven är färdig så går hanteringen vidare till datautbyte, där t.ex. mätvärden skickas tillbaka till mästaren, om man använder slaven för att mäta t.ex. temperatur eller vad som annat kan tänkas mätas.



Figur 10. *Profibus-kommunikation.*

4. Resultat

Resultatet från detta examensarbete är ett tilläggskort till mikrokontrollerkorten som skall användas som stöd i utbildningen inom datakommunikation och mikroprocessortillämpningar. En prototyp av kortet har tillverkats och testats. Denna prototyp fungerade att köra RS485-kommunikation med. Den blev testad genom att man anslöt den till en dators usb-port via en RS232-omvandlare, och kopplad ihop med en annan usb-port via en RS232-omvandlare och en RS485-omvandlare, sedan kunde man kommunicera via Hyperterminal som är ett simpelt program för att kommunicera i Windows.

Koden som skall användas har blivit portad från en kod som är menad åt en annan mikrokontroller. Denna kod har kompilerats och simulerats i programmet AVR-studio.

5. Diskussion

Detta examensarbete har varit intressant och lärorikt. Jag har lärt mig mera om att designa kretskort och undersöka hur de olika komponenter fungerar. Jag hade först tänkt att designa hela signalomvandlaren från noll, men så insåg jag att detta kommer att vara mycket tidskrävande, och då man ändå kan köpa den som en integrerad krets rakt från butikshyllan för en mycket liten slant, så såg jag ingen lönsamhet i att göra det helt själv, detta för att kretskortet skulle bli mycket stort då, man vill ju att allting skall vara så litet som möjligt nuförtiden, och dels för att det skulle kunna skapa mycket mera felsökningar och annat huvudbry om det inte skulle fungera som man tänkt sig.

Själva programmeringsdelen med att porta koden till en specifik mikrokontroller var även intressant, då den ursprungliga koden var skriven av en tysk programmerare och var ämnad för en helt annan typ av mikroprocessor än vad som används i mikrokontrollern vi använder på skolan. Där fick man börja med att reda ut alla timers och in- och utgångar och även vissa andra parametrar. Inte nog med det så var även all kommentering av koden på tyska, så det blev en tyskalektion för mig själv. Programkoden är bara en grund för Profibus DP-kommunikationen med mikrokontrollern, den kan vidareutvecklas nästan hur långt som helst ännu, så att man t.ex. kan använda mikrokontrollern för att läsa av ett värde på beställning och sedan skicka svaret till en PLC.

Det skulle även ha varit intressant att pröva koppla ihop mikrokontrollern med en PLC för att använda den som slav i ett Profibus DP-nätverk. Man skulle t.ex. ha kunnat sända ut från PLC:n att den skall tända en lampa eller bara skriva ut någonting på skärmen. På grund av tidsbrist så hann jag aldrig koppla upp en fältbuss med en PLC.

5. Källförteckning

1. *Atmega644P Datablad* www.atmel.com/Images/doc8011.pdf
(hämtat 11.04.2012)

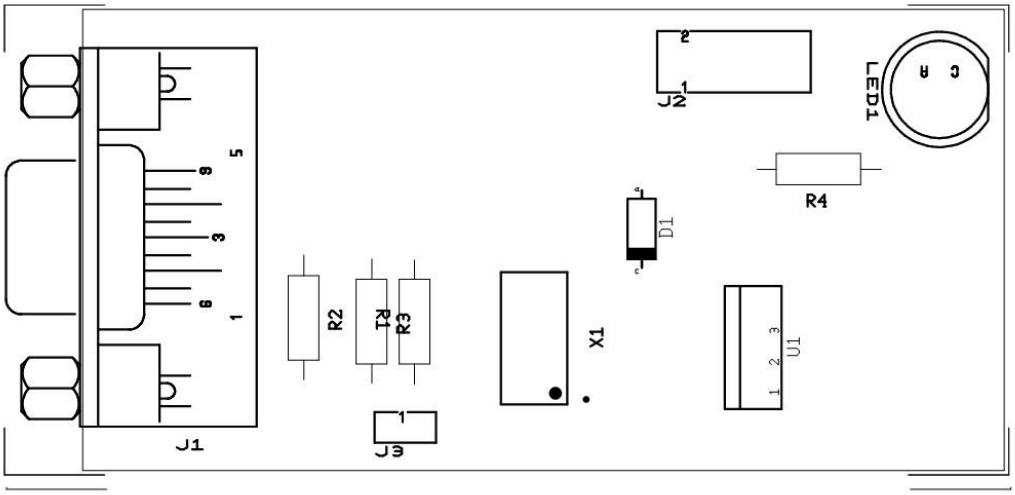
2. *SN75176 Datablad* www.ti.com/lit/ds/symlink/sn75176b.pdf
(hämtat 11.04.2012)

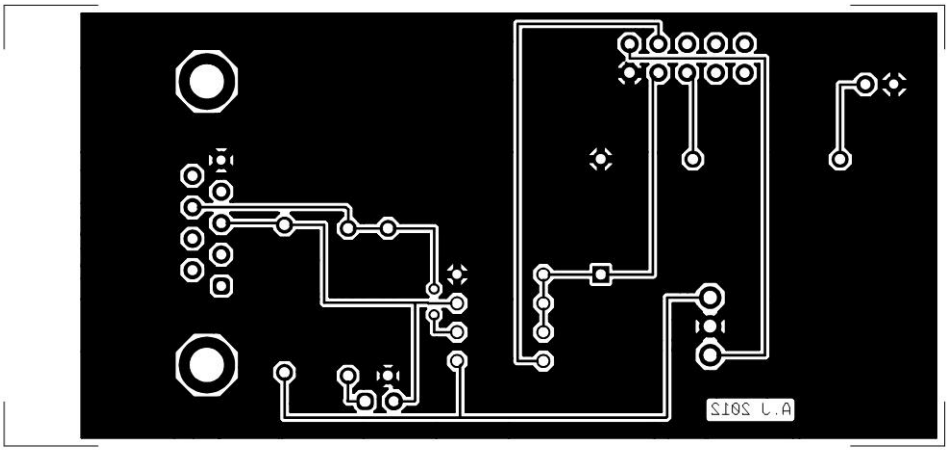
3. **Weigmann J & Kilian G** (2003) *Decentralization with PROFIBUS DP/DPV1*
Publicis Corporate Publishing
ISBN 3-89578-218-1

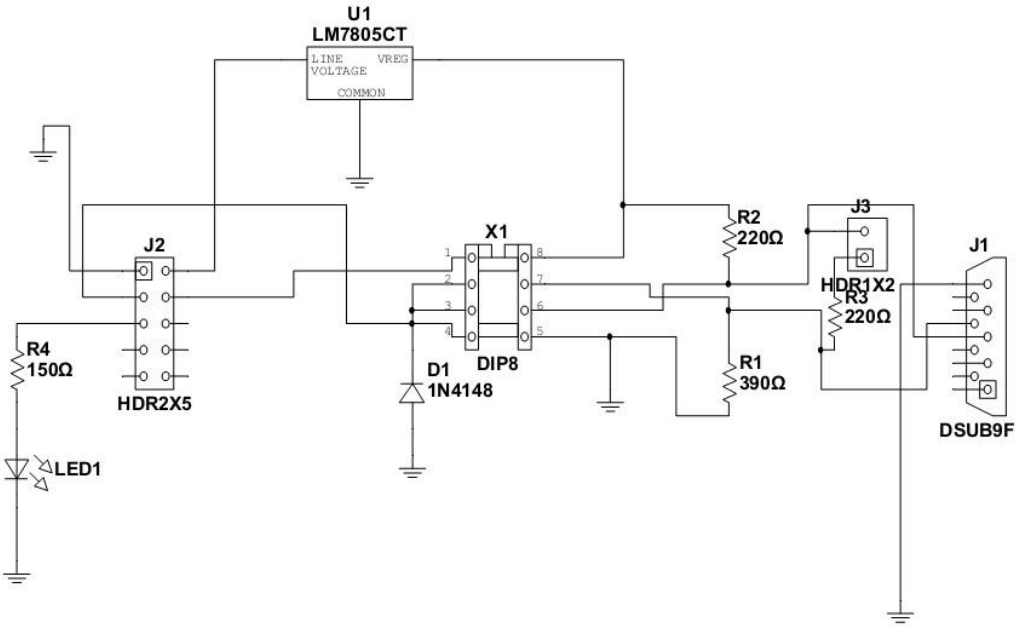
4. **Mayer K** (2001) *Datakommunikation i praktiken*
Pagina Förlags AB
ISBN 91-636-0742-5

5. *Profibus* http://en.wikipedia.org/wiki/Profibus
(hämtat: 11.04.2012)

6. **Björklund D** (2011) *Kompendium i datakommunikation*
Yrkeshögskolan Novia


J1
J2
J3
LED1
R2
R3
R4
D1
X1
U1


A.J 2012


U1
LM7805CT
LINE VOLTAGE
VREG
COMMON
J2
HDR2X5
X1
DIP8
D1
1N4148
R2
220Ω
R3
220Ω
R1
390Ω
R4
150Ω
LED1
J3
HDR1X2
J1
DSUB9F

Materiallista		
Namn	Komponent	Antal
R1	Motstånd 390Ω	1
R2, R3	Motstånd 220Ω	2
R4	Motstånd 150Ω	1
D1	Diod 1N4148	1
J1	DB9 Kontakt	1
J2	2x5 Header kontakt	1
J3	1x2 Header kontakt	1
LED1	Lysdiod Röd	1
U1	LM7805	1
X1	SN75168	1

```
/*!
 * \file    profibus.c
 * \brief   Ablaufsteuerung Profibus DP-Slave Kommunikation
 * \author  © Joerg S.
 * \date    9.2007 (Erstellung) 9.2009 (Aktueller Stand)
 * \note    Verwendung nur fuer private Zwecke / Only for non-commercial use
 * \note    Info: http://www.mikrocontroller.net/topic/106174
 * \date      2010.04.28 Porting to AVR (WinAVR), Einar S.
 * \note    Pins used: USART RX / TX, Portd:2 controls TX enable.
 * \note    Crystal: see header file.
 * \note      Ported to Atmega644P by A.Junell
 */


#include <avr/io.h>
// #include <avr/signal.h>
#include <avr/interrupt.h>

#include "profibus.h"


unsigned char uart_buffer[MAX_BUFFER_SIZE];
unsigned int  uart_byte_cnt = 0;
unsigned int  uart_transmit_cnt = 0;


// Profibus Flags und Variablen
unsigned char profibus_status;
unsigned char diagnose_status;
unsigned char slave_addr;
unsigned char master_addr;
unsigned char group;
volatile unsigned char new_data;

unsigned int delay_tbit= DELAY_TBIT;
unsigned int delay_syn = TIMEOUT_MAX_SYN_TIME;
unsigned int delay_rx = TIMEOUT_MAX_RX_TIME;
unsigned int delay_sdr = TIMEOUT_MAX_SDR_TIME;

#if (OUTPUT_DATA_SIZE > 0)
volatile unsigned char data_out_register[OUTPUT_DATA_SIZE];
#endif
#if (INPUT_DATA_SIZE > 0)
unsigned char data_in_register [INPUT_DATA_SIZE];
#endif
#if (VENDOR_DATA_SIZE > 0)
unsigned char Vendor_Data[VENDOR_DATA_SIZE];
#endif
#if (EXT_DIAG_DATA_SIZE > 0)
unsigned char Diag_Data[EXT_DIAG_DATA_SIZE];
#endif
unsigned char Input_Data_size;
unsigned char Output_Data_size;
//unsigned char Input_Data_size_Module[INPUT_MODULE_CNT];
//unsigned char Output_Data_size_Module[OUTPUT_MODULE_CNT];
unsigned char Vendor_Data_size;   // Anzahl eingelesene Herstellerspezifische Bytes

#define TX_IRQ

////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Initialize UART0
 * Even parity, 1 stop bit.
 */

void init_UART0 (void)
{
             UCSR0B = 0x00;                                                    // disable
while setting baud rate
             UCSR0A = 0x00;
             UCSR0C = 0x00;
             UCSR0A = _BV(U2X0);
             UBRR0H = 0;                                                       // set
baud rate hi.
             UBRR0L = UART_BAUD_187500; // set baud rate lo.
             UCSR0C = _BV(UMSEL00) | (1<<UPM00) | (1<<UCSZ01) | (1<<UCSZ00);     // 8-bit
characters
```

```
              DDRD |= (1<<PIN2);                        // RS485 TX enable is connected to
PD2.
}

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Initialize Profibus Timer and Variables
 */
void init_Profibus (void)
{
  unsigned char cnt;

  // Variablen initialisieren
  profibus_status = PROFIBUS_WAIT_SYN;   // Wait at least Tsyn until allowing RXdata
  diagnose_status = FALSE;
  Input_Data_size = 0;
  Output_Data_size = 0;
  Vendor_Data_size = 0;
  group = 0;

//  slave_addr = get_Address();
  slave_addr = 11;  // <<< Temporary address assignment.
  // TODO: Read address from EEPROM or switches.
  // Illegal addresses are forced to DEFAULT (126). Set Address can be used to change it.
  if((slave_addr == 0) || (slave_addr > 126))
    slave_addr = DEFAULT_ADD;

  // Clear data
  #if (OUTPUT_DATA_SIZE > 0)
  for (cnt = 0; cnt < OUTPUT_DATA_SIZE; cnt++)
  {
    data_out_register[cnt] = 0xFF;
  }
  #endif
  #if (INPUT_DATA_SIZE > 0)
  for (cnt = 0; cnt < INPUT_DATA_SIZE; cnt++)
  {
    data_in_register[cnt] = 0x00;
  }
  #endif
  #if (VENDOR_DATA_SIZE > 0)
  for (cnt = 0; cnt < VENDOR_DATA_SIZE; cnt++)
  {
    Vendor_Data[cnt] = 0x00;
  }
  #endif
  #if (DIAG_DATA_SIZE > 0)
  for (cnt = 0; cnt < DIAG_DATA_SIZE; cnt++)
  {
    Diag_Data[cnt] = 0x00;
  }
  #endif
  new_data=0;
  // AVR Timer init
  RS485_RX_EN;
  UCSR0B |= (1<<RXCIE0);
  DDRD |= 0xf0; // Debug outputs.
  PORTD |= 0x80;
  OCR1A = TIMEOUT_MAX_SYN_TIME;
  TIMER1_RUN; // Start counting.
  TIMER2_RUN;
//  TIMSK |= 0x04;          // Enable interrupt on Timer1 overflow.
  sei();
}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief ISR UART0 Receive
 */
SIGNAL(SIG_UART_RECV)
{

  // Retrieve RXdata to buffer
  uart_buffer[uart_byte_cnt] = UDR0;

  // Only read data after Tsyn have expired
```

```
  if (profibus_status == PROFIBUS_WAIT_DATA)
  {
    profibus_status = PROFIBUS_GET_DATA;
  }

  // Read data allowed?
  if (profibus_status == PROFIBUS_GET_DATA)
  {
    uart_byte_cnt++;
    // Check for buffer overflow!
    if(uart_byte_cnt == MAX_BUFFER_SIZE) uart_byte_cnt--;
  }
  TCNT1 = 0;
}
////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Profibus auswertung
 * Called by TIMER0 ISR. It happens appx. TIMEOUT_MAX_SDR_TIME after last valid incoming
character.
 */
void profibus_RX (void)
{
  unsigned char cnt;
  unsigned char telegramm_type;
  unsigned char process_data;

  // Profibus Datentypen
  unsigned char destination_add;
  unsigned char source_add;
  unsigned char function_code;
  unsigned char FCS_data;   // Frame Check Sequence
  unsigned char PDU_size;   // PDU Groesse
  unsigned char DSAP_data;  // SAP Destination
  unsigned char SSAP_data;  // SAP Source


  process_data = FALSE;


  telegramm_type = uart_buffer[0];


  switch (telegramm_type)
  {
    case SD1: // Telegramm ohne Daten, max. 6 Byte

        if (uart_byte_cnt != 6) break;

        destination_add = uart_buffer[1];
        source_add      = uart_buffer[2];
        function_code   = uart_buffer[3];
        FCS_data        = uart_buffer[4];

        if (addmatch(destination_add)       == FALSE) break;
        if (checksum(&uart_buffer[1], 3) != FCS_data) break;

        process_data = TRUE;

        break;

    case SD2: // Telegramm mit variabler Datenlaenge

        if (uart_byte_cnt != uart_buffer[1] + 6) break;

        PDU_size        = uart_buffer[1];
        destination_add = uart_buffer[4];
        source_add      = uart_buffer[5];
        function_code   = uart_buffer[6];
        FCS_data        = uart_buffer[PDU_size + 4];

        if (addmatch(destination_add)              == FALSE) break;
        if (checksum(&uart_buffer[4], PDU_size) != FCS_data) break;

        process_data = TRUE;

        break;

    case SD3: // Telegramm mit 5 Byte Daten, max. 11 Byte
```

```
      if (uart_byte_cnt != 11) break;

      PDU_size        = 8;              // DA+SA+FC+Nutzdaten
      destination_add = uart_buffer[1];
      source_add      = uart_buffer[2];
      function_code   = uart_buffer[3];
      FCS_data        = uart_buffer[9];

      if (addmatch(destination_add)       == FALSE) break;
      if (checksum(&uart_buffer[1], 8) != FCS_data) break;

      process_data = TRUE;

      break;

  case SD4: // Token mit 3 Byte Daten

    //  if (uart_byte_cnt != 3) break;

      destination_add = uart_buffer[1];
      source_add      = uart_buffer[2];

      if (addmatch(destination_add)       == TRUE) break;

      break;

  default:

      break;

} // Switch Ende



// Nur auswerten wenn Daten OK sind
if (process_data == TRUE)
{
  master_addr = source_add; // Master Adresse ist Source Adresse


  // Service Access Point erkannt?
  if ((destination_add & 0x80) && (source_add & 0x80))
  {
    DSAP_data = uart_buffer[7];
    SSAP_data = uart_buffer[8];


    // Ablauf Reboot:
    // 1) SSAP 62 -> DSAP 60 (Get Diagnostics Request)
    // 2) SSAP 62 -> DSAP 61 (Set Parameters Request)
    // 3) SSAP 62 -> DSAP 62 (Check Config Request)
    // 4) SSAP 62 -> DSAP 60 (Get Diagnostics Request)
    // 5) Data Exchange Request (normaler Zyklus)

    switch (DSAP_data)
    {
      case SAP_SET_SLAVE_ADR: // Set Slave Address (SSAP 62 -> DSAP 55)

          // Siehe Felser 8/2009 Kap. 4.2

          // Nur im Zustand "Wait Parameter" (WPRM) moeglich

          slave_addr = uart_buffer[9];
          //IDENT_HIGH_BYTE = uart_buffer[10];
          //IDENT_LOW_BYTE = uart_buffer[11];
          //if (uart_buffer[12] & 0x01) adress_aenderung_sperren = TRUE;

          profibus_send_CMD(SC, 0, SAP_OFFSET, &uart_buffer[0], 0);

          break;

      case SAP_GLOBAL_CONTROL: // Global Control Request (SSAP 62 -> DSAP 58)

          // Siehe Felser 8/2009 Kap. 4.6.2

          // Wenn "Clear Data" high, dann SPS CPU auf "Stop"
          if (uart_buffer[9] & CLEAR_DATA_)
          {
            LED_ERROR_AN;  // Status "SPS nicht bereit"
          }
```

```
            else
            {
              LED_ERROR_AUS; // Status "SPS OK"
            }

            // Gruppe berechnen
            for (cnt = 0;  uart_buffer[10] != 0; cnt++) uart_buffer[10]>>=1;

            // Wenn Befehl fuer uns ist
            if (cnt == group)
            {
              if (uart_buffer[9] & UNFREEZE_)
              {
                // FREEZE Zustand loeschen
              }
              else if (uart_buffer[9] & UNSYNC_)
              {
                // SYNC Zustand loeschen
              }
              else if (uart_buffer[9] & FREEZE_)
              {
                // Eingaenge nicht mehr neu einlesen
              }
              else if (uart_buffer[9] & SYNC_)
              {
                // Ausgaenge nur bei SYNC Befehl setzen
              }
            }

            break;

        case SAP_SLAVE_DIAGNOSIS: // Get Diagnostics Request (SSAP 62 -> DSAP 60)

            // Siehe Felser 8/2009 Kap. 4.5.2

            // Nach dem Erhalt der Diagnose wechselt der DP-Slave vom Zustand
            // "Power on Reset" (POR) in den Zustand "Wait Parameter" (WPRM)

            // Am Ende der Initialisierung (Zustand "Data Exchange" (DXCHG))
            // sendet der Master ein zweites mal ein Diagnostics Request um die
            // korrekte Konfiguration zu pruefen

            if (function_code == (REQUEST_ + FCB_ + SRD_HIGH))
            {
              // Erste Diagnose Anfrage

              //uart_buffer[4]  = master_addr;                  // Ziel Master (mit SAP
Offset)
              //uart_buffer[5]  = slave_addr + SAP_OFFSET;      // Quelle Slave (mit SAP
Offset)
              //uart_buffer[6]  = SLAVE_DATA;
              uart_buffer[7]  = SSAP_data;                    // Ziel SAP Master
              uart_buffer[8]  = DSAP_data;                    // Quelle SAP Slave
              uart_buffer[9]  = STATION_NOT_READY_;           // Status 1
              uart_buffer[10] = STATUS_2_DEFAULT + PRM_REQ_;  // Status 2
              uart_buffer[11] = DIAG_SIZE_OK;                 // Status 3
              uart_buffer[12] = MASTER_ADD_DEFAULT;           // Adresse Master
              uart_buffer[13] = IDENT_HIGH_BYTE;              // Ident high
              uart_buffer[14] = IDENT_LOW_BYTE;               // Ident low
              #if (EXT_DIAG_DATA_SIZE > 0)
              uart_buffer[15] = EXT_DIAG_DATA_SIZE;           // Geraetebezogene Diagnose
(Anzahl Bytes)
              for (cnt = 0; cnt < EXT_DIAG_DATA_SIZE; cnt++)
              {
                uart_buffer[16+cnt] = Diag_Data[cnt];
              }
              #endif

              profibus_send_CMD(SD2, DATA_LOW, SAP_OFFSET, &uart_buffer[7], 8 +
EXT_DIAG_DATA_SIZE);
            }
            else if (function_code == (REQUEST_ + FCV_ + SRD_HIGH) ||
                     function_code == (REQUEST_ + FCV_ + FCB_ + SRD_HIGH))
            {
              // Diagnose Anfrage um Daten von Check Config Request zu pruefen

              //uart_buffer[4]  = master_addr;                  // Ziel Master (mit SAP
Offset)
              //uart_buffer[5]  = slave_addr + SAP_OFFSET;      // Quelle Slave (mit SAP
Offset)
```

```
              //uart_buffer[6]  = SLAVE_DATA;
              uart_buffer[7]  = SSAP_data;                    // Ziel SAP Master
              uart_buffer[8]  = DSAP_data;                    // Quelle SAP Slave
              if (diagnose_status == TRUE)
                uart_buffer[9]  = EXT_DIAG_;                  // Status 1
              else
                uart_buffer[9]  = 0x00;
              uart_buffer[10] = STATUS_2_DEFAULT;             // Status 2
              uart_buffer[11] = DIAG_SIZE_OK;                 // Status 3
              uart_buffer[12] = master_addr - SAP_OFFSET;     // Adresse Master
              uart_buffer[13] = IDENT_HIGH_BYTE;              // Ident high
              uart_buffer[14] = IDENT_LOW_BYTE;               // Ident low
              #if (EXT_DIAG_DATA_SIZE > 0)
              uart_buffer[15] = EXT_DIAG_DATA_SIZE;           // Geraetebezogene Diagnose
(Anzahl Bytes)
              for (cnt = 0; cnt < EXT_DIAG_DATA_SIZE; cnt++)
              {
                uart_buffer[16+cnt] = Diag_Data[cnt];
              }
              #endif

              profibus_send_CMD(SD2, DATA_LOW, SAP_OFFSET, &uart_buffer[7], 8 +
EXT_DIAG_DATA_SIZE);
            }

            break;

        case SAP_SET_PRM: // Set Parameters Request (SSAP 62 -> DSAP 61)

            // Siehe Felser 8/2009 Kap. 4.3.1

            // Nach dem Erhalt der Parameter wechselt der DP-Slave vom Zustand
            // "Wait Parameter" (WPRM) in den Zustand "Wait Configuration" (WCFG)

            // Nur Daten fuer unser Geraet akzeptieren
            if ((uart_buffer[13] == IDENT_HIGH_BYTE) && (uart_buffer[14] ==
IDENT_LOW_BYTE))
            {
              //uart_buffer[9]  = Befehl
              //uart_buffer[10] = Watchdog 1
              //uart_buffer[11] = Watchdog 2
              //uart_buffer[12] = Min TSDR
              //uart_buffer[13] = Ident H
              //uart_buffer[14] = Ident L
              //uart_buffer[15] = Gruppe
              //uart_buffer[16] = User Parameter

              // Bei DPV1 Unterstuetzung:
              //uart_buffer[16] = DPV1 Status 1
              //uart_buffer[17] = DPV1 Status 2
              //uart_buffer[18] = DPV1 Status 3
              //uart_buffer[19] = User Parameter

              // User Parameter groeße = Laenge - DA, SA, FC, DSAP, SSAP, 7 Parameter Bytes
              Vendor_Data_size = PDU_size - 12;

              // User Parameter einlesen
              #if (VENDOR_DATA_SIZE > 0)
              for (cnt = 0; cnt < Vendor_Data_size; cnt++) Vendor_Data[cnt] =
uart_buffer[16+cnt];
              #endif

              // Gruppe einlesen
              for (group = 0; uart_buffer[15] != 0; group++) uart_buffer[15]>>=1;

              profibus_send_CMD(SC, 0, SAP_OFFSET, &uart_buffer[0], 0);
            }

            break;

        case SAP_CHK_CFG: // Check Config Request (SSAP 62 -> DSAP 62)

            // Siehe Felser 8/2009 Kap. 4.4.1
                                      // It did only consider the last module. Changed
"=" to "+=" May 04 2010 E.S.

            // Nach dem Erhalt der Konfiguration wechselt der DP-Slave vom Zustand
            // "Wait Configuration" (WCFG) in den Zustand "Data Exchange" (DXCHG)

            // IO Konfiguration:
```

```
            // Kompaktes Format fuer max. 16/32 Byte IO
            // Spezielles Format fuer max. 64/132 Byte IO

            // Je nach PDU Datengroesse mehrere Bytes auswerten
            // LE/LEr - (DA+SA+FC+DSAP+SSAP) = Anzahl Config Bytes
                                        // FIXIT: module 1 Byte input + 16 Byte input
yields 18 Byte returned to master!
                                        Output_Data_size=0;
                                        Input_Data_size=0;
            for (cnt = 0; cnt < uart_buffer[1] - 5; cnt++)
            {
              switch (uart_buffer[9+cnt] & CFG_DIRECTION_)
              {
                case CFG_INPUT:

                    Input_Data_size += (uart_buffer[9+cnt] & CFG_BYTE_CNT_) + 1;
                    if (uart_buffer[9+cnt] & CFG_WIDTH_ & CFG_WORD)
                      Input_Data_size += Input_Data_size*2;

                    break;

                case CFG_OUTPUT:

                    Output_Data_size += (uart_buffer[9+cnt] & CFG_BYTE_CNT_) + 1;
                    if (uart_buffer[9+cnt] & CFG_WIDTH_ & CFG_WORD)
                      Output_Data_size += Output_Data_size*2;

                    break;

                case CFG_INPUT_OUTPUT:

                    Input_Data_size += (uart_buffer[9+cnt] & CFG_BYTE_CNT_) + 1;
                    Output_Data_size += (uart_buffer[9+cnt] & CFG_BYTE_CNT_) + 1;
                    if (uart_buffer[9+cnt] & CFG_WIDTH_ & CFG_WORD)
                    {
                      Input_Data_size += Input_Data_size*2;
                      Output_Data_size += Output_Data_size*2;
                    }

                    break;

                case CFG_SPECIAL:

                    // Spezielles Format

                    // Herstellerspezifische Bytes vorhanden?
                    if (uart_buffer[9+cnt] & CFG_SP_VENDOR_CNT_)
                    {
                      // Anzahl Herstellerdaten sichern
                      Vendor_Data_size = uart_buffer[9+cnt] & CFG_SP_VENDOR_CNT_;

                      // Anzahl von Gesamtanzahl abziehen
                      uart_buffer[1] -= Vendor_Data_size;
                    }

                    // I/O Daten
                    switch (uart_buffer[9+cnt] & CFG_SP_DIRECTION_)
                    {
                      case CFG_SP_VOID:
                          // Leeres Datenfeld
                          break;

                      case CFG_SP_INPUT:

                          Input_Data_size += (uart_buffer[10+cnt] & CFG_SP_BYTE_CNT_) + 1;
                          if (uart_buffer[10+cnt] & CFG_WIDTH_ & CFG_WORD)
                            Input_Data_size += Input_Data_size*2;

                          cnt++;  // Dieses Byte haben wir jetzt schon

                          break;

                      case CFG_SP_OUTPUT:

                          Output_Data_size += (uart_buffer[10+cnt] & CFG_SP_BYTE_CNT_) + 1;
                          if (uart_buffer[10+cnt] & CFG_WIDTH_ & CFG_WORD)
                            Output_Data_size += Output_Data_size*2;

                          cnt++;  // Dieses Byte haben wir jetzt schon
```

```
                    break;

                case CFG_SP_INPUT_OPTPUT:

                    // Erst Ausgang...
                    Output_Data_size += (uart_buffer[10+cnt] & CFG_SP_BYTE_CNT_) + 1;
                    if (uart_buffer[10+cnt] & CFG_WIDTH_ & CFG_WORD)
                      Output_Data_size += Output_Data_size*2;

                    // Dann Eingang
                    Input_Data_size += (uart_buffer[11+cnt] & CFG_SP_BYTE_CNT_) + 1;
                    if (uart_buffer[11+cnt] & CFG_WIDTH_ & CFG_WORD)
                      Input_Data_size += Input_Data_size*2;

                    cnt += 2;  // Diese Bytes haben wir jetzt schon

                    break;

                } // Switch Ende

                break;

            default:

                Input_Data_size = 0;
                Output_Data_size = 0;

                break;

          } // Switch Ende
        } // For Ende

        if (Vendor_Data_size != 0)
        {
          // Auswerten
        }


        // Bei Fehler -> CFG_FAULT_ in Diagnose senden

        // Kurzquittung
        profibus_send_CMD(SC, 0, SAP_OFFSET, &uart_buffer[0], 0);

        break;

    default:

        // Unbekannter SAP

        break;

  } // Switch DSAP_data Ende

}
// Ziel: Slave Adresse
else if (destination_add == slave_addr)
{

  // Status Abfrage
  if (function_code == (REQUEST_ + FDL_STATUS))
  {
    profibus_send_CMD(SD1, FDL_STATUS_OK, 0, &uart_buffer[0], 0);
  }
  // Master sendet Ausgangsdaten und verlangt Eingangsdaten (Send and Request Data)
  else if (function_code == (REQUEST_ + FCV_ + SRD_HIGH) ||
           function_code == (REQUEST_ + FCV_ + FCB_ + SRD_HIGH))
  {

    // Daten von Master einlesen
    #if (INPUT_DATA_SIZE > 0)
    for (cnt = 0; cnt < INPUT_DATA_SIZE; cnt++)
    {
      data_in_register[cnt] = uart_buffer[cnt + 7];
                        new_data=1;
    }
    #endif


    // Daten fuer Master in Buffer schreiben
    #if (OUTPUT_DATA_SIZE > 0)
```

```
        for (cnt = 0; cnt < OUTPUT_DATA_SIZE; cnt++)
        {
          uart_buffer[cnt + 7] = data_out_register[cnt];
        }
        #endif


        #if (OUTPUT_DATA_SIZE > 0)
        if (diagnose_status == TRUE)
          profibus_send_CMD(SD2, DIAGNOSE, 0, &uart_buffer[7], 0);  // Diagnose anfordern
        else
          profibus_send_CMD(SD2, DATA_LOW, 0, &uart_buffer[7], Input_Data_size);  // Daten
senden
        #else
        if (diagnose_status == TRUE)
          profibus_send_CMD(SD1, DIAGNOSE, 0, &uart_buffer[7], 0);  // Diagnose anfordern
        else
          profibus_send_CMD(SC, 0, 0, &uart_buffer[7], 0);          // Kurzquittung
        #endif
      }
    }

  } // Daten gueltig Ende

}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Profibus Telegramm zusammenstellen und senden
 * \param type          Telegrammtyp (SD1, SD2 usw.)
 * \param function_code Function Code der uebermittelt werden soll
 * \param sap_offset    Wert des SAP-Offset
 * \param pdu           Pointer auf Datenfeld (PDU)
 * \param length_pdu    Laenge der reinen PDU ohne DA, SA oder FC
 */
void profibus_send_CMD (unsigned char type,
                        unsigned char function_code,
                        unsigned char sap_offset,
                        char *pdu,
                        unsigned char length_pdu)
{
  unsigned char length_data;


  switch(type)
  {
    case SD1:

      uart_buffer[0] = SD1;
      uart_buffer[1] = master_addr;
      uart_buffer[2] = slave_addr + sap_offset;
      uart_buffer[3] = function_code;
      uart_buffer[4] = checksum(&uart_buffer[1], 3);
      uart_buffer[5] = ED;

      length_data = 6;

      break;

    case SD2:

      uart_buffer[0] = SD2;
      uart_buffer[1] = length_pdu + 3;  // Laenge der PDU inkl. DA, SA und FC
      uart_buffer[2] = length_pdu + 3;
      uart_buffer[3] = SD2;
      uart_buffer[4] = master_addr;
      uart_buffer[5] = slave_addr + sap_offset;
      uart_buffer[6] = function_code;

      // Daten werden vor Aufruf der Funktion schon aufgefuellt

      uart_buffer[7+length_pdu] = checksum(&uart_buffer[4], length_pdu + 3);
      uart_buffer[8+length_pdu] = ED;

      length_data = length_pdu + 9;

      break;
```

```
    case SD3:

      uart_buffer[0] = SD3;
      uart_buffer[1] = master_addr;
      uart_buffer[2] = slave_addr + sap_offset;
      uart_buffer[3] = function_code;

      // Daten werden vor Aufruf der Funktion schon aufgefuellt

      uart_buffer[9] = checksum(&uart_buffer[4], 8);
      uart_buffer[10] = ED;

      length_data = 11;

      break;

    case SD4:

      uart_buffer[0] = SD4;
      uart_buffer[1] = master_addr;
      uart_buffer[2] = slave_addr + sap_offset;

      length_data = 3;

      break;

    case SC:

      uart_buffer[0] = SC;

      length_data = 1;

      break;

    default:

      break;

  }
  profibus_TX(&uart_buffer[0], length_data);

}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Telegramm senden
 * \param data Pointer auf Datenfeld
 * \param length Laenge der Daten
 */
void profibus_TX (char *data, unsigned char length)
{
  RS485_TX_EN;
              OCR1A = TIMEOUT_MAX_TX_TIME;


  profibus_status = PROFIBUS_SEND_DATA;

  uart_byte_cnt = length;         // Anzahl zu sendender Bytes
  uart_transmit_cnt = 0;          // Zahler fuer gesendete Bytes

//  IFG2 |= UCA0TXIFG;              // TX Flag setzen
//  IE2  |= UCA0TXIE;               // Enable USCI_A0 TX interrupt
  UCSR0B |= _BV(UDRIE0);    // <<< AVR Version
}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Checksumme berechnen. Einfaches addieren aller Datenbytes im Telegramm.
 * \param data Pointer auf Datenfeld
 * \param length Laenge der Daten
 * \return Checksumme
 */
unsigned char checksum(char *data, unsigned char length)
```

```
{
  unsigned char csum = 0;

  while(length--)
  {
    csum += data[length];
  }

  return csum;
}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief Zieladresse ueberpruefen. Adresse muss mit Slave Adresse oder Broadcast (inkl.
SAP Offset)
          uebereinstimmen
 * \param destination Zieladresse
 * \return TRUE wenn Zieladresse unsere ist, FALSE wenn nicht
 */
unsigned char addmatch (unsigned char destination)
{
  if ((destination != slave_addr) &&                // Slave
      (destination != slave_addr + SAP_OFFSET) &&   // SAP Slave
      (destination != BROADCAST_ADD) &&             // Broadcast
      (destination != BROADCAST_ADD + SAP_OFFSET))  // SAP Broadcast
    return FALSE;

  return TRUE;
}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief ISR UART Transmit
 */
SIGNAL(SIG_UART_DATA)
{

  // Alles gesendet?
  if (uart_transmit_cnt < uart_byte_cnt)
  {
    // TX Buffer fuellen
    UDR0 = uart_buffer[uart_transmit_cnt++];
  }
  else
  {
    // Alles gesendet, Interrupt wieder aus
//    IE2 &= ~UCA0TXIE;
             UCSR0B &=~ _BV(UDRIE0);    // AVR, nothing to transmit, disable this
interrupt
  }
  TCNT1=0;
}
////////////////////////////////////////////////////////////////////////////////////////////
////////

////////////////////////////////////////////////////////////////////////////////////////////
////////
/*!
 * \brief ISR TIMER0
 */

SIGNAL (SIG_OUTPUT_COMPARE1A)                          // Timer1 Output Compare 1A.
{
  TIMER1_STOP;             // Guard ourselves.



  switch (profibus_status)
  {
    case PROFIBUS_WAIT_SYN: // TSYN abgelaufen

        profibus_status = PROFIBUS_WAIT_DATA;
        OCR1A = TIMEOUT_MAX_SDR_TIME;
        uart_byte_cnt = 0;
```

```
        break;

    case PROFIBUS_WAIT_DATA:  // TSDR abgelaufen aber keine Daten da
        break;

    case PROFIBUS_GET_DATA:   // TSDR abgelaufen und Daten da
        profibus_status = PROFIBUS_WAIT_SYN;
                          // We have already waited TIMEOUT_MAX_RX_TIME of bus idle. So
subtract that from Tsyn.
        OCR1A = TIMEOUT_MAX_SYN_TIME-TIMEOUT_MAX_RX_TIME;
              //            uart_buffer[0] = 0xDC;

      //  profibus_RX();

        break;

    case PROFIBUS_SEND_DATA:  // Sende-Timeout abgelaufen, wieder auf Empfang gehen

        profibus_status = PROFIBUS_WAIT_SYN;
        OCR1A = TIMEOUT_MAX_SYN_TIME;
        RS485_TX_DIS;    // Auf Receive umschalten <<<

        break;

    default:
        break;

  }

//  OCR1A = 0;

//  PORTD &=~0x04;  // <<<DEBUG
  // Timer 0 Start
  TIMER1_RUN; // <<<
}
////////////////////////////////////////////////////////////////////////////////////////////
////////
void doit(void)
{
             if(new_data==1)
             {
      data_out_register[0] +=new_data;
               new_data=0;
    }
//profibus_status=PROFIBUS_WAIT_SYN;
}
////////////////////////////////////////////////////////////////////////
int main (void)
{
////////////////////////////////////
//  watchdog_aus          ();
//  init_MCU              ();
//  switch_SMCLK_XT1_8MHz ();
  init_UART0            ();
  init_Profibus         ();
//  interrupt_settings    ();
  ////////////////////////////////////
//  doit();
  while(TRUE)
  {
    doit();
    // Hier Code fuer Hauptprogramm
  }
}
```