HOVEDPROSJEKT:

Programmerbar ADC med robust lager og GPS

aka

GAUCE

FORFATTERE:

Martin Rognerud
Ole Kristian Tørresen

Dato: 18.05.2005

Sammendrag av hovedprosjekt

Tittel:	Programmerbar ADC med robust lager og GPS	Nr. : 4
		Dato : 19.05.2005
Deltaker(e):	Martin Rognerud	
	Ole Kristian Tørresen	
Veileder(e):	Odd Olav Walmann	
Oppdragsgiver:	Forsvarets forskningsinstitutt	
Kontaktperson:	Terje Angeltveit	
Stikkord	USB, GPS, ADC, CompactFlash	
Antall sider:	Antall bilag:	Tilgjengelighet (åpen/konfidensiell): Åpen

Kort beskrivelse av hovedprosjektet:

Forsvarets forskningsinstitutt har en miljølab der de tester ut sine og andres proenkter under ekstreme forhold (slag/risting, temperatur og luftfuktighet) for å se om de klarer de belastninger proenktet blir utsatt for i løpet av sin levetid. For å få mest mulig virkelighetsnære simuleringer trenger de å vite hvordan omgivelsene der proenktet skal være oppfører seg. For å finne ut dette plasseres sensorer ut i omgivelsene og dataen lagres. Etter som det er nødvendig å teste proenktet i ekstreme situasjoner er dermed omgivelsene ekstreme og dette kan være veldig fysisk krevende for de personene som er tilstede for å overvåke disse målingene. Et eksempel på ekstreme omgivelser er en MTB i høy sjø. Derfor er det ønskelig å slippe å faktisk måtte være til stede under innsamlingen av data. Her kommer vårt prosjekt inn med at det er datalogger som lagrer dataene slik at de kan leses ut når testene er over. Det er også muligens ønskelig å bruke dem i forskningsraketter og der er det ikke plass til at noen er med og tar opp dataene.


GAUCE
ADC
Programmerbar ADC med robust lager og GPS
Nios II
USB
Integrert
Programmerbar
USB
HIG
Forsvarets
forskningsinstitutt
GPS
Nios
Kompakt
ecos
Martin Rognerud og Ole Kristian Tørresen
Våren 2005
CompactFlash

Forord:

Denne rapporten er utarbeidet for FFI av Ole Kristian Tørresen og Martin Rognerud i forbindelse med et hovedprosjekt som avslutter det 3-årige studiet ved Høgskolen i Gjøvik våren 2005.

Oppgaven til dette hovedprosjektet var gitt av Forsvarets forskningsinstitutt, med Terje Angeltveit som kontaktperson. Denne rapporten er det vårt arbeid med dette hovedprosjektet munnet ut i.

Vi ønsker å rette en stor takk til disse personene:

Odd Olav Walmann som var vår veileder gjennom dette prosjektet for hans hjelp og tips..

Terje Angeltveit som var vår kontaktperson ved FFI for å svare på spørsmål og gi oss den bistanden vi trengte.
.
Lærer staben ved elektro på HIG for de tre siste årene og for deres hjelpsomhet med å besvare spørsmål vedrørende hovedprosjektet.

John Elvesveen og Arne Myre for deres hjelpsomhet med å finne fram og bestille komponenter vi trengte.

De andre studentene i 02HINED og 02HINEA for deres moralske støtte og deres evne til å distrahere oss fra å jobbe med hovedprosjektet når det ble for mye jobbing.

Gjøvik, 19.05.2005

Martin Rognerud Ole Kristian Tørresen

Innholdsfortegnelse:

Sammendrag av hovedprosjekt 2
Forord: 5
Innholdsfortegnelse: 6
1.0 Innledning 7
1.1 Organisering 7
1.2 Oppgaven 7
1.3 Bakgrunnen til studentene 7
1.4 Arbeidsformer 7
2.0 Teori 8
2.1 USB 8
Deskriptorer 9
Enumerasjon 9
Etterspørsels typer 11
HID 13
2.2 FPGA 14
2.3 Nios II 16
2.4 eCos 18
3.0 Utstyr 20
3.1 SOPC Builder 20
3.2 Quartus II 20
3.3 Nios II IDE 22
3.4 Nios II SDK Shell 22
3.5 eCos 23
3.6 Nios Development Kit, Cyclone Edition 24
3.7 Notepad2 24
3.8 Chinook Free 25
4.0 Utførelsen 26
4.1 Utviklingssettet 26
4.2 USB 27
4.3 GPS 29
4.4 ADC 29
4.5 GAUCE applikasjonen 30
4.6 PC programmet 30
Grensesnittet (GUI) 30
Flytskjema for PC programmet 33
Valg av samplings frekvenser: 39
Tegning av testkort og forslag til ferdig krets: 39
5.0 Konklusjon 40
6.0 Litteraturliste 42

1.0 Innledning

1.1 Organisering

Rapporten beskriver hva vi gjorde, hvordan vi gjorde det og hvorfor vi gjorde det vi gjorde under dette hovedprosjektet. Vi har 3 hovedkapitler: Teori, utstyr og utførelse. Det er mye programkode som har blitt generert i løpet av dette hovedprosjektet og det ligger som vedlegg. Vi har brukt mye figurer, da spesielt i et par veiledninger der vi har brukt mange skjermskudd, blant annet fordi et bilde sier mer enn tusen ord i mange sammenhenger.

1.2 Oppgaven

Oppgaven gikk ut på å lage en robust datalogger som kunne tåle å bli plassert i mange forskjellige typer framkomstmidler under ekstreme forhold. Denne dataloggeren måtte kunne programmeres på forhånd, her foreslo vi at dette kunne gjøres ved hjelp av USB og dette ble godtatt. Den innkommende dataen er analog og ligger mellom 0 og +5V og skulle samples med 12 bit og ha en maks samplingsfrekvens på 4 kHz per kanal. Det skulle være opp til 8 kanaler som var aktive samtidig. Vi skulle ikke i dette hovedprosjektet bry oss om analising filtrene som må til for å forsikre seg om at samplingsteoremet blir opprettholdt og vi ikke får speilinger fra høyere frekvenser.
CompactFlash var oppgitt fra oppdragsgiver som en mulig lagringskilde. Lageret måtte være på minimum 500MB. Oppdragsgiver ønsket også muligheten til å tilkoble GPS og at muligheten for at en GPS skulle integreres i selve enheten.
Annet enn det var det ingen spesielle krav til komponenter som måtte brukes fra oppdragsgiver. Det var heller ikke krav om at vi måtte bruke komponenter med spesielle toleranser.

1.3 Bakgrunnen til studentene

Vi har gått tre år på elektroingeniørlinja ved Høgskolen i Gjøvik, med fordypning innen microData det siste året. Der ligger fokuset på digitale kretser og digital signal behandling inni og mellom digitale komponenter. Forut for dette har den ene deltageren allmennfaglig utdanning mens den andre har utdanning innen serviceelektronikk.

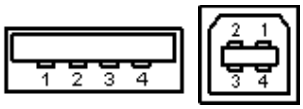
1.4 Arbeidsformer

Etter at vi hadde valgt hvilken oppgave vi skulle ha som hovedprosjekt dro vi til oppdragsgiver for å diskutere hva de ønsket. Senere kontakt med FFI ble gjort over E-post og telefon. Deretter skrev vi et forprosjekt der vi la opp i store trekk hvordan vi ville gjennomføre oppgaven. For å kunne utføre det vi la opp til å gjøre måtte vi studere en del i begynnelsen av prosjektet for å bygge opp den faglige kompetansen til å utrette det vi ville. Dette på grunn av at vi benyttet oss av teknologier slik som FPGA og USB som vi ikke tidligere har hatt noe om eller som skolen har noe kompetanse på. Vi valgte også å benytte et operativsystem, eCos, på FPGAen som vi ikke hadde noe kunnskap fra tidligere. Spesielt i forbindelse med eCos har fora på Internett vært veldig viktige for å innhente informasjon. Etter dette begynte vi å programmere FPGAen og PC-programmet og deretter testet vi ut de forskjellige komponentene sammen.

2.0 Teori

2.1 USB

Universal Seriell Buss, er en seriell bus som navnet tilsier og den er meget fleksibel. Etter som den skal være fleksibel og meget enkel for sluttbruker å bruke innebærer dette en ganske omfattende protokoll. Der det finner mange forsjellige klasser som man kan bruke for eksempel printere hubber etc. USB har tre hastigheter Lav, full etter høy hastighet. Lav hastighet (1,5 Mbps) var det første versjonen av USB som ble benyttet men dette ble etter vært økt til full hastighet (12 Mbps) og så senere økt til høy hastighet (480 Mbps) ved USB 2.0. USB grensesnittet inneholder også spenningstilførsel der en kan trekke opp til 500mA på stasjonær PC. Hvor mye strøm en ønsker å trekke fra hosten/hubben setter en opp i en deskriptor. Det er ikke alle hoster/hubber som har muligheten til å detektere hvor mye strøm en trekker så det er i noen tilfeller mulig å slippe unna med å trekke mer enn det en har oppgitt men dette er ikke å anbefale etter som apparatet ikke vil virke under mange hoster/hubber.
For å kunne koble til en USB-enhet til et system trengs både hardware og software. Dette skyldes for eksempel at maskinen skal finne ut om det trekkes mer strøm enn det som er tillatt og for å finne ut om det er lav eller høy hastighet, mer informasjon om dette i under punktet om enumerasjon.
USB-linja består av en +5 V kabel og en jord (GND) kabel pluss et tvinnet par kabler som dataen sendes over kalt D+ og D-. Dataene sendes differensielt over disse to linjene for å gjøre de mindre følsomme for støy og en oppnår også en sammenlagt 0 effekt.
USB benytter seg som regel av to typer kontakter (det finnes flere) en A og en B kontakt der A kontakta er den som sitter i hosten eller hubben mens B kontakta er på apparatet hosten skal kommunisere med. Dette er gjort slik at det ikke skal være mulig å plugge i kabelen feil vei. Under ser vi hvordan disse to kontaktene ser ut.



Til høyre ser vi type A og til venstre type B.
Der pinnene er koblet opp som en ser under:

Pinne nummer	Lednings farge	Funksjon
1	Rød	V_{BUS} (5 volts)
2	Hvite	D-
3	Grønn	D+
4	Svart	GND

Tabell 1 standar farger på USB

Krav til nøyaktigheten på klokke frekvensen som styrer USB-trafikken er selvsagt økende etter hvilke overførings hastighet en benytter. Under er kravene som er satt opp i USB standarden:
Høy hastighet (480.00Mb/s) har en toleranse på ±0,5‰ eller ± 500ppm.
Full hastighet (12.000Mb/s) har en toleranse på ±0,25% eller 2,500ppm.
Lav hastighet (1.50Mb/s) har en toleranse på ±1,5% eller 15,000ppm.

Deskriptorer

Deskriptorer er datastrukturer som inneholder informasjon om apparatet eller om deler av apparatet. Som for eksempel hvilke endpoint som brukes, hvor store pakker de tåler, type data overføring samt navn på proenkt og beskrivelser av virkemåte hvis det er ønsket.
Deskriptorene er altså formatet hosten ønsker å motta apparatets beskrivelse av seg selv og hvordan det skal kommuniseres med.
Deskriptorene er bygget opp slik at det er visse deskriptorer som alle har og ut fra disse finner vi informasjon om det er flere under liggende deskriptorer.

Her er en liste over standard definerte deskriptorer med beskrivelse om de er alltid nødvendige eller ikke.

Deskriptor type:	Nødvendig:
`Device`	Alltid
`Device_qualifier`	Bare hvis en bruker apparater som benytter seg av både full og høy hastighet
`Configuration`	Alltid
`Other_speed_configuration`	Bare hvis en bruker apparater som benytter seg av både full og høy hastighet
`Interface`	Alltid
`Endpoint`	Ikke hvis en bare bruker endpoint 0
`String`	Nei (valgfri)
`Interface_power`	Bare vis en vil støtte styring av spenninger over USBen (2.0)

Tabell 2. Spesifikke klasser trenger flere deskriptorer enn disse.

Her er en liste over deskriptorer vi har brukt:

Descriptor	Beskrivelse
`Device`	Inneholder hva slags I/O enhet det er
`configuration`	Beskriver I/O enhetens egenskaper og muligheter
`interface`	Grensesnittet mellom I/O enheten og PC hosten
`endpoint`	Endpoint nr og overføringstype
`Hid`	En klasse deskriptor, informasjon om kommunikasjon
`Report`	Formatet til datakommunikasjon

Tabell 3. Deskriptorer brukt i prosjektet.

Her ser vi eksempler på at klasser har egne deskriptorer, for som en kan se bruker vi en HID deskriptor og en report deskriptor i tillegg til de standardiserte.

Enumerasjon

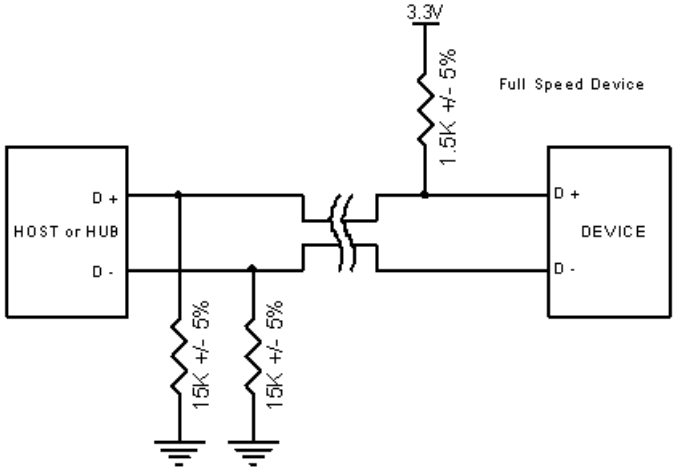
1. Enumerasjonen begynner når hubben (transceiveren) merker at det blir satt inn et nytt apparat i USB pluggen og den konfigurerer dette. Det kan også være at apparatet blir plugget inn i en hub som er koblet til hosten eller at hosten starter opp med apparatet innkoblet.

2. Hubben har 15 kΩ pull-down motstander på både D+ og D- linja mens apparatet som er koblet til har tilkoblet en 1,5KΩ pull-up motstand på en av disse (på D+ for full hastighet og på D- for lav hastighet) så den linja som er tilkoblet pull-up motstanden blir liggende høyere enn den andre og dermed vet hubben at det er kommet et apparat på porten.

3. Hosten får beskjed av hubben at noe har skjedd og spør så hubben om å sende sin statusrapport der hosten finner ut at et nytt apparat er blitt tilkoblet.

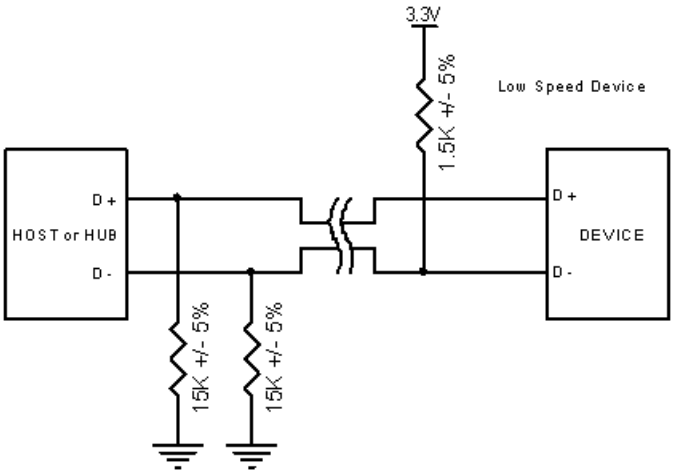
4. Hubben benytter seg av det som er beskrevet i punkt 2 til å finne ut hvilke hastighet apparatet har.
Hvis apparatet egentlig benytter høy hastighet legger det seg her på full hastighet for så å gjøre det til høy hastighet senere. Denne informasjonen sendes til hosten ved neste statusrapport.

FULL HASTIGHET:



Figur 1 Koblings oppsett for full hastighet

LAV HASTIGHET:



Figur 2 koblings oppsett for lav hastighet

5. Hubben resetter apparatet med og sette både D+ og D- lave. Dette foregår over minimum 10 millisekunder.

6. Under resettingen sender apparatet et alternerende signal hvis det skal bruke høyhastighets overføring. Dette signalet oppfatter hubben og setter overføringen til høy hastighet

7. Hosten finner ut om apparatet er ferdig med å resettes og det er da mulig å nå apparatet med adressen 00h.

8. Hosten henter device deskriptorer fra apparatet gjennom adresse 0 endpoint 0.
Dette gjøres or å finne ut maksimum størrelse på pakkene (det finnes mer informasjon men denne blir utelatt på dette tidspunkt).

9. Hosten gir en unik adresse til apparatet.

10. Så sender hosten en ny forespørsel etter diverse deskriptorer på den nye adressen og finner da ut om antall konfigurerings muligheter til apparatet og sender deretter en forespørsel etter konfigureringsdeskriptoren til apparatet for å finne ut hvor mye som ligger i denne deskriptoren og deretter sender den etter konfigureringsdeskriptoren igjen men denne gangen med alt det som følger med.

11. På grunnlag av den informasjonen hosten har fått tak i deskriptorene prøver hosten å finne en driver til apparatet.

12. Driveren konfigurerer så apparatet.

Etterspørsels typer

For å kommunisere med USB-enheten bruker hosten forskjellige Requester etter som hvilken type informasjon den trenger. Requestene sendes med en kontrolloverføring til endpoint 0 som har med all kontrolldataen.
Disse requestene brukes for eksempel i enumerasjonen for å få tak i konfigurasjonsinformasjonen til USB-enheten. Da sendes alle disse requestene automatisk av hosten og vi som programmerer trenger ikke tenke noe på det. Men hvis en skriver drivere selv kan en aktivisere visse request, for eksempel hvis USB-enheten har flere forskjellige konfigurasjoner en kan bruke kan en sende `Set_Configuration` og forandre hvilken type kommunikasjon en skal benytte med den. I dette prosjektet benytter vi HID og da trenger vi ikke tenke mye på disse på PC siden fordi Windows tar seg av den delen. Vi må bare hente inn informasjonen fra der Windows har lagret denne.
Men på mottaker siden trenger vi å tenke på dette slik at programmet vårt reagerer riktig på de forskjellige requestene som Windows sender.
Her ser vi oppbygningen til et slikt request (setup pakka):

Byte offset:	Navn:	Størrelse:	Type:	Beskrivelse:
0	bmRequestType	1	Bit-Map	**D7 Data Phase Transfer Direction** 0 = Host to Device 1 = Device to Host **D6..5 Type** 0 = Standard 1 = Class 2 = Vendor 3 = Reserved **D4..0 Recipient** 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	bRequest	1	Verdi	Forespørselen
2	wValue	2	Verdi	Videre beskrivelse av forespørselen
4	wIndex	2	Index/offset	index
6	wLength	2	lengde	Lengden av data som skal overføres hvis forespørselen trenger et svar

Figur 3 Bitmap til Request pakken

Noen standard etterspørseler:

bmRequestType	bRequest	wValue	wIndex	wLength	Data pakke
1000 0000b	GET_STATUS (0x00)	0	0	2	USB-enheten returnerer statusen
0000 0000b	CLEAR_FEATURE (0x01)	Feature Selector	0	0	Ingen
0000 0000b	SET_FEATURE (0x03)	Feature Selector	0	0	Ingen
0000 0000b	SET_ADDRESS (0x05)	Device Address	0	0	Ingen
1000 0000b	GET_DESCRIPTOR (0x06)	Descriptor Type & Index	0 eller språk ID	Deskriptor lengde	USB-enheten returnerer deskriptoren
0000 0000b	SET_DESCRIPTOR (0x07)	Descriptor Type & Index	0 eller språk ID	Deskriptor lengde	Hosten sender deskriptoren
0000 0000b	GET_CONFIGURATION (0x08)	0	0	1	USB-enheten returnerer konfigurasjonen
0000 0000b	SET_CONFIGURATION (0x09)	Den nye konfigurasjonen	0	0	Ingen

Figur 4 standar etterspørsler

CLEAR_FEATURE og SET_FEATURE kan sette mottakeren i test moens eller vekke den opp med remote_wake_up
Dette var bare en meget enkel innføring i dette temaet. Mer informasjon finner vi i kapitel 6 i ”USB Complete” av Jan Axelson.

HID

Human Interface Device er en av klassene en kan bruke under USB protokollen. Grunnen til at vi valgte å bruke denne er at den er godt dokumentert og den passer bra til typen data vi skal sende. Etter navnet Human Interface Device er denne klassen beregnet på eksterne enheter som er håndtert av mennesker, slik som mus, tastatur, joystick etc. Dette er jo ikke tilfelle hos oss men det er ingen nødvendighet at enheten har kontakt med mennesker men at data strømmen er lik den som brukes i slike typer apparater. Med dette innebærer at det sendes små mengder data til uregelmessige tidspunkter og dette går godt over ens med våre krav.
HID er som sagt en klasse under USB og derfor må vi spesifisere dette i deskriptorene og dette fører til at vi får to ekstra deskriptorer i tillegg til standard deskriptorene, dette er HID deskriptoren og rapport deskriptoren som beskriver hvilke ekstra deskriptorer som følger med HIDen og hvordan rapportene skal se ut. Bakdelen til HID er at den er treg men etter som vi bare skal bruke USBen til å overføre kommandoer og ikke til store data mengder er dette ikke noe problem. Windows har også innebygde drivere for HID så vi slipper å skrive disse som er en stor fordel. Dette gjør at vi kan finne alle funksjonene vi trenger for å snakke sammen med den ytre enheten. Disse funksjonene finnes i forsjellige dll-filer som er standard i Windows. Disse filene er Hid.dll, Setupapi.dll og Kernel32.dll.
Strukturene som brukes til å sende informasjonen fra USBen finnes også i H filer som er standardisert i Windows men etter som disse er spredt utover en menge forskjellige filer valgte vi å definere disse strukturene i egen kode. Eksempler på filer som inneholder slike strukturer samt delvis deklarerte funksjoner er: Hidpi.h, Setupapi.h og Hidsdi.h samt flere andre.

Oversikt over funksjoner som ble brukt:

API funksjoner:	DLL:	Beskrivelse:
`HidD_GetHidGuid`	hid.dll	Henter GUID, 128 bits adresse, for HID
`SetupDiGetClassDevs`	setupapi.dll	Henter adressen til en array som inneholder alle tilkoblede I/O enheter som bruker HID
`SetupDiEnumDeviceInterfaces`	setupapi.dll	Returnerer en peker til en array av grensesnitt
`SetupDiGetDeviceInterfaceDetail`	setupapi.dll	Returnerer I/O enhetens banenavn
`SetupDiDestroyDeviceInfoList`	setupapi.dll	Fjerner allokert minne fra SetupDiGetClassDevs
`CreateFile`	kernel32.dll	Åpner kommunikasjonskanal til I/O enheten
`HidD_GetAttributes`	hid.dll	Henter identifikasjon til I/O enheten
`HidD_GetPreparseData`	hid.dll	Henter peker til egenskapene for I/O enheten
`HidP_GetCaps`	hid.dll	Returnerer en array med egenskapene til I/O enhetene
`HidD_FreePreparseData`	hid.dll	Fjerner allokert minne fra HidD_GetPreparseData
`WriteFile`	kernel32.dll	Sender data til I/O enheten
`ReadFile`	kernel32.dll	Henter data fra I/O enheten
`CloseHandle`	kernel32.dll	Fjerner allokert minne fra CreateFile

Figur 5 funksjoner benyttet i programmet

For å identifisere IO-enheter som er enumerert til HID enheter benytter Windows en Global Unik IDentifikasjon. Denne kalles GUID'en og er en 128bits adresse. GUID er ikke bare brukt til å finne HID enheter, de er også brukt til en rekke ting innen Windows for å identifisere forskjellige ting Windows ønsker å skille fra hverandre.

2.2 FPGA

FPGA står for Field Programmable Gate Array, som blir noe sånt som felt-programmerbar portmatrise på norsk, altså en rekonfigurerbar logisk komponent. En FPGA krets består av en matrise med blokker, omgitt av programmerbare I/O blokker og koblet sammen med programmerbare sammenkoblinger. Med hjelp av dette kan en realisere alt fra AND-porter til

mikroprosessorer i en FPGA krets. Xilinx og Altera er de to største proensentene av FPGAer. FPGAer er tregere enn ASICer, og bruker mer strøm. Men de har visse fordelere som at de kommer på markedet fortere og at det er billigere å bruke FPGA enn ASIC hvis det skal lages mindre enn 10000 enheter. De fleste FPGAer i dag kan rekonfigureres ved den spenningen FPGAen kjøres på uten bruk av høy spenning eller annet. FPGAer brukes ofte i DSP sammenheng, enten ved at DSPen implementeres på FPGAen eller at FPGAen brukes som grensesnitt mot I/O som USB, RS232, PCI og annet. Innen forskjellige former for prototyping brukes det FPGAer, som for eksempel en ASIC. Da skrives det en hardwarebeskrivelse i et HDL språk som Verilog eller VHDL og denne programmeres så ut på en FPGA. Da kan en teste om hardwarebeskrivelsen er riktig ved å teste på FPGAen. FPGAer brukes mer og mer innen områder som stemmegjenkjenning, kryptografi, maskinsyn bioinformatikk og medisin.

2.3 Nios II

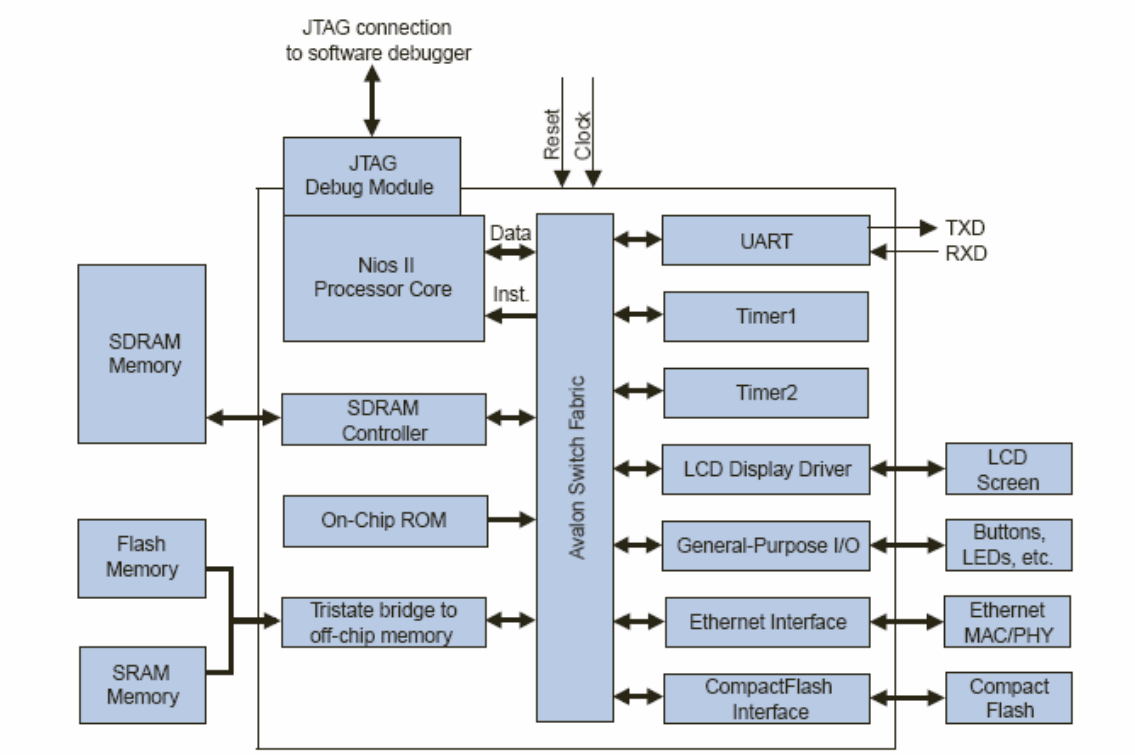
Nios II er en såkalt embedded processor, direkte oversatt til ”innebygd prosessor”, som leveres av Altera. Nesten alle utviklingssettene til Altera kommer med programvare for å utvikle Nios II systemer. Nios-prosessorarkitekturen er en 32 bits prosessorarkitektur med 32 bit databuss og 32 bit adressebuss. Den bygger på en RISC arkitektur i likhet med PowerPC fra IBM og ARM[1] fra ARM Holdings og i motsetning til den originale x86 prosessoren fra Intel. Nios II kan ha en ytelse som er sammenlignbar med ARM9, en av de mest solgte 32 bits prosessorene.

Et Nios II prosessor system er sammenlignbart med en mikrokontroller eller en ”computer på en chip” som inneholder en kombinasjon av CPU, minne og grensesnitt mot ekstern enheter som SPI, UART, LAN, ekstern minne og lignende.

Nios II er en myk[2] innebygd prosessor. Dette betyr at prosessoren ikke er fysisk en prosessor lignende en Pentium, men at den kan programmeres inn på en FPGA i likhet med et VHDL-program. Nios II er egentlig en god del VHDL eller Verilog kode som en programmerer ut på en FPGA. Det fører til at en kan med letthet programmere flere Nios-prosessorer ut på en FPGA og få de til å styre forskjellige enheter. En måte å bruke en Nios-prosessor på er å implementere en DSP på en FPGA og bruke en Nios-prosessor til å styre denne. Da kan for eksempel Nios-prosessoeren ta imot data eksternt over en USB og sende det til DSPen. Etter at dataene er behandlet, kan Nios-prosessoren lagre de på en CompactFlash eller kanskje sende det over en PCI buss til en Pentiumprosessor.

Mange vanlige mikrokontrollere kommer med et begrenset antall grensesnitt mot for eksempel UART. Med UART kan en bare koble en enhet på en kontakt, som for eksempel en mus på en PC. Den samme kontakten kan ikke deles av flere enheter. Hvis en vil koble flere enheter med UART til en og samme mikrokontroller, som en GPS, Bluetooth-moenl, kompassmoenl og andre enheter, trenger man flere enn de to som kanskje kommer med en mikrokontroller. Ved bruk av Nios II systemer kan en lett via SOPC Builder programmet bare sette på så mange serielle porter en har lyst på. Da kan en for eksempel ha en port de hver av de nevnte enhetene og styre de hver for seg.

[1] ARM er den mest solgte 32 bits prosessorarkitekturen. Over 75 % av alle innebygde prosessorer er ARM. Den finnes i alt fra leketøy, harddisker og rutere til Game Boy og mobiltelefoner.
[2] På engelsk brukes ofte softcore processor om den type prosessor som Nios II er. Men være klar over at softcore også kan oversettes til mykporno på norsk slik at en ikke blir forfjamsa når en får opp mange sider med mykporno når en egentlig søker etter en prosessor. Softcore CPU får opp de riktige resultatene.



Figur 6 Eksempel på et Nios II system.

Quartus II er den programvaren Altera leverer som brukes til å designe systemer og programmer som programmeres ut på FPGAene fra Altera. Nios II pakken kommer med et program som heter SOPC Builder som er integrert i Quartus II. Når en skal sette opp et nytt Nios II system starter en et nytt prosjekt i Quartus II, starter SOPC Builder og legger til de enhetene en trenger. Dette kan være enheter som Nios II CPUen, en PIO for å styre LED, et grensesnitt mot RAM, en DMA, et par UART og kanskje andre komponenter som en enten skriver selv eller blir levert av andre selskaper. Se figur 1 for et eksempel på hva som et Nios II system kan bestå av. Det som er innenfor den hvite firkanten er det som blir realisert på en FPGA. Det som er utenfor er eksterne enheter. Etter at en har satt de riktige innstillingene er det bare å generere et system og plassere det i Quartus-prosjektet. Så man en bare plassere de riktige pinnene og så kan en programmere det ut på en FPGA. Etter å ha fått denne hardwaredelen ut på en FPGA er det bare å sette seg ned og begynne å programmere i C eller C++ ved hjelp av Nios II IDE som også følger med Nios II pakken.

Nios II IDE er basert på den åpne kildekode Eclipse som opprinnelig ble utviklet av IBM. Nios II IDE setter opp et programvareprosjekt for en og gjør at en kan konsentrere seg om programmeringen. IDEen bruker den anerkjente kompilatorkolleksjonen GCC for å kompilere programmer til Nios II. IDEen har innebygd debugger slik at en bare kan velge en debug kommando for så å debugge programmet.

2.4 eCos

I 1997 var det over 100 forskjellige kommersielle støttende innebygde operativsystemer. Innebygde operativsystemer er slikt som for eksempel styrer en mobiltelefon eller en ruter. Ingen av de over 100 operativsystemene hadde noe særlig stor markedsandel. Mange 1000 andre ble utviklet for engangsprosjekter. Markedet var veldig fragmentert og det fantes ingen standarder. Derfor var det vanskelig å finne et operativsystem for et bestemt prosjekt. Cygnus Solutions[3] bestemte seg for å utvikle et innebygdt operativsystem som kunne tilfredsstille en stor variasjon av prosjekter, fra de små til de store kompliserte. eCos var det operativsystemet de lagde. eCos står for embedded Configurable operating system, innebygd konfigurerbart operativsystem. Cygnus fant ut at det var to grunner til at selskaper lagde sine egne RTOSer[4]: De hadde ikke lyst til å betale lisens kostnader og de ville ikke at kode som de ikke hadde fullstendig kontroll over skulle kjøre på deres systemer. Siden RTOSer både er dyre å skrive og å debugge i både tid og penger og fordi å skrive et RTOS krever en fullstendig forståelse av hele systemet, gjorde at disse RTOSene ofte var dårligere enn de som fantes på markedet.

Cygnus fant da ut at det beste ville være å lage et åpen kildekode innebygd operativsystem, altså eCos. Da slapp selskaper å betale lisenskostnader og de hadde full kontroll over hva slags kode som kjørte på selskapenes systemer. De første design diskusjonene rundt eCos begynte våren 1997. Det primære målet ble å bringe fram en kosteffektiv, høykvalitets innebygd programvareløsning til markedet. Et annet viktig mål ble at eCos skulle bli laget slik at det brukte så lite ressurser som mulig. Ved å samarbeide med forskjellige halvleder selskaper klarte Cygnus å lage et RTOS som abstrakterer hardware laget og som var veldig konfigurerbart. Dette gjorde at eCos passer i mange forskjellige innebygde systemer. På grunn av at eCos er så konfigurerbart gjør at selskaper kommer seg fortere på markedet uten å måtte skrive om mye av kildekoden til eCos.

Å redusere kostnader er alltid et poeng i innebygde systemer, som i alle andre former for produksjon og utvikling. På grunn av at eCos er åpen kildekode koster det ingenting å anskaffe. Det kan bli lastet ned fra nettet og ”prøvekjørt” uten å måtte betale noe. En annen attraktiv fordel med eCos i forhold til andre innebygde og sanntidsoperativsystemer er at det ikke er noen kostnader ved bruk av det, altså ingen enhetsavgifter til programvareskaperne.

eCos er lisensiert under en litt modifisert versjon av GPL[5]. Dette vil si at en har tilgang til kildekoden for hele operativsystemet, de forskjellige verktøyene som brukes sammen med eCos er også lisensiert under den samme lisensen. Kort forklart fungerer denne lisensen slik at hvis en forandrer selve eCos kildekoden og distribuerer den i et produkt, må disse forandringene gjøres tilgjengelig under den samme lisensen som eCos. Dette gjør at eCos stadig blir forbedret av de som bruker det og at faktisk hvilken som helst interessert person kan bruke og lære av denne kildekoden.

---

[3] Cygnus var det første selskapet som tilbydde support for åpen kildekode (Free Software eller Open Source, disse begrepene brukes litt om hverandre men er ikke det samme. Ta en titt her: http://www.gnu.org/philosophy/free-software-for-freedom.html). Cygnus ble dannet i 1989.

[4] RTOS står for real-time operating system, eller sanntids operativsystem. Innebygde operativsystemer er ofte sanntidsoperativsystemer og sanntidsoperativsystemer er ofte innebygde operativsystemer, derfor brukes de to begrepene litt om hverandre. eCos er både et innebygd og et sanntids operativsystem.

[5] GNU General Public License. Se http://www.gnu.org/copyleft/gpl.html. Se http://ecos.sourceware.org/license-overview.html for eCos lisensen.

Et nøkkelord når det gjelder eCos er konfigurerbarhet. eCos er bygd opp av programvarekomponenter. Ved å velge blant disse gjenbrukbare programvarekomponentene kan en tilfredsstille de fleste krav når det gjelder ytelse og størrelse på programmet. Det er vanlig at et innebygdt operativsystem kommer med de fleste funksjonene for å støtte en rekke forskjellige konfigurasjoner. For eksempel med et enkelt program som ”Hello World”[6] vil de fleste innebygde operativsystemene støtte ting som mutexer, taskswitching og tråder, selv om de ikke trengs til et så enkelt program. Dette systemet vil da bli mye større, mer komplisert og kanskje tregere enn det som hadde vært nødvendig. Ved bruk av eCos kan en ta bort det som ikke trengs, og bare beholde det som er viktig. Et eCos-system kan være så lite som noen hundre byte og opp til flere hundre kilobyte når sånt som nettverksfunksjonalitet og webservere er innebygd.

eCos konfigureres ved hjelp av et program som heter ”eCos Configuration Tool”. Her velger man de komponentene en skal bruke i et system, ofte ut fra ferdiglagde maler. Komponentene går helt ned på kodenivå, slik at det kan være bare deler av en kildekodefil som blir med. Dette løses som regel av ifdef og lignende ord som her:

```
#ifdef ET_ELLER_ANNET
…//Gjør noe her
#else
…//Gjør noe annet her
#endif
```

En slik kildekodenivåkonfigurerbarhet gjør at det genererte systemet blir mindre, det går raskere, blir enklere å debugge og gjør at en kan bruke tregere prosessorer fordi det kan spare en god del klokkesykluser.

Innebygd Linux fantes da Cygnus bestemte seg for å lage eCos, men bare selve Linux-kjernen uten noen programmer krever 2 MB med RAM, en må opp i mellom 4 og 8 MB for å ha noe som fungerer greit. Innebygd Linux krever da for mye ressurser til å kunne brukes i de billigste systemene. Dette var en medvirkende årsak til at eCos ble lagd.

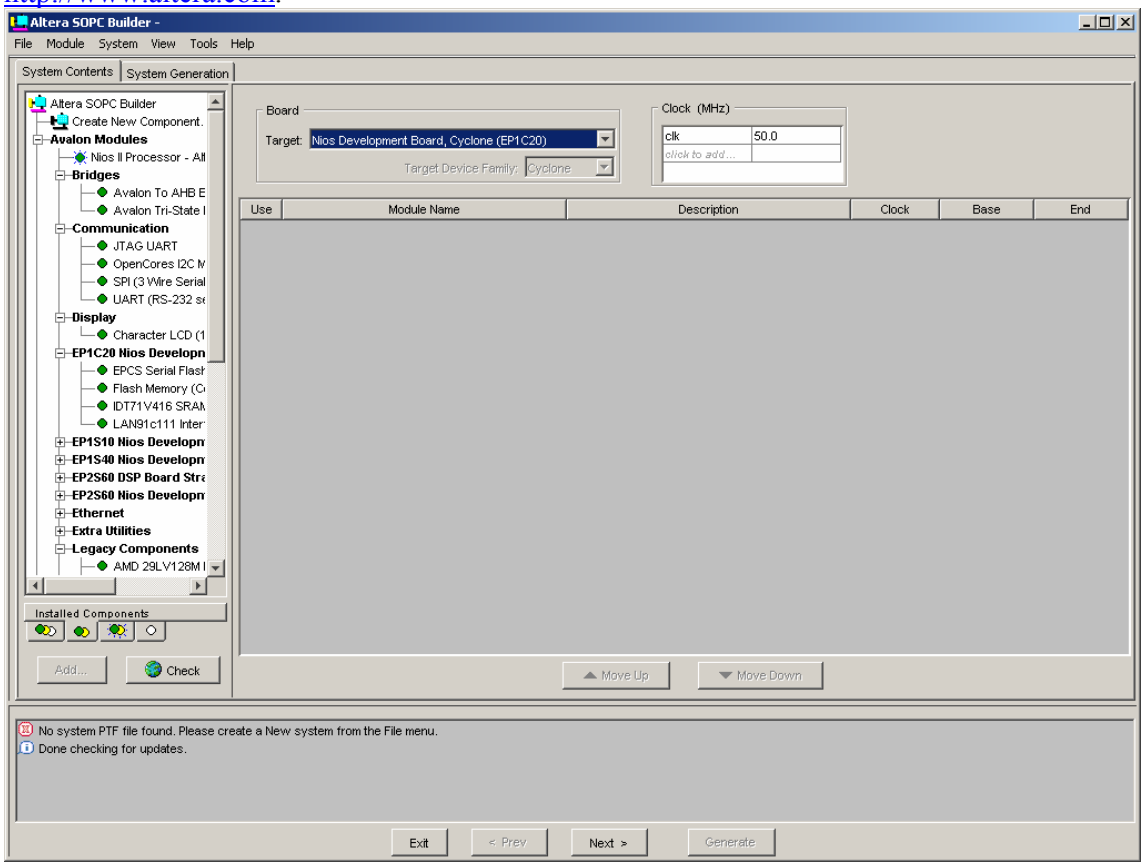
Vi valgte eCos fordi det er et åpen kildekode system og fordi det er ett av to RTOSer for Nios II. Det er aktive og gode fora som en kan stille spørsmål i og så er selve kildekoden til RTOSet ganske godt dokumentert.

[6] ”Hello World” er det programmet en nybegynner i et programmeringsspråk først skriver. Ta en titt på http://en.wikipedia.org/wiki/Hello_world for mer forklaring rundt dette og en god del eksempler.

3.0 Utstyr

3.1 SOPC Builder

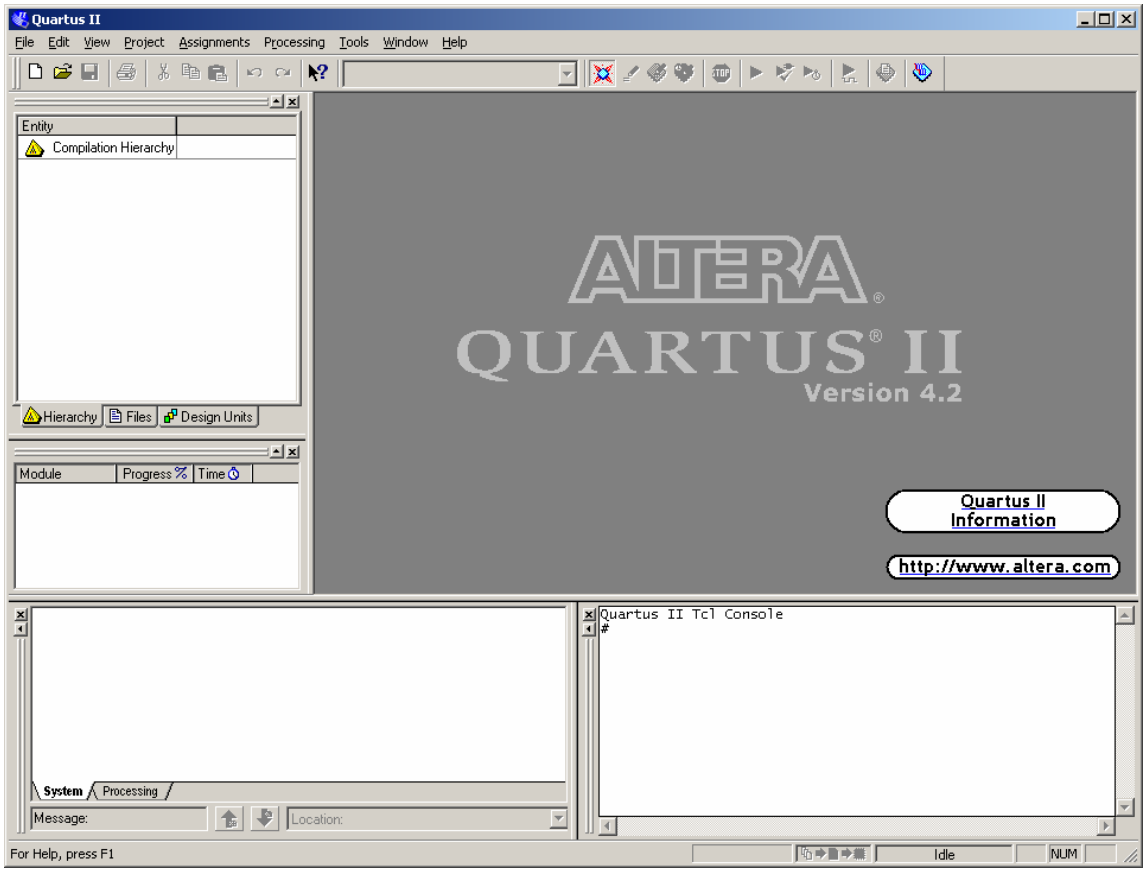
Designet av hardwaredelen av GAUCE ble gjort i SOPC Builder. Dette er et program som er et slags tilleggsprogram til Quartus II og det gjør generering av et system bestående av CPU, SPI, UART, timer, RAM grensesnitt og annet ganske enkelt. Dette programmet er en del av Nios II systemet og kan lastes ned som prøveversjon fra Alteras hjemmeside: http://www.altera.com.



Figur 7 SOPC Builder

3.2 Quartus II

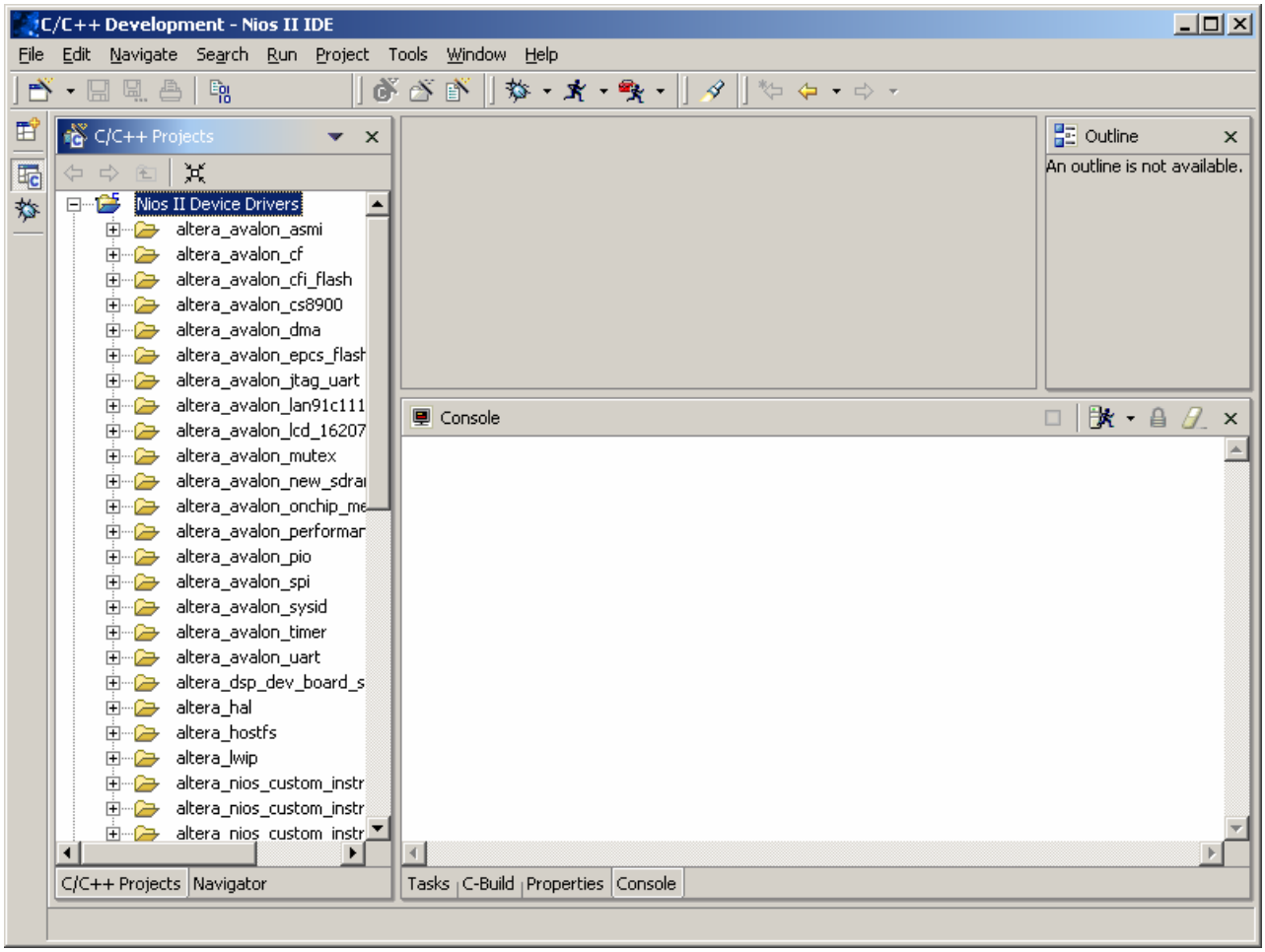
I Quaruts II plasserer en det systemet som SOPC Builder genererte og setter på de fysiske I/O pinnene på det. Dette programmet kan også lastes ned fra Altera hjemmeside: http://www.altera.com. I Norge er det Arrow (http://www.arrowne.com/) som forhandler Altera produkter. Vi har brukt Quartus II 4.2 sp1 til dette prosjektet.



Figur 8 Quartus II

3.3 Nios II IDE

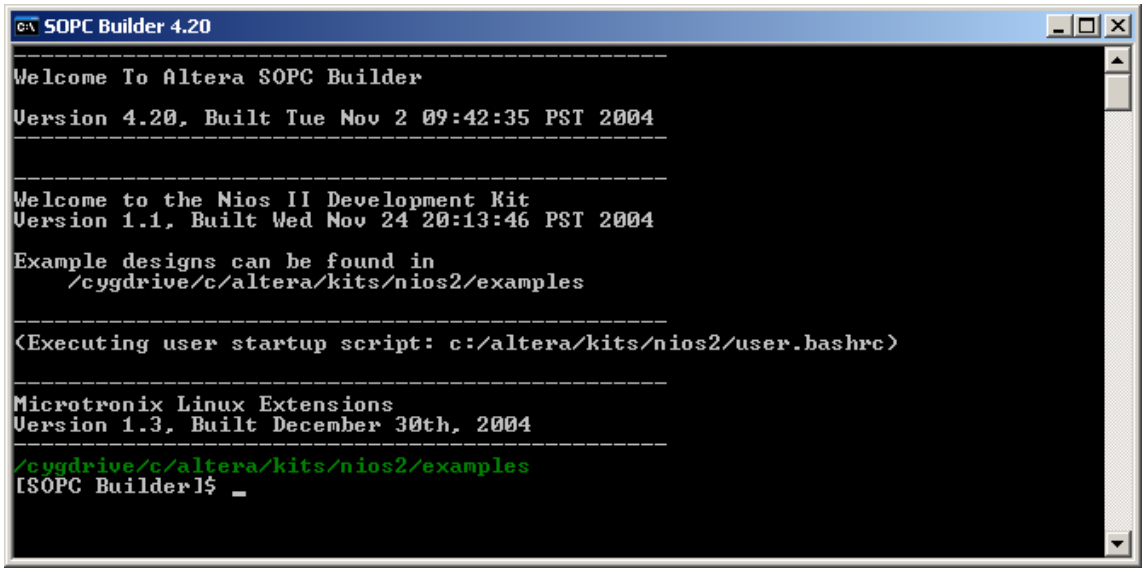
Nios II IDE brukes for å utvikle programmer for Nios II prosessoren. Det er et helt vanlig Integrated Development Enviroment og innholder alt det en kan forvente seg av et. Det er bygd på Eclipse-systemet (http://www.eclipse.org/) som ble startet hos IBM. Nios II IDE er en del av Nios II systemet. Vi har brukt Nios II 1.1 i dette prosjektet.



Figur 9 Nios II IDE

3.4 Nios II SDK Shell

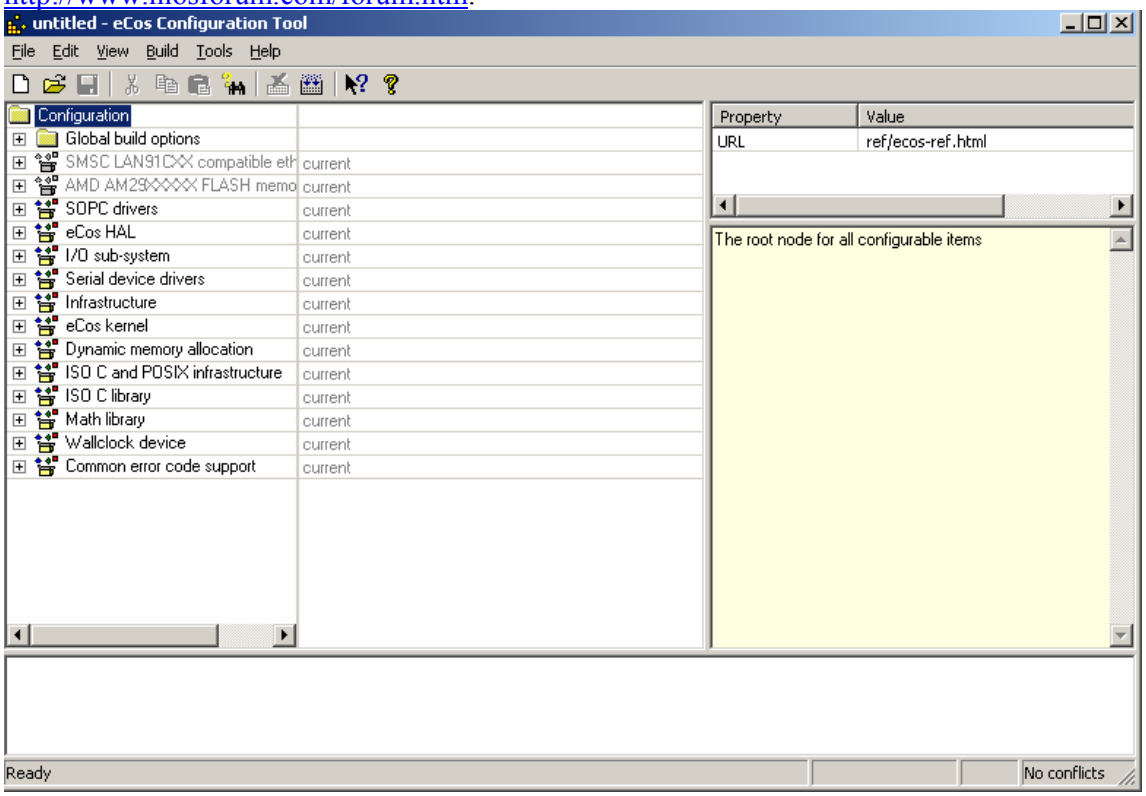
Dette er et bash-skall som vanligvis følger med de flest Unix og Linux systemene. Det er et kraftig skall som gjør at en har tilgang til verktøy som make og gcc utenom Nios II IDEen. Start av eCos Configuration Tool og kompilering av eCos-systemet skjer ved bruk av dette skallet. Dette skallet er også en del av Nios II systemet.



Figur 10 Nios II SDK Shell

3.5 eCos

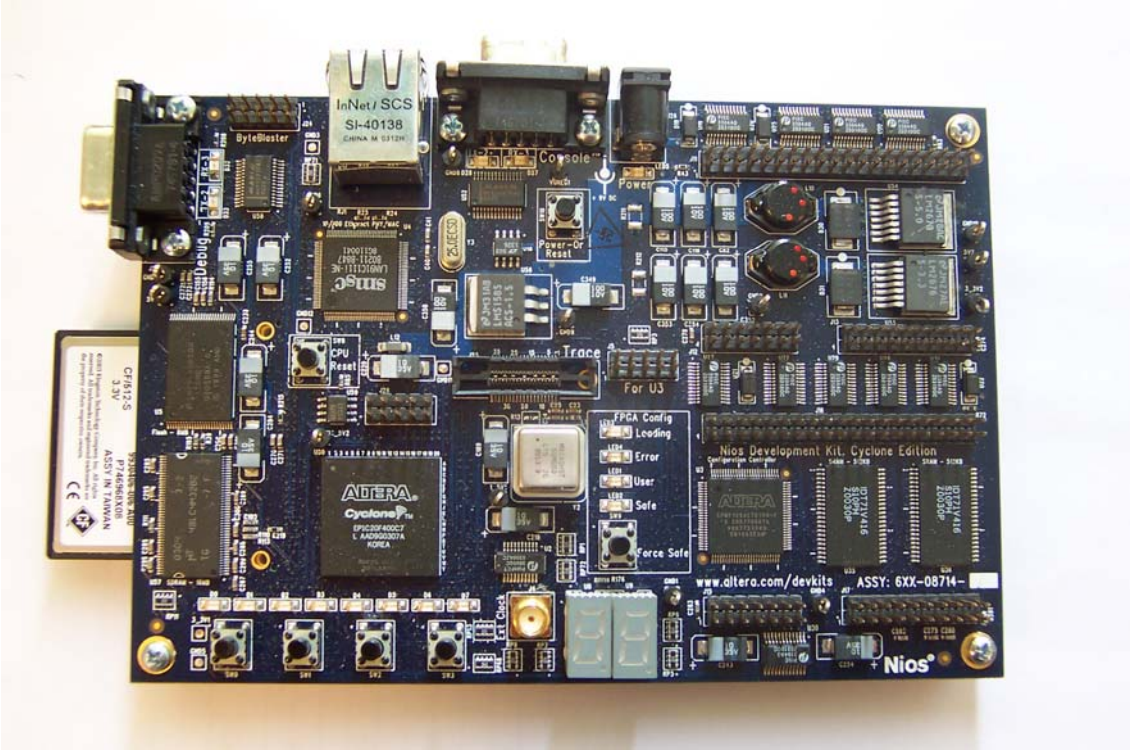
eCos er et operativsystem og med det følger det med en del verktøy for å opprette et fungerende system. Alt nevnt er eCos Configuration Tool som gjør generering av et systembibliotek til en enklere prosess enn det kunne ha vært. Det offisielle hjemmesiden til eCos er http://sourceware.org/ecos/, men den versjonen vi har brukt kan lastes ned fra http://www.niosforum.com/forum.htm.



Figur 11 eCos Configuration Tool

3.6 Nios Development Kit, Cyclone Edition

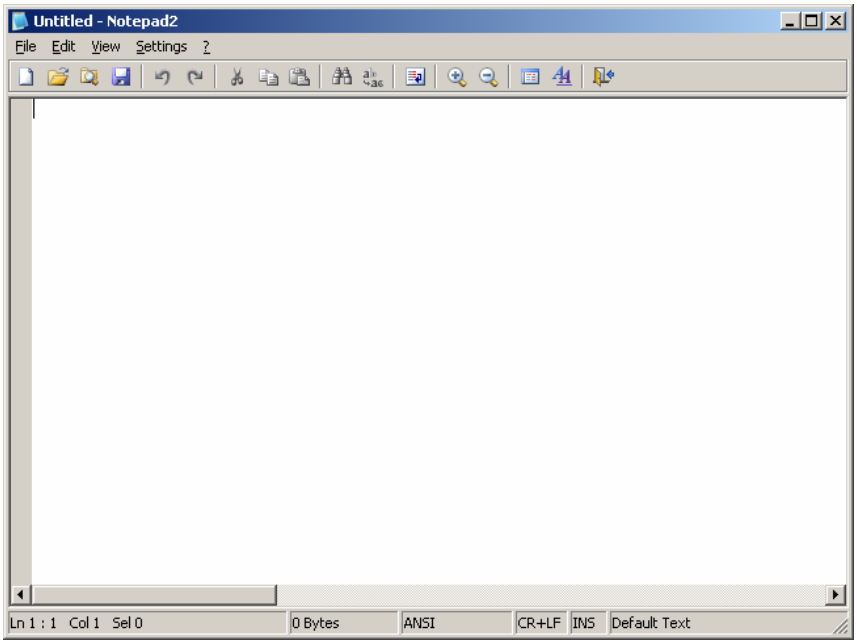
Dette utviklingssettet er det vi har jobbet med hovedprosjektet vårt på. Ved kjøp av dette følger Nios II systemet og en fullversjons Quartus II med.



Figur 12 Nios Development Kit, Cyclone Edition

3.7 Notepad2

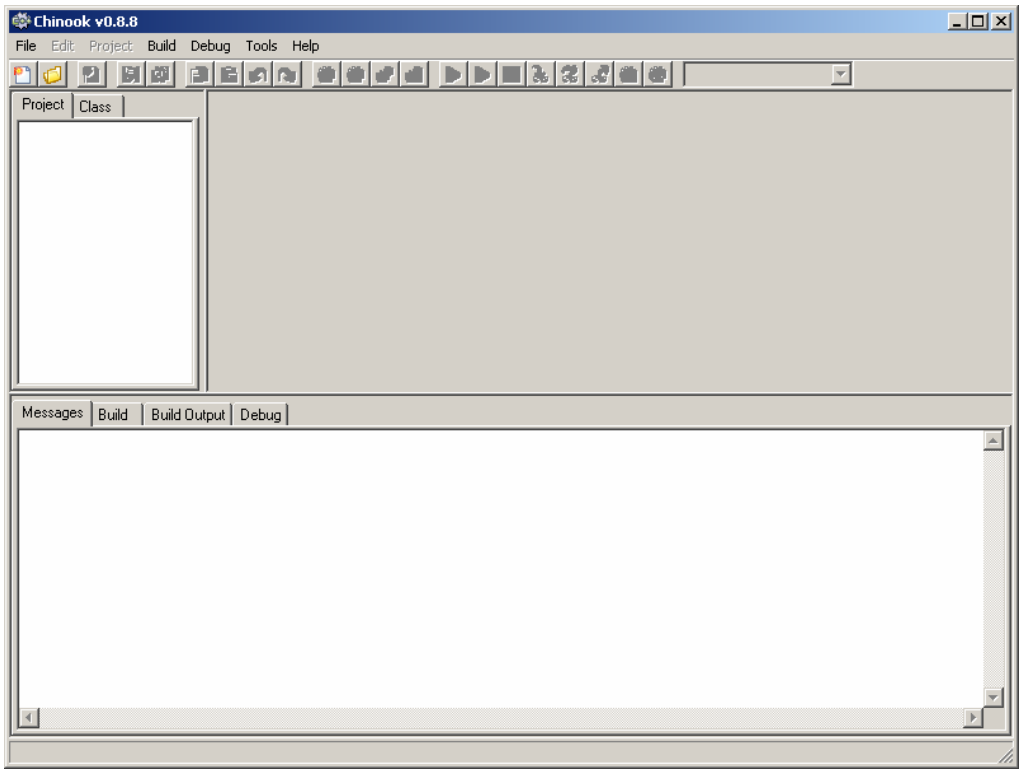
Til redigering av all slags forskjellige filer, ble det utrolig gode norepad2 brukt. Dette viser C-kode i farger slik at kommentarer får en farge, C-nøkkelord en annen og så videre. Programmet er en klon av Notepad som følger med Windows. En kan laste ned Notepad2 fra http://www.flos-freeware.ch/notepad2.html.



Figur 13 Notepad2

3.8 Chinook Free

Til kompilering av PC programmet som benyttet seg av wxWindows rammeverket benyttet vi Chinook Free, en gratis C++ kompilator som har innebygd wxWindows rammeverket. Chinook Free kan lastes ned fra http://www.degarrah.com/chinookfree.php. En må registrere seg på forumet for å få tilgang til å laste den ned.



Figur 14 Chinook Free

4.0 Utførelsen

Det vi har laget under dette hovedprosjektet har vi kalt GAUCE[7]. GAUCE består av et CompactFlash-kort for å lagre samplinger på, en ADC for å hente samplinger fra, en GPS-modul for å få sted og tid fra, et USB grensesnitt for å programmere GAUCE, et program for å programmere GAUCE fra en PC og en hardwaredel realisert på en FPGA bestående av en Nios II prosessor og diverse grensesnitt mot USB, SPI, UART og annet. Vi brukte et operativsystem kalt eCos for å styre hardwaredelen. Nios II prosessoren ble lagt inn på en FPGA fra Altera kalt Cyclone som sitter på et utviklingssett fra Altera som heter Nios Development Kit, Cyclone Edition. Til dette settet lagde vi to utvidelsesmoduler, en med USB og den andre med GPS og ADC. Dette kapitelet beskriver hver enkelt del og hva vi gjorde.

4.1 Utviklingssettet

Vi hadde egentlig planlagt å bruke en PIC-kontoller for å realisere dette hovedprosjektet. Mye kunne blitt annerledes da. Etter at vi bestemte oss for denne oppgaven, gikk vi rundt og funderte litt på hvordan vi skulle løse den. Vi hadde begge litt erfaring med bruk av PIC-mikrokontrollere fra faget Elektronikk konstruksjon. I og med at det skulle være programmerbart hadde vi lyst å bruke USB, så vi funderte på å bruke en løsning med en PIC-mikrokontroller med innebygd USB-grensesnitt. Da vi besøkte FFI for å snakke om hovedprosjektet fikk vi låne en rapport der det var noen andre som hadde gjort mye av det samme vi skulle gjøre. De hadde brukt en PIC-mikrokontroller for å styre en CPLD der CPLDen ble brukt som et skiftregister for å gjøre om seriell data til parallell data som deretter ble skrevet til et CompactFlash kort. Vi begynte å tenke i disse baner.

Da vi var på FFI, ble det sagt at de hadde et utviklingssett fra Altera som inneholdt en FPGA. Vi fikk navnet på settet og undersøkte litt rundt det på nettet. Etter litt søking fant vi denne linken: http://www.altera.com/proencts/devkits/altera/kit-nios_1C20.html. Der sto det om Nios II prosessoren og hva slags utstyr som kunne kobles til denne. Vi skjønte da at vi kunne bytte ut mikrokontrolleren med Nios II-prosessoren og dermed bruke det utviklingssettet vi kunne låne av FFI. Etter en e-post med spørsmål om de kunne sende den oppover, mottok vi utviklingssettet og satte oss rett ned for å sette oss inn i det. Vi innrømmer at en del av grunnen til at vi forkastet PIC-løsningen var at vi syntes at vi kunne lære mer ved å bruke dette utviklingssettet og fordi det virket mer spennende.

Etter litt tenking i startfasen av prosjektet fant vi ut at vi trengte å skrive til et FAT-filsystem[8] på CompactFlash-kortet. Uten å ha et FAT-filsystem på CompactFlash-kortet fantes det ingen enkle måter å få ut data fra det. Med et FAT-filsystem er det bare å sette CompactFlash-kortet inn i en adapter og så laste over dataene til PCen. Vi var inne på tanken på å skrive en driver for FAT, men det ville ta lang tid og det ville sannsynligvis ikke blitt noe bra. Etter litt søking på nettet fant vi denne siden: http://www.niosforum.com/forum.htm. Der kunne vi laste ned et operativsystem, eCos, for Nios II som hadde innebygd driver for FAT-filsystemet. På det

[7] GAUCE står for GPS, ADC, USB, Compact Flash og eCos. Det uttales som Gaus, som i Gausdal. Vi måtte bare ha et navn på hovedprosjektet og GAUCE var det vi fant på.
[8] FAT-filsystemet er det filsystemet som DOS støttet fra starten av. FAT brukes som filsystem i de fleste digitale kameraer, MP3-spillere og lignende i dag. Stort sett alle operativsystem kan skrive og lese til FAT.

utviklingssettet vi hadde fått fra FFI, fantes det bare den første versjonen av Nios-prosessoren, men skolen og FFI spleiset på et nytt utviklingssett slik at vi fikk tilgang til Nios II systemet.

4.2 USB

Etter å ha mottatt utviklingssettet begynte vi rett på USB. USB hadde vi satt opp på framdriftsplanen som det som sannsynligvis kom til å ta lengst tid og derfor tenkte vi å begynne med det først. Vi fikk rett i at USB tok lengst tid.

Vi hadde så klart lagt store planer og etter litt søking på internett fant vi denne siden: http://www.plxtech.com/proencts/NET2000/NET2272/default.asp. Det fantes ikke noen i Norge som solgte denne chipen, men det gikk an å kjøpe den over nettet. Dette krevde dessverre et VISA-kort og skolen hadde dessverre ikke det tilgjengelig. Etter litt mer søking fant vi denne: http://www.semiconenctors.philips.com/pip/ISP1582BS.html. Den virket som en god erstatning for NET2272. Vi tok en del ringerunder og til slutt fikk vi tak i ei hyggelig dame[9] hos Eurodis Electronics som bestilte 10 prøveeksemplarer av ISP1582 til oss.

Da vi mottok disse prøveeksemplarene så vi at de var ganske små. Det var som forventet. Det vi ikke hadde forventet var at skolen ikke hadde utstyr til å lage kretskort som vi kunne lodde disse brikkene på. Vi kunne rett og slett ikke lage kretskort som hadde baner som var under 0.5 millimeter brede. Etter samtale med FFI, fant vi ut at vi kunne få laget kort hos http://www.elprint.no/ og så lodde det hos FFI. Produksjon av et stykk kretskort ville komme på den nette sum av 1200 kroner. Det syntes vi rett og slett var litt for dyrt for et kort der vi ikke kunne garantere at alle tilkoblinger var riktig. Dermed så vi på en USB-brikke som var blitt brukt tidligere i et prosjekt ved HIG og som det var lett å få tak i hos Elfa, nemlig PDIUSBD12.

Vi begynte på å skrive driver for PDIUSBD12 for eCos med utgangspunkt i et par drivere som fantes i eCos, Phillips' eksempelkode og programkoden fra det tidligere prosjektet. Selv med alle disse eksemplene var det ingen enkel jobb. Phillips eksempelkode var satt opp på en dårlig måte og var skrevet for en 8031 mikrokontroller. Programkoden fra det tidligere prosjektet var heller ikke satt opp så godt. Det var hyppig bruk av globale variable der, slik at en ikke hadde helt kontrollen over hva som ble forandret hvor. De to driverne som fantes i eCos var skrevet for andre USB-brikker enn den vi skulle bruke, slik at de bare var nyttige for å se de store strukturene i programkoden.

For å få tilgang til USB-brikken under et Nios-system og dermed også i eCos, måtte det skrives en såkalt komponent for SOPC Builder, programmet en bruker for å sette opp en Nios II system. Denne komponenten ligger på CDen under katalogen sopc. Komponenten består stort sett av to filer. En fil med Verilog-kode som rett og slett bare setter de signalene som kommer på Avalon-bussen, den bussen alle komponentene i et Nios II system kommuniserer over, lik de som ligger på bussen til USB-brikken og omvendt avhengig av slike signaler som Write Enable og Read Enable og Chip Select. Den andre filen, en ptf-fil, inneholder informasjon som hvor bred bussen til USB-brikken er (8 bit) og hvor lenge Write Enable og Read Enable skal holdes lave for at USB-brikken skal registrere det. I SOPC Builder er det bare å legge til en komponent kalt pdiusbd12 og gi det et navn som usb og så generere et Nios II system. Hvordan en skal få dette opp og til å virke er grundig forklart i vedlegg 7.3.

[9] Ved navn Helle Bekkeli for de som var nysgjerrige.

Etter å ha satt opp en komponent skrev vi en kort h-fil, pdiusbd12_regs.h som inneholder et par makroer for å skrive og lese til og fra bussen til USB-brikken. Ved hjelp av denne h-filen er det enkelt å skrive og lese til og fra bussen til USB-brikken. Nios-systemet holder selv rede på hvilke linjer som skal legges lave for å kunne skrive til USB-brikken. Ptf-filen innholder informasjon som gjør at Nios-systemet vet hvilke linjer som skal legges lave og høye for at en kan skrive til USB-brikken. For eksempel må man for å skrive til USB-brikken legge write enable og chip select lave og read enable høy og så legge data på den parallelle bussen mellom FPGAen og USB-brikken. Hvis en bare sender data over bussen må A0-linjen være lav, hvis det er en kommando må A0-linjen være høy. Ved å bruke de kommandoene fra pdiusbd12_regs.h slipper en å styre med enkelt linjer, det gjør Nios-systemet for en.

Driveren bygger på makroene fra pdiusbd12_regs.h og implementerer en del enkle funksjoner som brukes av resten av driveren. Disse funksjonene er lik de som Phillips har opprettet i D12CI.C fra Phillips programkode for PDIUSBD12. Funksjonene er en implementasjon av de forskjellige kommandoene som er beskrevet i databladet[10] til PDIUSBD12. Dette er funksjoner som D12_ReadChipID for å lese identifikasjonsnummeret til USB-brikken og D12_WriteEndpoint for å sende data til USB-brikken og så videre ut fra den.

Alle USB-transceivere har noe som blir kalt for endpointer. En skriver og leser til og fra endpointer for å sende og motta data. Phillips har på denne USB-brikken, PDIUSBD12, gjort en liten brøler. Vanligvis finnes det ett kontrollendpoint og ett eller flere dataendpointer. Dataendpointene kan enten være UT eller INN, der UT er data ut fra PCen og inn på USB-brikken og INN er data inn til PCen. Kontrollendpointet er oftest både INN og UT. Phillips har et vanlig kontrollendpoint der det både er INN og UT, men de har slått sammen de 4 andre endpointene til 2 endpoint som har både INN og UT. Dette skaper litt forvirring når en sammenligner PDIUSBD12 med andre USB-brikker. Phillips sier at PDIUSBD12 har 3 endpoint, et kontrollendpoint og to dataendpoint. Vanligvis ville en ha sagt at PDIUSBD12 har 5 endpointer, et kontrollendpoint og fire dataendpoint.

Driveren som vi skrev støtter kontrollendpointet fullt og UT fra PCen. Vi har dessverre ikke fått INN til å virke, men vi trengte bare UT fra PCen for å programmere GAUCE. Kontrollendpointet mottar forskjellige SETUP-pakker som brukes for å konfigurere enheten. Disse SETUP-pakkene er beskjeder som SetAddress og GetDescriptor og er beskrevet i teoridelen om USB. Driveren sammen med applikasjonen støtter alle vanlige requester og noen HID-requester som er nødvendig for å få GAUCE gjenkjent som en HID-enhet under Windows. Driveren støtter også datapakker ut fra PCen, men vi har ikke klart å sende data fra GAUCE til PCen.

Driveren består av en ISR, Interrupt Service Routine, som tilkaller en DSR, Deferred Service Routine, som deretter tilkaller forskjellige funksjoner avhengig av hva slags avbrudd som skjedde. Bruken av DSR gjør at det kan komme avbrudd som kan behandles samtidig med at DSRen kjører. Avbruddene er altså aktivert under kjøring av DSR, vanligvis slåes de av ved kjøring av ISR. For mer detaljer på dette området viser vi til programkoden som er lagt ved som vedlegg ?. usbs_pdiusbd12_isr er ISR-funksjonen som tilkalles når det skjer et avbrudd. Lesing av programkoden kan fint startes ved denne koden.

[10] Databladet heter PDIUSBD12.pdf og finnes i datablad-katalogen på CDen.

Det er brukt en del ”tilbakekallingsfunksjoner” i driverkoden. Et eksempel på en slik funksjon er standard_control_fn som finnes i ep0 strukturen i driverkoden. Vi bruker disse ved å sette opp en peker til en tilsvarende funksjon i applikasjonen vår, som dermed gjør at vi kan erstatte kode i driveren med vår egen kode. Dette er gjort flere steder, en titt på programkoden for applikasjonen GAUCE viser hvor disse er. Vi måtte skrive en egen funksjon som tar plassen til standard_control_fn for å behandle en del av requestene og et par HID-requester på riktig måte.

4.3 GPS

Vi hadde opprinnelig tenkt å bruke MG4200 fra Motorola, men det var noen problemer rundt denne. Det står på hjemmesiden til MG4200 at den er i proenksjon, men distributøren i Norge hevdet at den ikke var det. Vi prøvde å få tak i MG4100, den tidligere modellen, men det ble for dyrt. Det måtte sendes minst 10 st fra USA, og det ville komme på rundt 1500 kroner med toll. Så vi fant en annen moenl, TIM-LL fra u-blox. u-blox har en butikk på Internett, slik at en kan kjøpe proenktene deres. FFI kjøpte et stykk av TIM-LL for oss siden skolen ikke hadde tilgang på VISA-kort.

TIM-LL monterte vi på et testkort sammen med ADCen. TIM-LL kommuniserer med Nios-systemet over en UART. Vi bruker NMEA protokollen for å kommunisere med TIM-LL. NMEA protokollen er beskrevet i dokumentet ANTARIS_Protocol_Specification.chm som ligger i dokumenter-katalogen på CDen. Vi har dessverre ikke fått testet ordentlig at GPS-moenlen virker på grunn av dårlig tid. Mottaksforholdene er ikke så gode på elektrolabben og heller ikke ut av vinenet der, slik at vi ikke har fått tatt imot noen signaler.

TIM-LL sender konstant ut forskjellige meldinger om tid, sted, fart og annet. I applikasjonskoden slår vi av alle disse meldingene og setter opp for å sjekke de i eget tempo. Det er et eller annet galt med driveren for UART, slik at vi ikke klarer å lese med enn 25 tegn på en gang fra UARTen, men da vi ikke får inn noen satellitter med moenlen, er det ingen meldinger som er på mer enn 25 tegn uansett. Et alternativ her er å bruke den UART-driveren som Nios-systemet har tilgjengelig, som vi har gjort med SPI, men det krever en del koding og det hadde vi rett og slett ikke tid til.

4.4 ADC

ADCen er en AD7888 fra Analog Devices. Den tar inn 8 analoge signaler og kan sample de med opp til 125 000 sampler i sekundet. AD7888 kommuniserer med Nios-systemet over en SPI tilkobling.

eCos har en del programkode for å støtte SPI, men det fantes ikke en driver for SPI på Nios-systemet i eCos. Vi hadde tenkt å skrive en, men vi fikk ikke tid til å gjøre det ordentlig, så vi skrev en driver som går bak eCos rygg kan en si. Den lille driveren er i altera_avalon_spi.c som finnes under gauceapp-katalogen på CDen. Denne driveren sender ut 16 bit med data over SPI-bussen og leser 16 bit tilbake og returnerer disse 16 bitene. Hvert sample er på 12 bit, slik at de 3 øverste bitene er brukt til å vise hvilken kanal sampelet er fra. Denne maskeringen skjer i applikasjonsprogramkoden.

4.5 GAUCE applikasjonen

GAUCE-applikasjonen styrer GAUCE. Applikasjonen bruker USB-driveren, UART-driveren og SPI for å kommunisere med eksterne enheter. Jobben til applikasjonen er å lese fra Flash dataene for samplingen. Disse dataene er hvilken forsinkelse det skal være før applikasjonen begynner å sample, hvor lenge den skal sample og hvilken frekvens det skal være på samplingen på de forskjellige kanalene. Alle disse verdiene programmeres inn på GAUCE over USB ved hjelp av programmet på PCen. Forsinkelsen bør gjøres om til et tidspunkt når GPSen fungerer slik den skal. Sånn som GAUCE er satt opp nå, finnes det ingen klokke eller lignende for å passe på tiden, slik at GAUCE starter opp på 0 sekunder hver gang den starter opp. Det bør kanskje vurderes å koble en sanntidsklokke med batteri til GAUCE slik at den har riktig tidspunkt uten å være avhengig av GPS.

I oppstarten setter applikasjonen også opp noen datastrukturer slik at den vil bli gjenkjent som en HID-enhet hvis den blir koblet opp til en PC. Når GAUCE er gjenkjent kan den bli programmert med det programmet som kjører på PCen.

Etter at forsinkelsestiden er over går GAUCE inn i en loop som varer så lenge den er programmert til å vare. I denne loopen leser applikasjonen inn sampler fra ADCen, maskerer de med kanalen og lagrer samplet i et midlertidig buffer. Når bufferet er fullt, skrives det til disk. Sånn som applikasjonen er satt opp nå, lages det en ny fil for hvert minutt samplingen foregår. Dette er på grunn av at hvis applikasjonen hadde skrevet til en fil, ville vi ha mistet all den dataen hvis GAUCE hadde mistet strømmen. Hver fil må lukkes for at den dataen filen inneholder skal bevares. Denne svakheten ligger i FAT-filsystemet. Bruk av et filsystem som JFFS2 som brukes på flashet på utviklingssettet har ikke denne svakheten, men da ville vi ikke kunne lese den dataen som ble skrevet til CompactFlash-kortet fra en Windows PC fordi den ikke har de riktige driverne. En Linux PC har støtte for JFFS2. Et alternativ kunne være å overføre dataen over USB, men da trenges det en del modifikasjoner som en raskere USB-brikke og støtte for DMA.

4.6 PC programmet

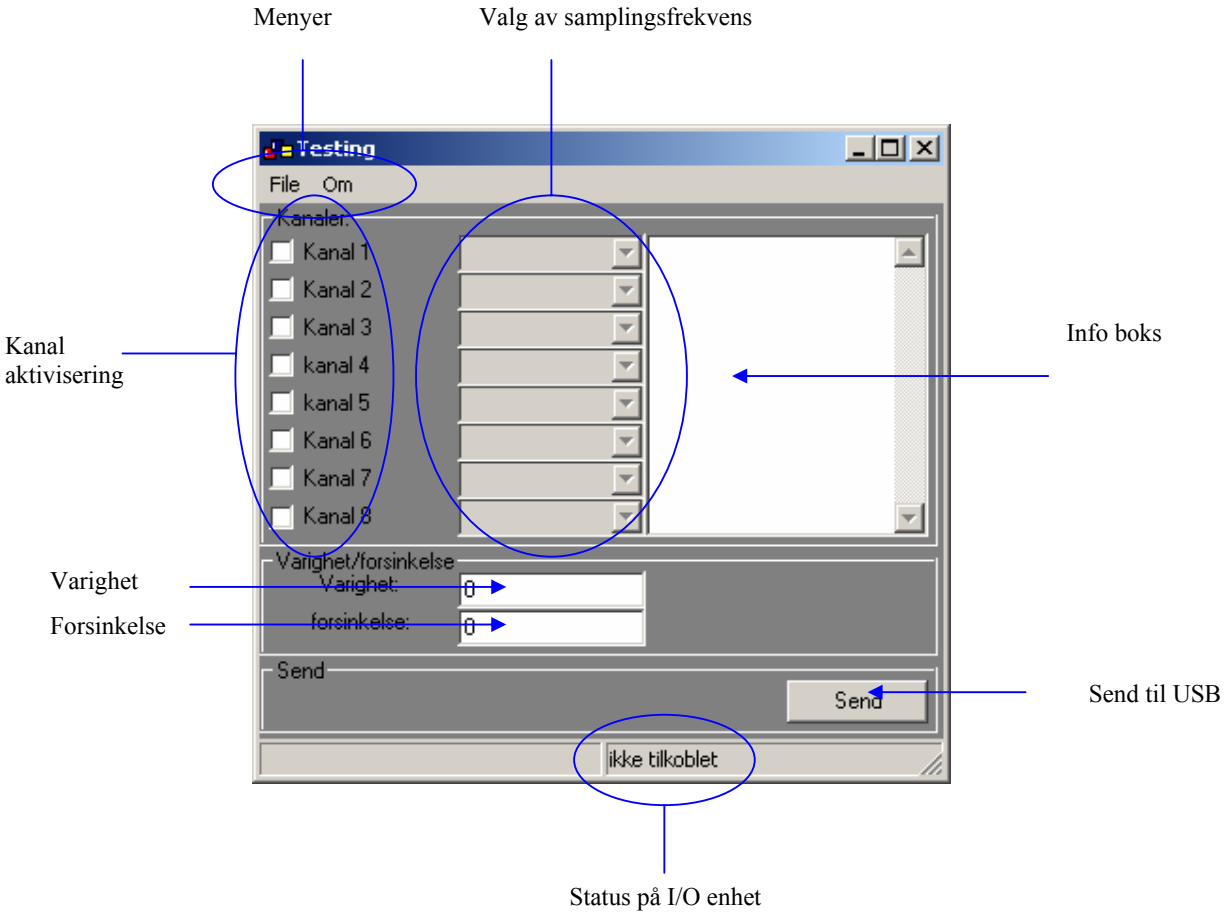
Hensikten med pc programmet var a det skulle være en enkel måte å programmere GAUCE[11] på. Det som skal kunne programmeres ved hjelp av programmet er hvilke kanaler som skal benyttes, hvilke samplings frekvenser disse skal ha og når samplingen skal begynne.

Grensesnittet (GUI)

Vi valgte å benytte et grafisk grensesnitt etter som dette er mer ryddig og oversiktelig. Dette medfører at det er enklere å sette seg inn i programmet. Etter som ingen av oss hadde programmert noe grafisk grensesnitt måtte vi først sette oss litt inn i hva som fantes. Vi endte opp med å velge å benytte et rammeverk til dette kalt wxWindows. Grunnlaget for dette er at wxWindows kan benyttes på flere forsjellige typer operativsystemer uten vesentlige forandringer (visse funksjoner er plattform avhengige). Vårt program er likevel ikke plattform uavhengig etter som vi har benyttet oss av dll filer som ligger under Windows. Andre grunner

[11] GAUCE er navnet på I/O - enheten

til at vi valgte å benytte wxWindows er at det er basert på C++ som vi har noe erfaring med fra før.
Etter som det ikke var veldig mange innstillings muligheter som skal være med valgte vi å lage alt i et skjermbilde. Dette for at det skulle bli mest mulig oversiktlig.



Figur 15 oversikt over pc programmets GUI

Menyer

Etter som vi har valgt å kjøre alt i et skjermbilde er det ikke mye vi trenger menyer til, men for å få det inn etter den kjente Windows malen har vi lagt til en menylinje med til menyer:

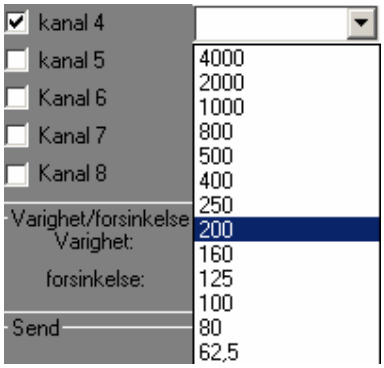
File: Herunder er det bare et valg og dette er Exit. Denne avslutter programmet.

Om: herunder er det å bare et valg og dette er Om bare inneholder navnet på programmet og versjonsnummeret. Herunder hadde vi også tenkt å legge en hjelpe funksjon men vi fikk ikke tid etter som vi brukte noe lenger tid med andre ting enn det vi hadde regnet med.

Kanal aktivisering

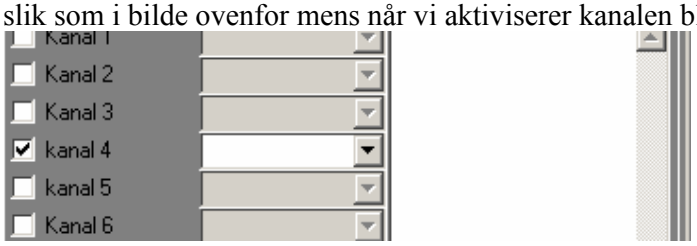
Dette er sjekkbokser som en krysser av ved å klikke på dem med venstre musknapp. Når en klikker på den betyr dette at denne kanalen vil bli logget. En kan selv velge hvilke som en vil aktivisere, det er ingen begrensninger på at en må bare ha etterfølgende kanaler eller noe slikt.

Valg av samplingsfrekvens
Disse boksene er som navnet tilsier der en velger hvilke samplingsfrekvens en ønsker på den tilhørende kanalen[12]. Dette gjøres ved hjelp av en nedtrekksmeny som velger den ønskede frekvensen.



Figur 16 rullgardin menyen

Grunnen til en ikke kan skrive inn hvilke frekvens en vil, er beskrevet under valg av samplingsfrekvenser.
Det er ikke mulig å velge samplings frekvens hvis ikke kanalen først er aktivisert med å krysse av den tilhørende kanal aktivisereren. Dette ser vi ved at fargen til valg boksen er grå slik som i bilde ovenfor mens når vi aktiviserer kanalen blir den hvit.



Figur 17 Enabla rullgardin meny

Infoboks
Infoboksen er som navnet tilsier der informasjon om valgene en har gjort i programmet. Informasjonen kommer opp når en trykker på send knappen. Dermed kan en se om det som ble sendt er det en ønsket.

Varighet
Varighet er der en skriver inn hvor lenge samplingen skal pågå. Dette er oppgitt i sekunder.
Maks varighet nå når det er et 512MB kort som brukes til lagring blir da.

$$t = \frac{2^{29}}{N_{\max\ kanaler} \cdot I_{bit\ pr\ sample} \cdot F_{\max\ fs}} = \frac{536870912}{8 \cdot 16 \cdot 4000} = 1048{,}576$$

Ommgjordt til min :

$$t_t = \frac{t}{60} = \frac{1048{,}576}{60} = 17{,}476$$

Vi ser at vi kan maks kjøre samplingen i litt over 17 minutter på full hastighet.

[12] Tilhørende sjekkboks er den som ligger rett til vestre

Bit pr sampling blir 16 bit i stede for 12 som er oppløsningen til ADCen fordi vi ikke kan overføre mindre enn en byte på SPI-bussen. 12 bit blir da 2 byte.

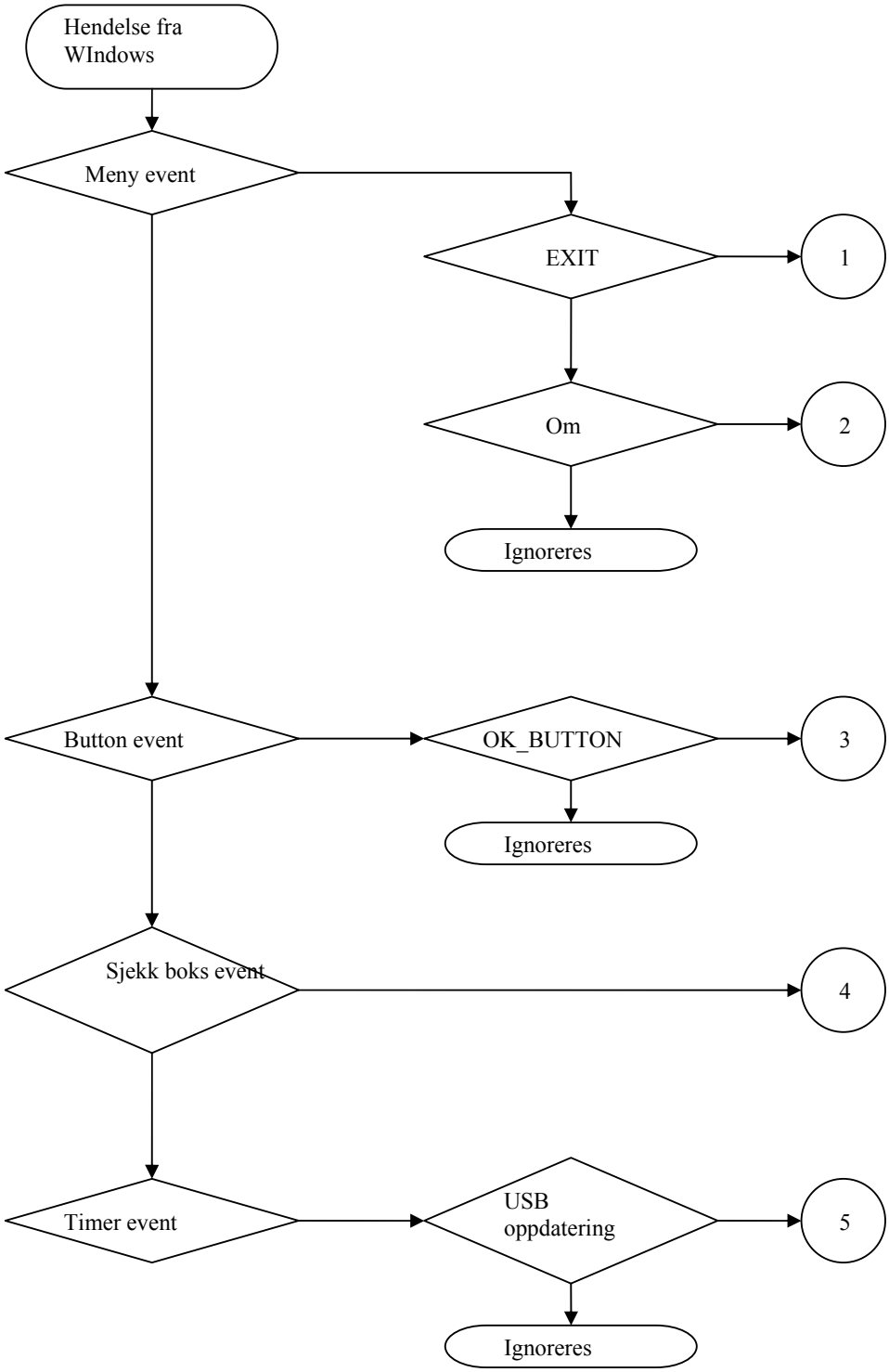
Forsinkelse
Det er her en skal skrive inn når en skal ha samplingen utført. På grunn av at vi ikke fikk gjort alt vi ville er det ikke lagt inn tilstrekkelig kode for denne i denne koden som ligger ved, men tanken var at en skriver inn dag og klokkeslett. Og ut fra dette og informasjonen fra GPSen finner GAUCE ut når den skal starte å samle inn data.

Send til USB
Denne knappen starter en rutine som sjekker om en har lagt inn tilstrekkelig informasjon så GAUCe har det den trenger for å utføre jobben sin. Hvis en ikke har lagt inn tilstrekkelig informasjon kommer det en feilmelding opp og en må fikse dette før en får sendt til GAUCE.

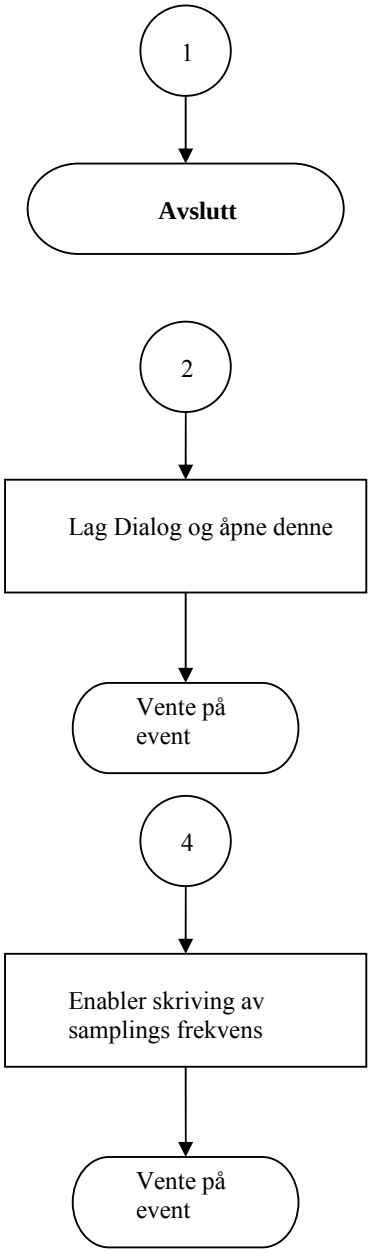
Status på I/O enhet
Viser om GAUCE er tilkoblet USB inngangen eller ikke. Denne oppdateres hvert millisekund.

Flytskjema for PC programmet

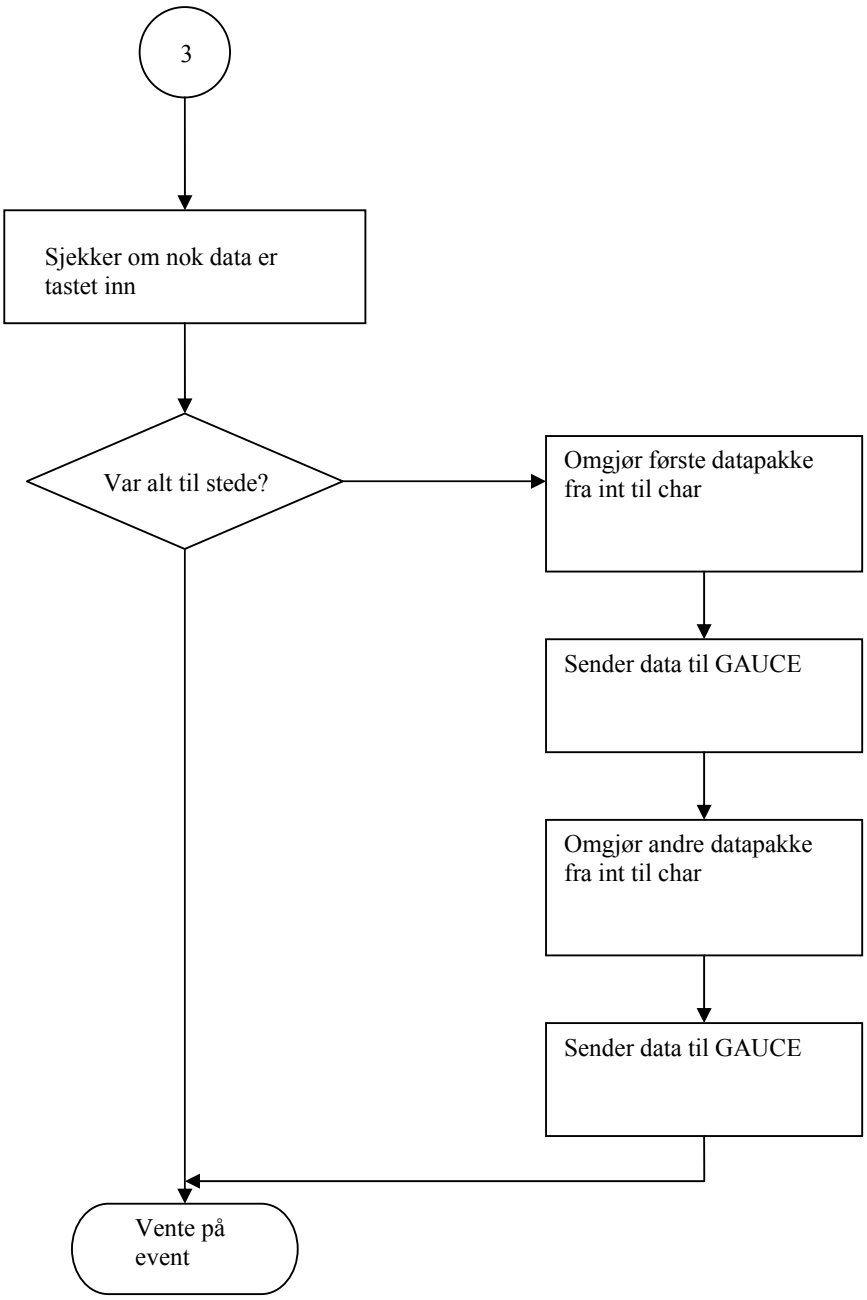
Merk at dette ikke er hele programmet for noen av boksene inneholder ganske mye kode for eksempel i forbindelse med USB kommunikasjon.


Hendelse fra
WIndows
Meny event
EXIT
1
Om
2
Ignoreres
Button event
OK_BUTTON
3
Ignoreres
Sjekk boks event
4
Timer event
USB
oppdatering
5
Ignoreres

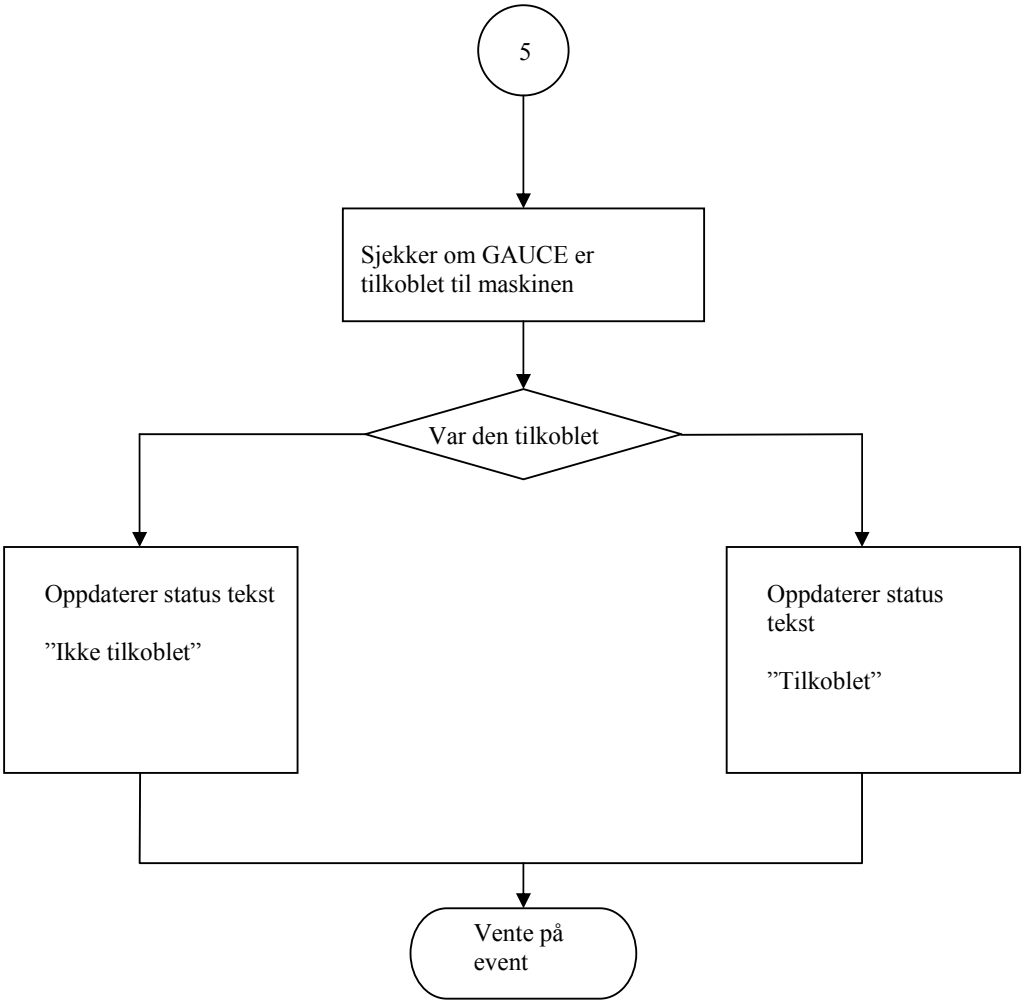
Figur 18 Flytskjema for hendelsestabellen



Figur 19 Flytskjema for Pc programmet



Figur 20 Flytskjema for sendknappen



Figur 21 Flyt skjema for timer event

1. Aktiviseres ved at en trykker på Exit i File menyen. Denne avslutter programmet.

2. Aktiveres ved at en trykker på Om i Om menyen. Denne lager en dialog boks. Denne dialogen har også en egen knapp som ikke blir omhandle her.

3. Aktiviseres når en trykker på Send knappen. Denne sjekker at en har skrevet tilstrekkelig med data og så sender den det til GAUCE vis den er tilkoblet.

4. Aktiviseres når en krysser av en av avkryssningsboksene. Denne enabler valg boksen for samplings frekvenser slik at vi bare kan velge samplings frekvens for de som er krysset av.

5. Aktiviseres med at timeren har nådd så langt som en har satt den til. I vårt program er den satt til 1 ms. Den søker gjennom alle HID enheter som er tilkoblet maskinen for å se om den finner GAUCE.

Valg av samplings frekvenser:

Maks antall sampler per sekund blir etter kravene vis vi setter opp til at den skal kunne sample på alle inngangene med maks samplingsfrekvens, 4 kHz. Dette tilsvarer 32000 tusen sampler i sekundet. Etter som vi bruker 0.5 s til å lese av ADCen og skrive dataene til CompactFlash-kortet og vi da i tillegg skal ha inn et GPS signal rekker vi ikke sample flere enn et helt antall ganger 32000. Dermed ble dette det antallet ganger vi sampler i løpet av et sekund og for å kunne dele opp disse samplene slik at ikke flere kanaler kommer over hverandre må de vente n*8 [13] antall samplinger før de kan samples igjen. Dermed blir alle mulige samplings frekvenser gitt ved formelen:

$$Fs = \frac{32000}{n \cdot 8}$$

Dette gir oss 4000 mulige frekvenser men mange av disse har uendelig mange eller mange desimaler så vi valgte å ikke ta med disse. Dette ga oss følgende frekvenser å velge mellom.

4000, 2000, 1000, 800, 500, 400, 250, 200, 160, 125, 100, 80, 62,5, 50,
40, 32, 25, 20, 16, 12,5, 10, 8, 5, 4, 2, 1

Tegning av testkort og forslag til ferdig krets:

Vi har et prosjekt som så å si er bare digitalt. Eneste delen som ikke er digital er inngangene til ADCen og de fikk vi som eneste krav at skulle tåle en inn spenning fra 0-5V. Dermed er det ikke noe konstruksjon av forsterkerkretser eller filtre. Derimot besto mesteparten av utfordringen i å sette seg inn i et nytt skjema design program. Dette programmet het Orcad og var en av betingelsene fra oppdragsgiver. Orcad har meget gode hjelpefiler på det meste av det en trenger å gjøre for å få skjemaet sitt tegnet og ut som et print utlegg, men det har noen glipper i disse hjelpefilene som kan skape vansker. Men etter en stund med prøving og feiling ble lærte vi oss det til slutt. Vi har skrevet en liten innføring i Orcad Captur og Layout som ligger på CDen under kretstegninger/forslag til ferdig krets.

Testkortene er beskrevet under vedlegg 4 (7.4)

Forslag til ferdig krets:

Alle kretsene her er tegnet ut fra databladene til de forsjellige kretsene og etter hvordan noen av dem er benyttet i Nios utviklingskortet. Step up/down er beskrevet i vedlegg 5 (7.5) Det meste av komponentene som er benyttet hadde oppgitt alt inkludert avkoblingskondensatorer og for de som ikke hadde oppgitt avkoblingskondensatorer benyttet vi 0,1 µF som avkoblings verdi. For å også kunne koble vekk høyfrekvent støy der dette kunne ha en innvirkning la vi til en 1 nF kondensator i parallell med 0,1 µF kondensatoren.
De eneste komponentene det ikke var oppgitt noe spesifikk verdi på er L1 og L2 som sitter og filtrerer spenningen inn på høyhastighetsspenningstilførselen på FPGAen men det skal være en Ferrite bead som har høy impedans i 50MHz området og oppover. Data bladene til kretsene finner en på CDen under datablad-katalogen.
Tegningen til den foreslåtte kretsen er på CDen under kretstegninger/forslag til ferdig krets. Grunnen til at det ikke ble laget noe kretskort forslag til denne kretsen er at det vil være ønskelig å legge til DMA.

[13] Der N er et heltall.

5.0 Konklusjon

Vi klarte dessverre ikke å komme helt i mål med dette prosjektet. Noen av grunnene har blitt beskrevet tidligere i denne rapporten. Her skal vi prøve å ta de opp og forklare de litt hver for seg.

Nesten alt som dette prosjektet handler om har vært nytt for oss. Ingen lærere eller andre på skolen har vært noe særlig borte i det vi har drevet med, så det har ikke vært så mye å hente der. Derfor måtte vi lære oss alt selv, og prøve og feile litt. Det tar alltid tid å sette seg inn i noe nytt.

Vi har ikke hatt større programmeringsoppgaver før. De to største delene av kildekoden til dette prosjektet er brukergrensesnittet på PCen og driveren for USB. PC-programmet er på rundt 1000 linjer og driveren er på rundt 1500 linjer. Når en ikke har skrevet så mye kode før gjør en mange feil slik som å ikke helt vite hvor en peker peker og hvordan organisere flerkildekodefil programmer på best mulig måte.

Vi hadde problemer med å skrive til CompactFlash-kortet raskt nok, men det ser ut til at det løste seg. Ved bruk av funksjonen fwrite brukte GAUCE 1.2-1.4 sekunder på å skrive 32000 sampler til disken. Dette er 64000 byte. Ved bruk av write fra unistd.h, brukte GAUCE 0.5 sekunder på å skrive 32000 sampler til disk, en ganske markant forbedring. Ved bruk av fprintf brukte GAUCE 3 sekunder på å skrive til disk. Vi brukte mye tid på å prøve å skrive fortere til disk, og vi fant write til slutt. Hvis vi hadde hatt mer erfaring i programmering og kunnskap om C, hadde vi visst at write sannsynligvis var den funksjonen som var raskest. Nå vet vi det. Det kan være greit å skrive til disk enda fortere en gang senere. Da ville det nok lønne seg å implementere DMA. DMA avlaster prosessoren og kontrollerer overføringer fra minne til disk. Slik som systemet er satt opp nå, har vi en teoretisk maks overføringshastighet på rundt 600 kB/s. Med DMA kan vi sannsynligvis få flere MB/s, altså nærmere den teoretiske maks hastigheten til CompactFlash-kortet. Dessverre er ikke de riktige pinnene koblet til på utviklingssettet for å få DMA til å virke, så det må lages et nytt kort til dette. Hvis de som skal lage en prototyp ut av det vi har gjort på dette hovedprosjektet, anbefaler vi de å i alle fall legge til rette for DMA.

Det viste seg at USB ikke var så veldig lett å få til. Vi startet helt fra grunnen av uten noen særlig kunnskaper om USB, og vi var kanskje dumme da vi tok på oss å implementere det. Vi mener det viktigste med et hovedprosjekt er å lære, og vi har virkelig fått lært mye om USB. Vi startet med å lage et komponent som kunne brukes i SOPC Builder og det tok litt tid for å få timinger og slikt til å være riktig. Samtidig med dette begynte vi å skrive driveren for USB-brikken. Vi hadde heldigvis noen eksempler på hvordan en skulle drive denne USB-brikken. Vi hadde tilgang på programkode som var skrevet for et hovedprosjekt våren 2004, men det brukte en helt annen mikrokontroller, så det var ikke så veldig mye nyttig der. Phillips hadde også et eksempel på programkode for USB-brikken, men også denne brukte en annen mikrokontroller og så var den ikke så bra skrevet. Det fulgte med et par USB-drivere med eCos og de hjalp også en del. Uten disse å se etter, hadde det nok tatt enda en del lengre tid å skrive denne driveren. Vi brukte alt for lang tid på den sånn som det var. Heldigvis fikk vi alt unntatt å sende data fra USB-brikken til å virke. Vi er dessverre usikre på hvorfor vi ikke klarer å sende data ut fra GAUCE, og vi vet ikke sikkert om det er feil hos PCen eller USB-driveren. GAUCE er ikke avhengig av å sende data til PCen sånn som den er satt opp nå, men

det hadde vært kjekt å sende for eksempel GPS data til PCen. Et annet punkt her er å kanskje bruke en raskere USB-brikke og så sende samplingsdata til PCen igjennom GAUCE, i stedet for å måtte ta ut CompactFlash-kortet og bruke en CompactFlash leser. Da må det nok implementeres DMA på både CompactFlash-kortet og USB-brikken for å få det raskt nok. Vi var inne på å bruke en annen USB-brikke, ISP1582BS, også fra Phillips, enn den vi brukte nå, PDIUSBD12, til kommunikasjon mellom GAUCE og PCen. ISP1582 støtter USB 2.0 så med den og DMA kunne vi hatt opp mot 480 Mb/s mellom GAUCE og PCen. Vi gikk bort fra den på grunn av at vi ikke kunne lodde så små brikker på skolen og fordi PDIUSBD12 virket lettere å få i gang med tanke på de eksemplene vi visste om.

Det tar tid å sette seg inn i programmer. Vi hadde en del problemer med å få opp USB-brikken da vi mente at driveren var god nok til å få den til å fungere. Et av disse problemene var at vi hadde glemt å sette alle ubrukte pinner på FPGAen til input, tri-stated. De sto som output til jord og da gikk det så klart galt når vi lastet opp Nios-systemet til FPGAen og startet applikasjonen.

Et annet eksempel som det er verdt å ta opp her er et programmeringseksempel. For å få GAUCE gjenkjent som HID må GAUCE sende over en del data. Et av trinnene i denne prosessen med oversendelse av data er en kopiering av noen strukturer inn i en streng. Etter denne kopieringen fikk en del av disse kopierte bytene feil verdi. Vi kopierte 41 byte inn i en streng som skulle kunne fint ta 41 byte. Men da strengen bare var på 50 char elementer, altså 50 byte, gikk det galt. Ved å utvide denne strengen til 84 elementer, gikk kopieringen i orden og vi fikk sendt over riktig data. Vi er usikre på hva som kan være årsaken her. Dårlig RAM, kopieringsrutiner som krever for mye plass eller noe annet? Dessverre klarte vi ikke å finne dette ut, vi bare fant ut at det fungerte og måtte bare slå oss til ro med det.

Tanken var å lage en prototyp av dette hovedprosjektet. Det fikk vi ikke gjort. En prototyp kunne vi uansett ikke kunne lage her på skolen på grunn av at skolens utstyr ikke klarer å etse så tynne baner som vi trenger nøyaktig nok og det er ikke utstyr til å lodde noen av de komponentene vi trenger å bruke. Skolens utstyr klarer ikke å lage gode nok baner som er under 0.5 millimeter brede. Når vi ikke klarte å få programvaren til å fungere slik vi ville, så vi ikke vitsen i å bestille et kort til 1500 kr eller mer avhengig av antall lag og leveringstid. Vi var også litt usikre på akkurat hva slags komponenter vi trengte å bruke. Utviklingssettet har for eksempel to stykker Flash-brikker som begge kan inneholde konfigurasjonsdata for FPGAen. Den ene må sende dataen igjennom en CPLD for å gjøre de serielle. Vi var usikre på om vi måtte ha en CPLD, eller om vi kunne klare oss uten. Vi kan nok klare oss uten CPLD.

Vi hadde noen problemer med kretstegningen etter som vi også her måtte sette oss inn i et nytt program, Orcad. Det har en meget god hjelpefil på visse områder, så mye av det vi måtte lære oss gikk fort, men akkurat i overgangen mellom skjemategning og kortutlegg er det lite informasjon lett tilgjengelig i Orcad. Et tidligere hovedprosjekt hadde også benyttet Orcad så vi leste i den rapporten og de hadde beskrevet dette steget noe bedre så vi kom i gang og etter vært skjønte vi mer sammenhengen i det.

PC programmet var også nytt for oss, vi hadde heller ikke programmert GUI før. Men rammeverket vi valgte var meget godt dokumentert så etter startsfasen der en ikke helt har fått orden på alle de løse endene og ikke er helt sikker på hva som henger sammen med hva kom det fort. Vi er selvsagt ikke ferdigutdannet i det, men vi har lært mye siden vi startet på prosjektet.

I sluttfasen av dette prosjektet utga Altera nye versjoner (versjon 5.0) av Nios II-systemet og Quartus II. Det ble også lagt ut en ny eCos på http://www.niosforum.com/forum.htm. De fulle versjonene av Quartus II og Nios II var ikke tilgjengelig i det vi skriver dette, så vi har dessverre ikke fått prøvd GAUCE på den nye utgivelsen. Sannsynligvis er det litt som må forandres for å få brukt GAUCE på den nye versjonen av Nios-systemet.

Selv om vi ikke fikk laget en prototyp av dette hovedprosjektet mener vi at vi har tilegnet oss mye kunnskap og at vi har laget en god plattform for videreutvikling av prosjektet. Det var veldig mye nytt å sette oss inn i og det tok en del tid. Dette prosjektet har generert en god del materie, mengden av programkode som ligger som vedlegg er et eksempel på dette. Vi hadde en god del problemer med USB, og det gjorde at Gant-skjemaet vårt sprakk ganske fort. Vi har brukt mer tid på prosjektet enn det vi hadde satt opp, men det var vårt første måte med et større prosjekt og gangen i det, så en kan ikke forvente at alt skal gå så glatt da.

6.0 Litteraturliste

Teori om eCos: Mye informasjon hentet fra ”Embedded Software Development with eCos” av Anthony Massa, som blant annet kan finnes her:
http://www.informit.com/content/downloads/perens/0130354732.pdf.

Teori om Nios II: En del funnet i Nios II dokumentasjonen i dokumentet n2cpu_nii5v1.pdf og noe annet på http://www.fpgajournal.com/articles/20040518_nios2.htm.

Teori om USB: En del informasjon er funnet i USB Complete 2. edition av Jan Axelson. Og noe i USB Design by Example av John Hyde. Samt en god del på www.beyondlogic.com.

Teori om GUI programmering og wxWindows programmering ble funnet på www.wxwindows.org.

Deklarasjoner for Windows funksjoner hentet vi fra
http://msdn.microsoft.com/library/default.asp.

Ved programmering og oppsett av Nios-systemet har den dokumentasjonen som følger med Nios II vært uvurderlig. Disse dokumentene ligger under documents-katalogen i en Nios installasjon. Hvis en ikke har tilgang til en Nios installasjon finnes den samme dokumentasjonen på http://www.altera.com/literature/lit-index.html spesielt under Nios II Processor linken.

Programmering av GAUCE applikasjonen hadde nok ikke vært mulig uten at vi hadde lest eCos Reference Manual på http://ecos.sourceware.org/docs-latest/ref/ecos-ref.html. Denne manualen installeres også samtidig med eCos under \ecos-current\doc\html.