Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Ken Robin Dugan, Magnhild Eeg, Simen Løcka Eine

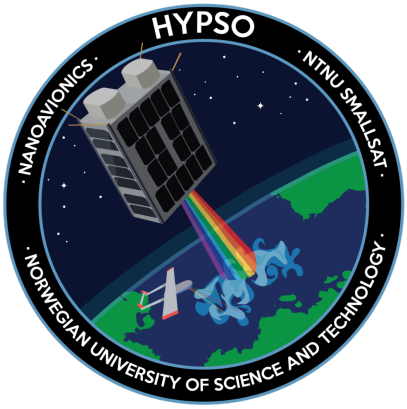
Implementering av Cubesat Space Protocol over UART i programvare for minisatellitt

Bacheloroppgave i Elektroingeniør
Veileder: Dominik Osinski
Medveileder: Roger Birkeland
Mai 2022

NTNU
Kunnskap for en bedre verden

Ken Robin Dugan, Magnhild Eeg, Simen Løcka Eine

Implementering av Cubesat Space Protocol over UART i programvare for minisatellitt



Bacheloroppgave i Elektroingeniør
Veileder: Dominik Osinski
Medveileder: Roger Birkeland
Mai 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

NTNU
Kunnskap for en bedre verden

1 Bacheloroppgave

Oppgavetittel:
Implementering av Cubesat Space Protocol over UART i programvare for minisatellitt
Thesis title:
Implementation of Cubesat Space Protocol over UART in small satellite software

Forfattere:
Ken Robin Dugan
Magnhild Eeg
Simen Eine

Prosjektnummer: E2219
Innleveringsdato: 20.05.2022
Gradering: [X] åpen [ ] lukket

Studium:	Elektroingeniør BIELEKTRO
Studieretning:	Elektronikk og Sensorsystemer
Veileder internt:	Dominik Osinski
Institutt:	Institutt for Elektroniske Systemer
Oppdragsgiver:	NTNU Small Satellite Lab (SmallSat)
Kontaktperson:	Roger Birkeland

Sammendrag:
Denne bacheloroppgaven er skrevet for NTNU Small Satellite Lab (SmallSat). Oppgaven beskriver hvordan nettverksprotokollen "Cubesat Space Protocol" (CSP) kan sendes over et fysisk UART/RS-422-grensesnitt ved å implementere datalinkprotokollen "Keep It Simple, Stupid" (KISS) i programvaren til satellittene HYPSO-1 og HYPSO-2. Oppgaven har som hensikt å forbedre overføringsraten internt på HYPSO-2 ved å implementere den raskere databussen RS-422 i parallell med den nåværende CAN-databussen. Resultatet fra bacheloroppgaven er et fungerende tillegg i programvaren hypso-SW som åpner for seriell kommunikasjon ved bruk av KISS over RS-422. Rapporten inneholder detaljerte forklaringer av funksjonaliteten til koden og beskriver hvordan kodeendringene skal integreres inn i HYPSO-prosjektet. Teori og foreløpige tester indikerer at endringen i programvare vil forbedre overføringsraten både i det fysiske laget og i datalinklaget. Rapporten er skrevet for å være praktisk nyttig for fremtidige medlemmer av SmallSat. Det skal være mulig å slå opp i de individuelle kapitlene for å lese om enkelttemaer i bacheloroppgaven.

Summary:
This bachelor's thesis is written for NTNU Small Satellite Lab (SmallSat). The thesis outlines how the "Cubesat Space Protocol" (CSP) can be sent over a UART/RS-422 interface by implementing the data link layer protocol "Keep It Simple, Stupid" (KISS) in the software of the HYPSO-1 and HYSPO-2 satellites. The intention of the bachelor's thesis is to improve the data rate of HYPSO-2 by adding the high-speed RS-422 data bus and the currently used CAN bus in parallel. The result of the bachelor's thesis is a working addition to the HYPSO software that enables serial communication by using KISS over RS-422. This thesis contains detailed explanations of the additions to the code and instructions on how to integrate the work into the rest of the HYPSO project. Preliminary test results indicate that the changes made in the software will improve the data rate in both the physical layer and the data link layer. The text is written to be practically useful to the other members of the SmallSat team. The intention is that other team members should be able to look up in individual chapters of interest without needing to read the entire thesis.

Stikkord:
RS-422, CSP, CAN, UART, integrerte systemer, datarate, minisatellitt.

Keywords:
RS-422, CSP, CAN, UART, embedded systems, data rate, small satellite.

Innhold

Figurer

Tabeller

2 Innledning

HYPSO-2 er en minisatellitt under utvikling av NTNU Small Satellite Lab (SmallSat). Designet og funksjonaliteten til HYPSO-2 bygger på den tidligere satellitten HYPSO-1 som ble skutt opp i bane i januar 2022 (Grøtte, 2021). HYPSO-2 skal etter planen skytes opp i 2024 og skal ta hyperspektrale bilder av algekolonier i havet. En av endringene i HYPSO-2 sammenliknet med HYPSO-1 er tillegg av en ekstra seriell buss internt i satellitten for lokal dataoverføring. HYPSO-1 benytter seg av én enkelt seriell CAN-buss for intern overføring av hyperspektrale bildefiler, mens HYPSO-2 skal bruke en seriell RS-422-buss i parallell med den eksisterende CAN-bussen. Bacheloroppgaven denne rapporten bygger på består av å forbedre den interne overføringsraten på HYPSO-2 ved å inkludere seriell kommunikasjon over RS-422 i programvaren til satellitten. Rapporten har som hensikt å skape en konkret nytteverdi for nåværende og fremtidige medlemmer av SmallSat-teamet ved å detaljert beskrive fremgangsmåte og designvalg i arbeidsprosessen i bacheloroppgaven. Rapporten skrives på et faglig nivå som skal kunne la en annen bachelorstudent med elektronikkbakgrunn lese og forstå innholdet. For ordens skyld spesifiseres det at "gruppen" herifra refererer til bachelorgruppen og "teamet" refererer til SmallSat-teamet eller software-teamet innad i SmallSat.

3 Leserveiledning

Rapporten er skrevet i et ikke-tradisjonelt format for å både gjøre den lett å bruke av fremtidige medlemmer av SmallSat og for å få med alt relevant akademisk innhold. Dette er i kontrast med det tradisjonelle rapportformatet med separate kapitler for teori, metode, resultater og drøfting. Det viktigste og mest praktisk nyttige innholdet for resten av SmallSat-teamet står under "Problemstilling", "Hva prosjektet har resultert i", "Programmering" og "Integrasjon". Resten av innholdet er utdyping, forklaring og drøfting om det som står i disse kapitlene. Det kan også være interessant for fremtidige medlemmer av SmallSat-teamet å lese underkapitlene i drøftingsdelen som er merket med "Veien videre:". Disse underkapitlene tar for seg det gruppen vurderer som naturlige steg videre i arbeidet med seriell kommunikasjon i HYPSO-2. Underkapitlene skal være navngitt tydelig nok til at det skal kunne gå an å slå opp de temaene en er interessert i å lese om uten å måtte lese gjennom hele rapporten. Det er også forsøkt å holde temaene samlet i hvert sitt kapittel slik at en ikke trenger å hoppe mellom kaptilene for å lese om et spesifikt tema.

4 Problemstilling

HYPSO-1 bruker CAN-buss for å overføre hyperspektrale bilder fra On-board Processing Unit (OPU) til Payload Controller (PC). Dette utgjør en flaskehals fordi kombinasjonen av de forskjellige protokollene som kjører på CAN-bussen gjør dataoverføringen mye tregere enn tidligere antatt. Den trege overføringsraten gjør at satellitten bruker unødvendig mye energi for å overføre filer. Ved å legge til en RS-422-buss i parallell med CAN-bussen, kan man bruke den raskere RS-422 til å overføre store hyperspektrale bildefiler og CAN til annen kommunikasjon slik at filoverføringen går raskere og effekttapet blir mindre (se vedlegg G og H). For å kunne sende data over RS-422 brukes datalinkprotokollen "Keep It Simple, Stupid", ellers kjent som KISS. Bacheloroppgaven går ut på å redigere kode i programvaren hypso-SW for å implementere KISS og dermed åpne for seriell kommunikasjon over RS-422.

4.1 Arbeidets art og omfang

Denne bacheloroppgaven er en mindre oppgave innad et større tverrfaglig ingeniørprosjekt. Dette har gjort at gruppen har vært begrenset til en del betingelser satt av andre deler av prosjektet. Det er blant annet begrensninger knyttet til både hardwaredesign og softwaredesign. Et eksempel på dette er valget av RS-422 som seriell buss. Dette er bestemt av payload controlleren som

allerede har RS-422-grensesnitt integrert (NanoAvionics, 2018). Rapporten tar for seg de aktive valgene gruppen har hatt anledning til å ta, hovedsakelig innenfor programvareutviklingen.

Målet med oppgaven er å produsere et fungerende tillegg i programkoden til HYPSO-2. Selve hovedbidraget i den endelige programkoden er relativt kort. Hovedutfordringen kommer av at satellitten som programvaren skal kjøre på ikke eksisterer enda. Dermed går mye av bachelorprosjektarbeidet ut på å utarbeide måter å teste programvaren slik at en med sikkerhet kan vite at den fungerer uten problemer på target hardware, - før target hardware er klart. I tillegg har SmallSat valgt å bruke en metode for utvikling der utbedringer av programvaren til HYPSO-2 blir gjort på den eksisterende og fungerende programvaren til HYPSO-1 i stedet for å lage to separate grener av programvaren. Dette betyr at all endring i koden må være bakoverkompatibel og ikke forstyrre funksjonaliteten til HYPSO-1. En relativt stor del av prosjektet består av integrering og sammenfletting av gruppens kode med resten av filsystemet til hypso-SW. En stor del av arbeidstiden i bachelorprosjektet består av samarbeid og kommunikasjon i det større SmallSat-teamet.

Fordi programvaren skal kjøre på en satellitt som ikke skal skytes opp før i 2024, og medlemmer i SmallSat-teamet byttes ut hvert semester, er det ekstremt viktig å forsikre seg om at programvaren som utvikles nå er modulær, fungerer som den skal, og ikke har feil eller mangler. Det er et stort problem for SmallSat dersom kode som bare fungerer delvis blir inkludert i hypso-SW og delene som står igjen å utvikle og forbedre blir glemt til neste gang noen skal arbeide med stoffet. Det er uvisst når neste person skal jobbe med intern dataoverføring på satellitten og hva som vil bli jobbet videre med, og dette er grunnen til at oppgaven har så stort fokus på testing av det ferdige produktet. Gruppen må forholde seg til muligheten for at det ferdige bidraget i programvaren blir brukt i satellitten og at det kan bli problematisk å endre eventuelle feil og mangler når satellitten er i bane. Det ferdige bidraget må altså være så komplett, ferdig testet, detaljert dokumentert og så oversiktlig å sette seg inn i som mulig.

4.2 Kilder og informasjonsinnhenting

I teorikapitlet og drøftingskapitlet henvises det til literatur og offentlig dokumentasjon. Fordi denne prosjektoppgaven er utarbeidet i et større ingeniørteam med tett samarbeid og oppfølging, er en del innhold i de andre kapitlene ikke støttet av offentlig tilgjengelig dokumentasjon. I stedet kommer dette innholdet fra møter, diskusjoner, workshops, chatmeldinger og issues på GitHub. Denne typen kilde lar seg ikke føre opp like lett i referanselister og antas som allment kjent og tilgjengelig informasjon innad i SmallSat. Noe kommunikasjon fra diskusjoner på GitHub er lagt som vedlegg i slutten av rapporten og refereres til i teksten der dette er naturlig. Disse vedleggene er kun for å peke til spesifikke diskusjoner og gir ikke et helhetlig bilde av kommunikasjonen og arbeidet i bachelorprosjektet. Det anbefales derfor at lesere av rapporten henvender seg til SmallSat for å få tilgang til hypso-SW på GitHub dersom en ønsker et dypere innblikk i vurderinger og dokumentasjon fra arbeidsprosessen.

5 Hva prosjektet har resultert i

Prosjektet har resultert i et tillegg i programvaren til opu-services som åpner for kommunikasjon over KISS, som igjen åpner for seriell dataoverføring over RS-422. Utviklingen av dette tillegget i koden har vært avhengig av virtuell testing av seriell kommunikasjon. Det er først laget et program for å teste seriell kommunikasjon i et lokalt virtuelt testmiljø, og for å gjøre seg kjent med relevante verktøy og bibliotek. Deretter er det laget et tillegg i koden til programvaren til bakkestasjonen, hypso-cli, for å inkludere kommunikasjon over KISS. Den serielle kommunikasjonen med hypso-cli blir testet med det nydesignede python-scriptet. Deretter blir de nødvendige endringene i OPUen lagt til programvaren opu-services, og den serielle kommunikasjonen blir testet mellom opu-services og hypso-cli. Det er også laget testscript for å automatisere testingen av programvaren. En demonstrasjon av testscriptet er lastet opp og integrert i GitHub Actions i en testbranch i hypso-SW. De konkrete stegene i arbeidsprosessen er forklart i kapittel 8, "met-

ode og testing", og endringene i programvaren er detaljert beskrevet i kapittel 9, "programmering".

Gruppen har også oppdatert biblioteket libcsp i hypso-SW. Resultatet av dette er at SmallSat nå bruker et vedlikeholdt bibliotek som også vil være lettere å vedlikeholde videre. Dette nye bibilioteket har inkludert mer feilhåndtering og bug fixes som både bachelorgruppen og resten av SmallSat-teamet har måttet fikse manuelt i den gamle versjonen av biblioteket. I tillegg gjør denne oppdateringen at byggingen av biblioteket skjer på en sikrere måte. Denne oppdateringen står detaljert forklart i kapittel 10, "oppdatering av libcsp".

I tillegg til utvidelsen av programvaren i hypso-SW, har prosjektgruppen undersøkt og drøftet forskjellige alternativer til veien videre i arbeidet med seriell kommunikasjon i HYPSO-2. Dette er beskrevet i metodedelen og drøftingsdelen (kapittel 8 og 12) i denne rapporten. Gruppen har kommet med konkrete forslag til hvordan den serielle dataoverføringen videre kan optimaliseres fra et softwareperspektiv. I tillegg er det laget en detaljert oppskrift for videre testing når riktig hardware er på plass. Det er også laget forslag til hva gruppen anser som naturlige steg videre dersom noen andre medlemmer av SmallSat-teamet skal videreutvikle arbeidet med dataoverføring.

6 Teoretisk grunnlag

Dette kapitlet inneholder informasjon som skal gjøre det lettere å forstå bachelorprosjektet for utenforstående. Det antas at medlemmer innad i SmallSat allerede har bakgrunnskunnskapen for å forstå innholdet i de senere kapitlene uten å måtte lese gjennom denne teoridelen.

6.1 Bakgrunn

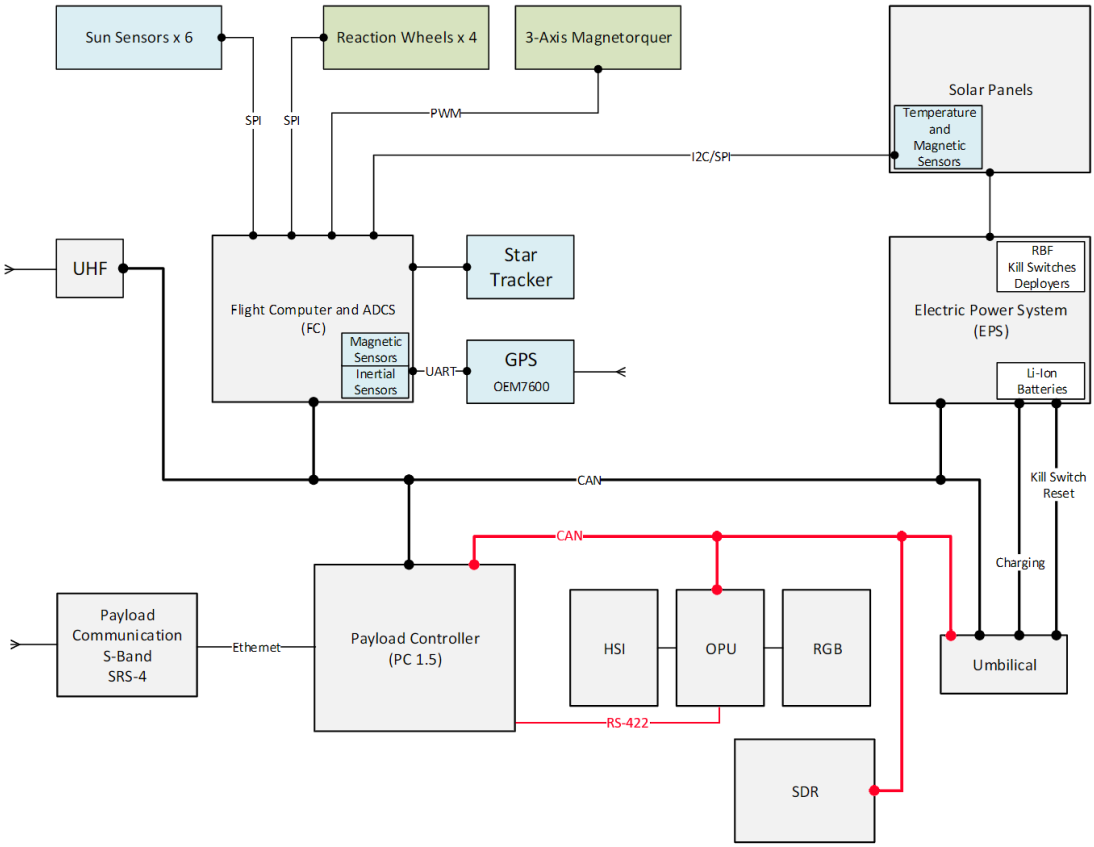
CAN-bussen som brukes på HYPSO-1 har en datahastighet på 1 Mb/s. Dette ble antatt at skulle være tilstrekkelig overføringshastighet da satellitten ble designet. Det viser seg at kombinasjonen av protokollene som kjører oppå hverandre og formatet på dataen bidrar til rundt 60-70% tap i overføringsrate (se resultater av hardwaretest i kapittel 8.5.1). Mye av dette skyldes overhead fra de forskjellige protokollene. Overhead er bits i en datapakke som er nødvendig for dataoverføringen, men som ikke inneholder noe av selve informasjonsdataen som en ønsker å sende. Dette kan for eksempel være headere og checksums (Cavanaugh, 1994). Det vil si netto dataoverføring over CAN bare er ca. 30-40% av den først antatte hastigheten. Fordi Payload Controlleren fra Nano Avionics allerede har integrert RS-422 grensesnitt (NanoAvionics, 2018), er det et naturlig valg å benytte seg av RS-422 for å forsøke å forbedre dataraten. Det er ikke gjort noe forsøk på å redusere mengden overhead, men selv om det å implementere RS-422 skulle bidratt til like mye overhead, kan selve databussen presses til å ha en høyere datahastighet slik at netto dataoverføring likevel blir forbedret (NovusAutomation, 2022). Det vil si at dersom RS-422 ender opp med å kjøre på 2-3 Mb/s med 60-70% overhead, vil en likevel se en 2-3 ganger forbedret hastighet.

SmallSat har ikke gjort noe forsøk på å redusere overhead ettersom dette sannsynligvis vil kreve endring i selve arkitekturen til satellitten. En kan se for seg at dersom en skulle laget et system fra bunnen av, kunne en laget en mye raskere direkte datalink uten alle protokollene og overhead, men ettersom mange av hardwarekomponentene kommer fra Nano Avionics med en del ferdige innstillinger og funksjonaliteter, er det naturlig å forholde seg til de rammene dette setter.

I tillegg til at den trege overføringen over CAN sørger for stort effekttap, utgjør dette også et problem for operatørene på bakkestasjonen. Overføringen mellom OPU og PC tar ca. 40 minutter (Erfaring fra bildeoverføring, Mai 2022) og i denne tidsperioden kan ikke overføringen avbrytes fordi PC ikke har mulighet til å sette dataoverføringen på pause og gjenoppta den samme filen etter avbrudd. PC har heller ikke mulighet til å laste opp og laste ned data samtidig i samme program. Alt dette betyr at overføringen og nedlastingen av bilder krever mye planlegging og

hensyn når en rotasjon av satellitten rundt jorden tar ca. 90 minutter. Det betyr at det alltid må settes av minst 40 minutter før en passering over en bakkestasjon der ingenting annet kan gjøres for å kunne laste ned en bildefil. Det å redusere overføringstiden til 10-15 minutter ved å benytte en databuss som er 2-3 ganger raskere, vil gjøre det mye lettere for operatørene å planlegge og gjennomføre bildetaking og nedlasting av bilder. I tillegg vil det være mulig for annen data å sendes over CAN-bussen samtidig som de store filene overføres over RS-422.

6.2 HYPSO-2



Figur 1: Blokkskjema HYPSO-2. Bildet er hentet fra HYPSO-2-dokumentasjon internt i SmallSat

Figur 1 illustrerer et foreløpig blokkskjema over HYPSO-2. Nederst kan en se Payload Controller (PC) og OPU som er koblet sammen med både CAN-buss og RS-422, markert i rødt. OPUen skal overføre hyperspektrale bildefiler over RS-422 til PC. Bildefilene blir trådløst overført til bakkestasjonen fra PC via S-band trådløs kommunikasjon. CAN-bussen mellom OPU og PC skal stå for annen type kommunikasjon enn bildeoverføring. En kan se at SDR (Software Defined Radio) også er koblet til den samme CAN-bussen. Dette betyr at den trege hastigheten på CAN-bussen også påvirker effekttapet i kommunikasjonen mellom SDR og PC. Den andre CAN-bussen i satellitten brukes ikke til hyperspektral bildeoverføring og utgjør dermed ikke en flaskehals på samme måte, så det har ikke vært nødvendig å øke kapasiteten på denne andre CAN-bussen. Det er altså bare den ene CAN-bussen (markert i rødt) mellom PC, OPU og SDR som denne bacheloroppgaven har arbeidet med.

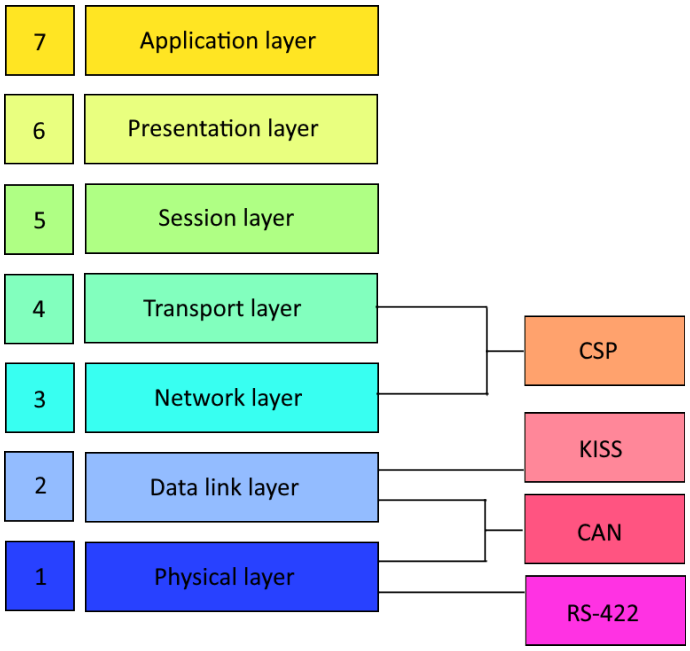
6.3 Om protokoller

CSP (Cubesat Space Protocol) er en nettverksprotokoll spesielt designet for å brukes i minisatellitter. CSP fungerer veldig likt som IP-protokoll, men med en kortere header og et mindre adresseringsfelt ettersom det er færre destinasjoner innad i en satellitt å sende data til. Libcsp er et bibliotek som inneholder alt av kode som trengs for å bruke CSP i programvare (J. D. C. Christiansen, 2021).

KISS er en datalinkprotokoll opprinnelig laget for å forenkle oppsett av innvevde systemer innenfor amatørradio. Hensikten med KISS er å la brukerprogramvaren få kontroll over kommunikasjonen i så lave protokoll-lag som mulig slik at dette ikke trengs å hardkodes inn i de innvevde systemene. KISS er bygget på rask, asynkron overføring av små datapakker uten noen mer avansert funksjonalitet. Protokollen fungerer som en datastrøm der datapakkene blir sendt uten toveishåndtrykk, ACKs, checksums eller control flow. Dette håndteres av protokoller i de høyere lagene. Dette betyr at KISS hovedsakelig bare sender dataen så raskt som mulig uten noen buffering, feilhåndtering eller databehandling (Chepponis and Karn, 1987).

CAN (Controller Area Network) er en databuss der flere komponenter er koblet til samme hovedledning/hovedbuss. Hver datapakke som sendes på databussen har en adresse. Alle transceivere lytter på databussen og sjekker om datapakken som blir sendt er adressert til dem. Transceiverene ignorerer datapakken dersom den ikke er adressert til dem. Bare en datapakke kan sendes over CAN-bussen av gangen. Headeren i hver datapakke inneholder informasjon om prioritet. Dersom to sendere prøver å sende datapakker samtidig, vil en av datapakkene vil ha prioritet over den andre og sendes først. Senderen med datapakken med lavere prioritet må vente til databussen er ledig før den kan prøve å sende på nytt. (Kvaser, 2022).

Det er CAN-protokollen som bestemmer hvordan transceiverne skal oppføre seg når data sendes over CAN-bussen. Det å vente på riktig adressert data og å vente i kø på å sende data er en del av protokollen og er ikke nødvendigvis et resultat av selve oppkoplingen. CAN i sin helhet er dermed en kombinasjon av både fysisk oppkopling med differensielle signalledninger og en protokoll.



Figur 2: OSI-modellen med relevante protokoller

OSI (Open Systems Interconnection) er en modell for å beskrive strukturen i datakommunikasjon, protokoller og hardware. Modellen deler nettverk inn i lag der tanken er at de øvre lagene ikke skal kunne "merke" hvordan de nedre lagene opererer (Øverby, 2021). CAN er en blanding av det fysiske laget og datalinklaget. KISS er en del av datalinklaget og CSP er en blanding av nettverkslaget og transportlaget. Grunnen til at noen av protokollene kan være en blanding av to lag er at nettverket i satellitten er relativt enkelt sammenliknet med f.eks. internett. Enkelte protokoller kan dermed utføre flere arbeidsoppgaver enn de ville gjort i et internettnettverk på bakken.

Det finnes mange illustrasjoner og metaforer for å beskrive hvordan OSI-modellen kan anvendes. En forenklet illustrasjon som kan forklare hvordan CAN, KISS og CSP fungerer kan være "konvolutter" og "brev". En kan se for seg protokoller nedover i hierarkiet i OSI-modellen som konvolutter som puttes inn i hverandre.

Hver gang man beveger seg ett nivå ned i modellen og benytter seg av en ny protokoll, legger man det ferdige brevet inn i en ny konvolutt og skriver en annen type adresse på den.

KISS og CAN-protokollen er som to konvolutter som skal kunne ha like stor plass til en type brev, CSP. KISS-konvolutten skal sendes over RS-422, mens CAN-konvolutten skal sendes over CAN. De fysiske databussene kan sees på som forskjellige "postselskap" med hver sin konvolutt-standard. Hvert postselskap kan bare bruke sin egen konvolutt-standard, men innholdet i konvoluttene kan være likt.

6.4 Socat

Socat er et verktøy som etablerer data-strømmer mellom to spesifiserte adresser. I dette prosjektet blir socat brukt til seriell kommunikasjon mellom virtuelle porter. En virtuell port er en type programvare som ser ut som en virkelig port, f.eks. en USB-port, i den forstand at brukerprogrammet kan lese og skrive data til porten som vanlig, men at dataen i virkeligheten blir håndtert av et annet program enn selve porten. Et nytt terminalvindu i Linux oppretter en ny virtuell port, men dataen som blir sendt inn og ut av porten/terminalvinduet blir ikke behandlet i selve vinduet (Rieger, 2022), (Luo, 2014).

6.5 Bygging av programvare

Programvaren i hypso-SW er hovedsaklig skrevet i C. Denne programvaren må kompileres og linkes til de riktige avhengige bibliotekene. I et større prosjekt som hypso-SW er dette en komplisert prosess som involverer spesifikke verktøy og konfigurasjoner. For at denne prosessen skal gå lettere og resultatet skal være likt for alle utviklere brukes et byggesystem. Byggesystemet gir en oppskrift på hvordan programvaren skal kompileres og hvordan den skal settes sammen. Byggesystemet kan også stille krav til for eksempel nødvendige biblioteker og versjoner av verktøy som brukes i byggeprosessen (Kitware, 2022).

Det finnes flere ulike byggesystem, og større prosjekter må ofte kunne støtte flere byggesystem for avhengige biblioteker. I SmallSat brukes byggesystemet CMake, som sørger for å konfigurere, bygge og eventuelt installere programvare. I hypso-SW er det definert oppskrifter for at CMake skal kunne sette sammen programvaren riktig. SmallSat er avhengig av biblioteket libcsp, som igjen bruker et annet byggesystem som heter Waf. Waf bruker Python til å skrive oppskrifter og kjøre byggingen (Nagy, 2021).

7 Prosjektering og programvareutvikling

Dette kapitlet beskriver metoder og aspekter i prosjektarbeidet som direkte har påvirket resultatet i bachelorprosjektet. Dette kapitlet er nyttig for å forstå bakgrunnen for en del valg som er tatt i utviklingen av prosjektet, men er ikke nødvendig for å forstå selve innholdet og resultatet av arbeidet.

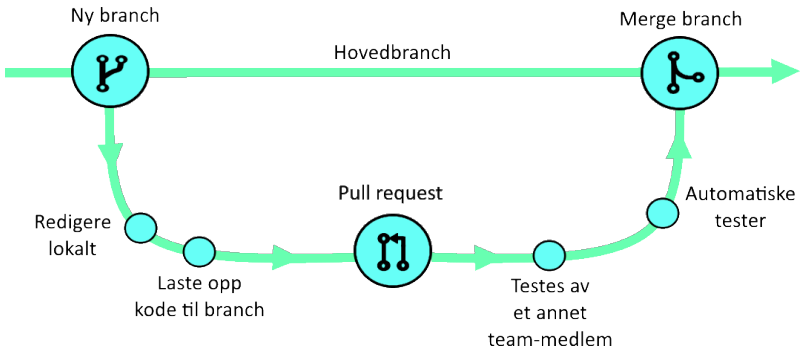
7.1 Bakoverkompatibilitet

Bakoverkompatibilitet (backwards compatibility) er et konsept der teknologi kan brukes sammen med tidligere versjoner/generasjoner av teknologien (Techopedia, 2012). I programvareutvikling betyr dette at nye versjoner av koden f.eks. skal kunne bruke samme typer grensesnitt, filtyper og data som i de eldre versjonene av koden. Det er lettere å skrive bakoverkompatibel kode om filstrukturen og koden er modulær og oversiktlig.

SmallSat bruker GitHub som arbeidsplattform for deling av kode og programvare. Per i dag er programvaren for HYPSO-1 og HYPSO-2 den samme og ligger i samme GitHub repository. Det

vil si at programvaren til HYPSO-1 kjører på den operative satellitten samtidig som utviklerne i software-teamet videreutvikler den samme programvaren til å få utvidet funksjonalitet når den lastes opp på HYPSO-2. Dette er et bevisst valg som ble tatt tidlig av SmallSat og begrunnelsen er at størrelsen på SmallSat-teamet og den konstante utskiftingen av folk som arbeider på SmallSat-prosjektet ikke gir kapasitet til å vedlikeholde to separate repositories med nesten lik kode. I dette tilfellet er bakoverkompatibilitet ekstremt viktig for at HYPSO-1 fortsatt kan være operativ selv med nye versjoner av programvaren. I løpet av bachelorprosjektet blir programvaren til bakkestasjonen, hypso-cli, og opu-services utvidet til å inkludere grensesnitt og kommunikasjon med en protokoll som ikke brukes på HYPSO-1, men dette skal ikke påvirke den videre funksjonaliteten til programvaren når den kjører på HYPSO-1. Det har blant annet vært viktig for gruppen å begrense omfanget av utviklingen av ny kode til så enkelt og rett frem som mulig. På et tidspunkt vil man måtte bryte bakoverkompatibiliteten. Dette skjer der stabilitet ikke kan garanteres, eller når tillegg av ny kode ikke kan gjøres uten en urimelig mengde ekstraarbeid.

7.2 Kvalitetskontroll



Figur 3: Oversikt over arbeidsmetode med programvareutvikling i HYPSO-prosjektet

SmallSat bruker GitHub som arbeidsverktøy for å dele kode og samarbeide med programutvikling. En main branch av programvaren vedlikeholdes og alle endringer i koden gjøres ved at det lages en utviklingsbranch av kodefilene til hovedbranchen. Deretter redigeres koden lokalt på utviklingsbranchen før det lages en pull request for å sammenflette kodeendringene i utviklingsbranchen med hovedbranchen. Alle pull requests må testes og godkjennes av en annen person i softwareteamet før de kan merges. Dette er et viktig steg i kvalitetskontrollen av programvaren, men det er også med på å forlenge tiden det tar å få ferdigstilt ny programvare. Denne typen kvalitetskontroll der koden gjennomgås av noen andre bidrar også til et dynamisk samarbeid der det noen ganger blir vurdert at det er verdt å godkjenne pull requests selv om den nye programvaren ikke består alle tester.

I tillegg til branches er det nyttig å gjøre seg kjent med forks i GitHub. Forskjellen på en branch og en fork er at en utviklingsbranch lagres innad i samme repository som hovedbranchen, mens en fork kopierer et helt repository som lagres separat. Dette inkluderer alle branches innad i repoet. I bachelorprosjektet har gruppen brukt utviklingsbranches til å utvikle kode og arbeidet med forks i forbindelse med oppdateringen av libcsp (se kapittel 10).

7.3 Scrum og utforming av prosjektmoduler

Scrum er en arbeidsmetode for kompleks programvareutvikling i tverrfaglige utviklerteam. Scrum-formatet består av tre hovedkomponenter: produkteier, scrum-master og scrum-team. Produkteier sørger for å instruere scrum-team og scrum-master i hva som skal utvikles. Videre er det opp til scrum-teamet å bestemme hvordan oppgavene skal løses på best mulig måte. Teamet arbeider tett i 1-4-ukers lange sprinter og har frihet og selvstendighet til å velge prioritet og

rekkefølge på oppgavene. Etter endt sprintperiode gjennomgås fremdriften og planlegges en ny sprintperiode. Det er scrum-master som skal overse sprintplanleggingen og legge til rette for det dynamiske samarbeidet i scrum-teamet (scrum.org, 2022).

Utformingen av bachelorprosjektet blir påvirket av scrum-formatet som benyttes innad i softwareteamet i SmallSat og et scrum-liknende format i prosjektgruppen. Scrum forutsetter at arbeidsteamet er selvstyrt og kan vurdere selv hva det har kapasitet til i løpet av sprintperioden. Det er blitt benyttet sprintperioder på to uker, samt flere ukentlige og semi-ukentlige møter med SmallSat-teamet, og dette har bidratt til å gjøre arbeidsoppgavene i bachelorarbeidet modulære og stegvise. En kan se for seg at dersom scrum ikke hadde blitt benyttet, ville bacheloroppgaven sannsynligvis bli utformet på en mer massiv måte der det sannsynligvis ville blitt brukt mindre tid på testing og utarbeiding av testmiljøer og mer tid på å utvikle programvaren i opu-services direkte. Fordi scrum-formatet har drevet frem utforming av arbeidspakker med en til to uker estimert arbeidstid, har dette sørget for en jevnere tidsfordeling også på de enklere og "mindre kritiske" arbeidsoppgavene, som utvikling av script for virtuelle porter. Utviklingene av arbeidspakkene har skjedd i samarbeid med resten av SW-teamet.

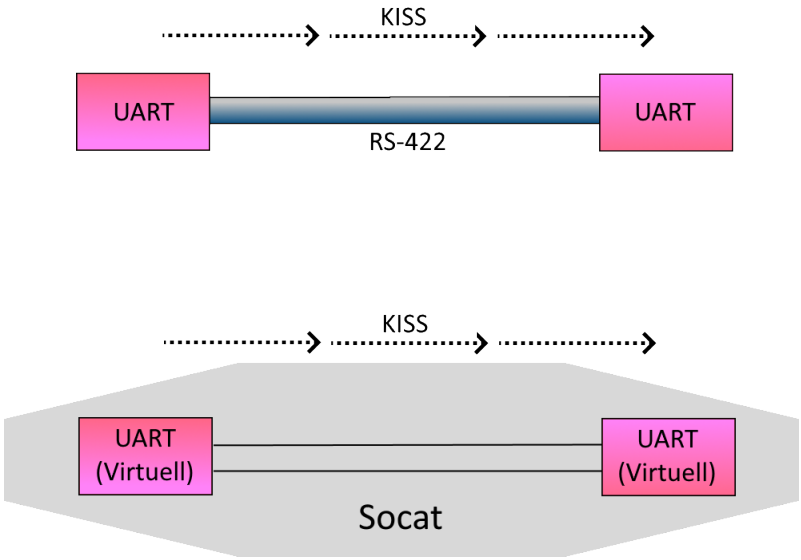
8 Metode og testing

Prosjektutviklingen i bachelorprosjektet har vært modulær og foregått i fem trinn. Det er trinn 3 som dreier seg om å løse den faktiske problemstillingen i bachelorprosjektet ved å la opu-services kommunisere over UART, men de tidligere trinnene har vært nødvendig for å teste at kommunikasjonen faktisk fungerer. Trinnene etter trinn 3 dreier seg om videre testing for kvalitetskontroll.

8.1 Trinn 1: Virtuelle porter og oppsett av testmiljø

For å kunne utvikle programvare i SmallSat-prosjektet, var gruppen avhengig av å sette opp et utviklingsmiljø hvor programvare kunne utvikles og testes (se vedlegg I og J).

For å kunne teste seriell kommunikasjon over CSP med bruk av biblioteket libcsp lages det et testprogram som sender CSP-meldinger over KISS. Dette testprogrammet baseres på eksempelkode fra libcsp og det testes med virtuelle porter gjennom socat. Testprogrammet viser at KISS over virtuelle porter fungerer som forventet, og gjør gruppen kjent med CSP-biblioteket før videre utvikling begynner.



Figur 4: Illustrasjon av fysisk seriell kobling og virtuell seriell kobling

For å lettere forstå sammenhengen mellom alle uttrykkene i forbindelse med seriell kommunikasjon, kan en ta en titt på illustrasjon 4. UART er integrert hardware i sender- og mottakerkomponentene som konverterer generell data til asynkron seriell data. RS-422 er den fysiske ledningen mellom to UART-enheter. Den øverste illustrasjonen viser hvordan seriell kommunikasjon fungerer i den fysiske satellitten. Den nederste illustrasjonen viser hvordan seriell kommunikasjon fungerer i det virtuelle testmiljøet. Socat er et program som både oppretter virtuelle porter og en virtuell link mellom portene. De virtuelle portene kan sees på som UART-enheter, bortsett fra at de er virtuelle, ikke fysiske hardwarekomponenter. KISS sender datapakker serielt over et medium som støtter asynkron seriell kommunikasjon. Så lenge en har en overføringsmetode som støtter UART, har det ingenting å si for KISS-protokollen hva den blir sendt over.

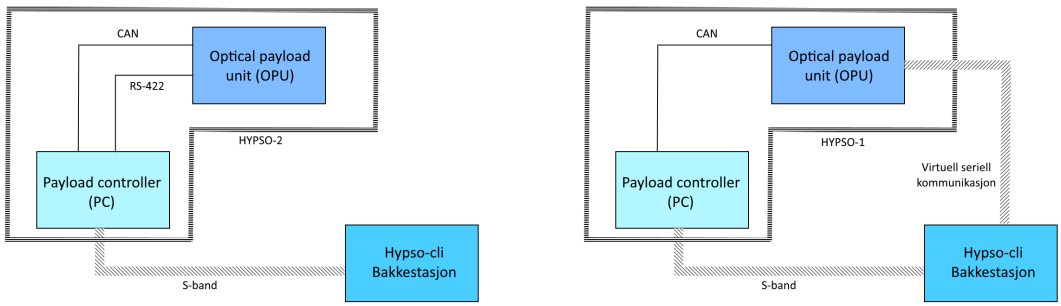
8.2 Trinn 2: Hypso-cli

Hypso-cli er programvaren til bakkestasjonen. Denne programvaren har ingen funksjonalitet som er direkte avhengig av kommunikasjon over KISS. Likevel går trinn 2 ut på å utvide hypso-cli til å kunne kommunisere over KISS (se vedlegg L og P). Det er tre grunner til dette:

1. Det må legges et kunnskapsgrunnlag om hvordan CSP og KISS er tatt i bruk i hypso-SW. Hypso-cli har relativt oversiktlig struktur, så det er et godt sted å stårte for å gjøre seg kjent med libcsp og KISS.
2. Testing av trinn 3, utviding av opu-services. I utgangspunktet skal all nyttelasten på satellitten kun respondere på kommandoer sendt fra hypso-cli. Hvis hypso-cli kan kommunisere over KISS med enkeltkomponenter på nyttelasten, gjør dette debugging og testing mye enklere enn å sende all kommunikasjon fra hypso-cli over S-band først.
3. Verktøy for videre feilsøking og utvikling av seriell kommunikasjon. Det et sannsynlig at det på et senere tidspunkt blir arbeidet videre med seriell kommunikasjon på satellitten, og det å ha en permanent mulighet til å debugge ved å koble kommunikasjon over KISS direkte til hypso-cli vil derfor være en fordel.

8.3 Trinn 3: Opu-services

Trinn 3 består av å utvikle opu-services til å forstå KISS-protokollen og å kunne sende og motta data over UART. Deretter testes dette tillegget i koden ved å sette opp virtuelle porter mellom opu-services og hypso-cli og sende data mellom disse over KISS (se vedlegg K og Q).



(a) Blokkskjema over relevant kommunikasjon i HYPSO-2

(b) Oversikt over testkommunikasjon

Figur 5: Blokkskjema for kommunikasjonskomponenter og testoppsett

Figur 5a viser hvordan de forskjellige komponentene og programvaren skal kommunisere med hverandre i HYPSO-2. Den serielle kommunikasjonen over KISS går kun mellom OPUen og PCen. HYPSO-cli kommuniserer med PCen over S-band. HYPSO-cli kommuniserer ikke med OPU direkte.

Figur 5 viser hvordan de forskjellige komponentene er blitt koblet sammen i forbindelse med testingen i prosjektarbeidet. HYPSO-cli og opu-services kobles sammen over virtuelle porter for å teste at OPUen kan opprette et KISS-grensesnitt og pålitelig kommunisere serielt over KISS. Dette oppsettet brukes dermed bare i testingen, ikke ved ordinær drift når satellitten er operativ.

8.4 Trinn 4: Softwarebasert testing

Det har vært mulig å få pålitelige resultater fra testing basert på brukerperspektiv i terminalvinduet. Det er gjort undersøkelser av flere andre metoder for softwarebasert testing av programvaren, og mer informasjon om dette står under kapittel 12, "Drøfting".

Under utviklingen av bachelorprosjektet er det utført manuell testing av funksjonaliteten til programvaren for å kontrollere at programvaren responderer korrekt ved forskjellig input fra terminalvinduet. Etter ferdig utvikling av opu-services er det også laget et testscript for å automatisere den manuelle testingen. Når testscriptet blir kjørt sender det kommandoer til hypso-cli, på samme måte som en ville gjort manuelt gjennom terminalen. Testscriptet åpner hypso-cli og opu-services med virtuelle porter og virtuell CAN, og tester kommunikasjonen mellom de to ved å pinge, laste ned en fil, hente statusinformasjon og teste diverse kommandoer over både KISS og CAN. Fordelen med å lage et ferdig testscript er at det blir lettere for andre å forstå hvordan koden i hypso-cli og opu-services skal fungere dersom dette skal jobbes videre med ved et senere tidspunkt. Dette forutsetter at scriptet er godt dokumentert og det er lett å forstå hvilken funksjonalitet som testes og hva ønsket resultat av testingen er.

GitHub Actions er et testverktøy på GitHub som automatiserer forskjellige typer softwarebasert- og hardwaretesting. GitHub Actions bygger koden i et isolert miljø og kjører utvalgte testscripts når en forsøker å merge en pull request (GitHub, 2022). Gruppen har integrert et testscript i GitHub Actions innad i en branch spesiallaget for testing i hypso-SW. Det integrerte testscriptet er det samme scriptet som beskrevet over, og tester derfor kun kommunikasjon gjennom virtuelle grensesnitt. Innholdet i dette testscriptet er vedlagt i vedlegg E. GitHub Actions bygger programvaren og kjører testene på en datamaskin koblet til hardware på SmallSat-laben og gjør det mulig å starte og utføre testene på target hardware fra en ekstern datamaskin. Testscriptet som er integrert i GitHub Actions er relativt forenklet og må utvides til å kjøre mer avanserte og robuste tester av seriell kommunikasjon. Fordi implementeringen av GitHub Actions i SmallSat-prosjek tet er relativt nytt, er det enda ikke ferdig implementert i hovedbranchen til hypso-SW og er i praksis ikke tatt i bruk innad i SW-teamet. Gruppen har valgt å lage en demonstrasjon av hvordan KISS mellom hypso-cli og opu-services kan testes gjennom GitHub Actions. Dette scriptet må sannsynligvis endres på etterhvert som SmallSat utvider bruken av GitHub Actions. Vedlegg C beskriver hvordan en kan kjøre testscripts i GitHub Actions.

8.4.1 Resultater fra softwaretesting

Manuell testing som gruppen utførte tidligere i arbeidet viser at bare det å suksessfult pinge mellom de to ikke nødvendigvis er nok for å bekrefte at filoverføring kan skje uten problemer. Testscriptet gir mer informasjon når det laster ned filer mellom OPU-services og hypso-cli over både CAN og KISS. Testingen bekrefter at det går fint for begge programmene å kommunisere og laste ned filer over KISS og CAN, men det er ikke mulig å få noe informasjon om overføringsraten ved virtuell testing. Bitraten for virtuell CAN er satt til 0. Hastighet for å få svar fra ping er 0-1 ms. Dette urealistisk raskt sammenliknet med fysisk dataoverføring. Testing med ulik baudrate for KISS gir ingen forskjell i målt hastighet. En kan dermed anta at kommunikasjon over virtuelle porter har en egen baudrate som ikke blir påvirket av baudraten som blir satt av hypso-SW. Virtuelle porter bør derfor hovedsakelig brukes for testing og debugging av selve kommunikas-

jonen og programvaren, og er lite egnet for testing og forbedring av overføringsraten.

8.5 Trinn 5: Hardwaretesting

Frem til dette trinnet er all utvikling og testing gjort virtuelt. Det vil si hypso-cli og opu-services blir bygget i hver sin terminal på samme datamaskin og kommunikasjonen testes ved virtuelle serielle porter med socat. Det er nødvendig å teste koden på riktig hardware for å bekrefte at OPUen kan sende og motta data over KISS. Per mai 2022 eksisterer ikke riktig hardware. Det er kun mulig å teste på flatsat. Dette er hardware ekvivalent til det i HYPSO-1, satt opp i laben til SmallSat. Gruppen har testet kommunikasjon over KISS og CAN mellom en datamaskin på laben og flatsat, slik det beskrevet i "Manual for Flatsat and LidSat" (Birkeland, 2021). Hardwaretestingen har som mål å finne ut av to ting: bekreftelse på at kommunikasjon over KISS til og fra hardware OPU fungerer, og informasjon om netto datarate sammenliknet med KISS og CAN.

OPUen på flatsat har kun en CAN-buss og en fysisk UART som brukes til debugging. Det er gjort forsøk å teste KISS på denne UART-porten, men det er problematisk å teste over denne porten fordi informasjon fra operativsystemet på OPUen også skrives til samme port. Det oppstår problemer når OPUen forsøker å sende eller motta meldinger over CSP samtidig som operativsystemet på OPUen skriver til samme port.

Som et alternativ til testing på flatsat, har gruppen prøvd å utføre en type hardwaretest med to USB/UART-adaptere koblet til en laptop. Hensikten her er å vise at overføringen over KISS er stabil, og dette dokumenteres med data som hentes fra hypso-cli etter filoverføringer til og fra opu-services.

8.5.1 Resultater og drøfting fra hardwaretesting

Hardwaretestingen på flatsat blir gjennomført ved at en tekstfil blir lastet opp på OPUen over CAN og lastet ned over KISS. Resultatene fra disse testene er markert med den midterste gråfargen i tabellen. Det er blitt testet å sende og motta data over KISS på debug-porten og det er dermed bekreftet at OPUen kan kommunisere over KISS. UART kommunikasjonen mellom datamaskinen og flatsat over debug-porten har fungert, men til tider har det vært problemer med denne kommunikasjonen. Spesifikt har filopplastning over KISS fra hypso-cli til flatsat ikke fungert. Gruppen kan ikke bekrefte eller avkrefte at det er systeminformasjonen som sendes over debug-porten fra flatsat som er årsaken til problemet med filopplastningen, men dette anses som den mest sannsynlige årsaken.

Gruppen anser de konkrete resultatene over netto datarate i flatsat-testen av KISS som lite pålitelige siden det er vanskelig å vite om debug-informasjonen som sendes over samme port påvirker resultatene, og i så fall i hvor stor grad dette skjer.

De viktigste resultatene i tabellen er "Tapt overføringsrate". Dette blir regnet ut ved:

$$\text{Tapt overføringsrate} = 1 - \frac{\text{Effektiv overføringsrate}}{\text{Baudrate}}$$

Dette beskriver den prosentvise datamengden som ikke er brukbar/ønskelig data. Dette er både overhead bits, men også pauser i mellom hver pakke som sendes. Overhead diskuteres detaljert både i problemstillingskapitlet og i drøftingskapitlet. Disse testresultatene gir noe informasjon om overhead, men de er ikke konkrete tall på overhead i hver protokoll.

De øverste og lyseste resultatene i tabellen er fra testing over virtuelle porter. Ved overføring over virtuelle porter gjennom socat er det ikke mulig å bestemme baudraten, slik som er forklart i kapittel 8.4.1, så disse resultatene kan derfor ikke si noe relevant om overføringshastighet. Testing over virtuelle porter har likevel vist at CSP kan sendes over KISS og at denne kommunikasjonen fungerer som forventet over de virtuelle portene.

Protokoll	Overføringsmedium	Baudrate [Kbps]	Effektiv datarate [Kbps]	Tapt overføringsrate [%]
KISS	Virtuell		520.8042	
CAN	Virtuell		580.2972	
KISS	Flatsat - WS2	500	243.7126	51.25748
KISS	Flatsat - WS2	500	257.7855	48.4429
CAN	Flatsat - WS2	1000	300.2901	69.97099
CAN	Flatsat - WS2	1000	299.9521	70.00479
KISS	USB til UART (laptop)	500	358.5536	28.28928
KISS	USB til UART (laptop)	1000	549.3287	45.06713

Tabell 1: Resultater fra hardware- og software-test

De nederste mørkegrå resultatene i tabellen er gjort med to USB/UART-adaptere på en laptop. Hensikten med denne testen var å forsøke å se mer realistiske tall på overføringsrate over KISS når gruppen har konkludert med at resultatene av testene på flatsat ikke er pålitelige. Her har det også vært mulig å endre baudrate for å sammenlikne hvordan dette påvirker overføringsraten. Gruppen gjennomførte ikke en tilsvarende test med CAN koblet på en laptop grunnet komplisert oppsett av drivere og oppkobling. I tillegg hadde denne typen test med CAN sannsynligvis ikke gitt noe mer informasjon enn testen på flatsat. Dette betyr at det ikke har latt seg gjøre å utføre tester på fysisk hardware med identiske parametre med KISS og CAN. Det er derfor vanskelig å sammenlikne resultatene fra de to grensesnittene. Det mest hensiktsmessige vil være å sammenlikne USB/UART-testen over KISS med flatsat-testen over CAN for å få et grovt bilde av netto overføringsrate. Dette gir ikke et eksakt resultat, men en pekepinn mot at KISS gir en god del mindre overhead enn CAN. Forslag til forbedring av overføringsrate er beskrevet i drøftingskapitlet.

I alle filoverføringstestene i tabellen har filer blitt fullstendig overført uten pakketap. Filene har blitt sjekket etter overføringen og alle filoverføringene har fungert feilfritt. I alle testene har det også vært forsøkt ulike andre kommandoer som beskrevet i "Hardware test guide". Disse kommandoene har både testet ping og pakker som er større og mindre enn maks tillatte pakkestørrelse. I alle testene har alle disse kommandoene fungert som forventet og gitt riktig resultat. Dette betyr at all kommunikasjon i protokoll-lagene over det fysiske laget har fungert feilfritt i alle testene. Det er problemer med det fysiske laget som påvirker testresultatene og det kan strengt tatt ikke utelukkes at dette skyldes noe uforutsett i måten koden kjøres på den fysiske OPUen. Likevel anser gruppen det som absolutt mest sannsynlig at de uønskede testresultatene kommer av debug-UART-porten og ikke faktisk OPUen. Gruppen konkluderer med at kommunikasjonen over KISS på target hardware må bekreftes på HYPSO-2, men at gruppen anser kommunikasjonen over KISS som stabil og fungerende.

8.5.2 Videre hardwaretesting

Ved et senere tidspunkt vil det være nødvendig å teste programvaren på HYPSO-2. Det vil da være hensiktsmessig å lage et automatisk testscript ved fysisk testing slik at testing av seriell kommunikasjon skjer sammen med alle de andre testscriptene som er laget for andre deler av satellitten. Det er lite hensiktsmessig å lage et ferdig testscript nå fordi det er usikkert om HYPSO-2 vil ha nøyaktig samme kommandoer som fungerer på samme måte som HYPSO-1, spesielt kommandoer som har med hardware å gjøre som f.eks. hvilke porter som skal brukes til KISS og CAN. I stedet for å lage et ferdig testscript, har gruppen laget en detaljert oppskrift på en testprosedyre som er vedlagt i vedlegg B. Denne oppskriften beskriver hvordan kommunikasjonen over CAN og KISS fungerer og hva som kreves av både hardware, programvare og kommandoer for å teste seriell kommunikasjon. Oppskriften beskriver detaljert hva hver kommando betyr og utfører.

9 Programmering

Dette kapitlet beskriver i detalj endringene gruppen har gjort i koden til hypso-SW. Kapitlet beskriver hovedtrekkene i funksjonene som blir endret og lagt til slik at fremtidige SmallSat-medlemmer kan forstå funksjonaliteten og hensikten med kodeendtringene. For å få en fullstendig oversikt over alle kodeendringene, må en slå opp i historikken til pull-requestene i hypso-SW på GitHub:

- "#711 Add KISS interface to services".
- "#721 Update libcsp".
- "#695 Cli kiss".

9.1 Hypso-cli

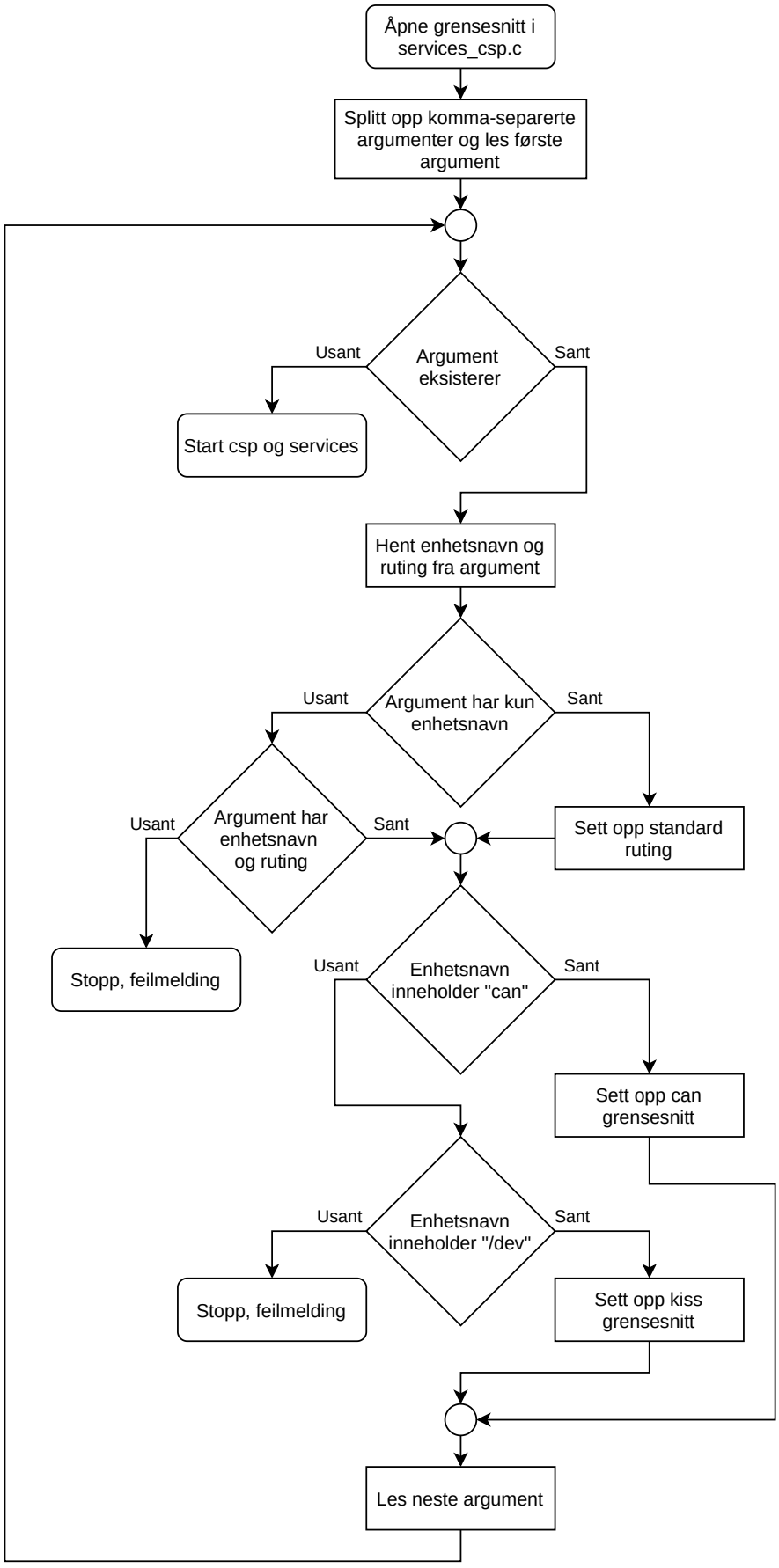
KISS over UART var allerede integrert i hypso-cli, men grensesnittet ble brukt til enkel debugging. Gruppens tillegg for KISS i hypso-cli sørger for at KISS over UART kan brukes på samme måte som CAN.

Måten KISS var implementert tidligere var ved at det ble åpnet et KISS-grensesnitt når kommandoen *csp init usart* ble kalt. Dette åpnet et CSP-grensesnitt med allerede bestemt seriell-port og baudrate. Gruppen vurderte at det var hensiktsmessig å utvide KISS-funksjonaliteten i hypso-cli til å kunne åpne et KISS-grensesnitt når hypso-cli startes fra kommandolinjen med et argument. Her kan brukeren få mulighet til å sette seriell-port og ruting. Dette ble implementert med samme syntaks som CAN-grensesnittet, slik at de kan brukes på samme måte. For å gjøre dette måtte arbeidet deles opp i flere punkter:

- Legge til argument for å initialisere KISS fra kommandolinjen.
- Gjøre KISS init funksjonen tilgjengelig for resten av hypso-cli gjennom header-filen *cli_csp.h*.
- Oppdatere ruting-funksjonen fra den utdaterte *csp_route_set* funksjonen til *csp_rtable_set* slik at vi kunne sette opp rutingen for grensesnittet riktig.
- Endre beskrivelsen av kommandoen og helptexten for å beskrive den nye funksjonaliteten.

9.2 Sdr-services og opu-services

Programvaren til SDR og OPU, sdr-services og opu-services, bruker samme kodefiler for å snakke over CSP, som betyr at ved å implementere KISS i filene *services_csp.c* og *services_csp.h* kan både sdr-services og opu-services kommunisere over KISS. CSP over CAN var allerede implementert i services, men KISS var ikke implementert. Måten det ble valgt å sette opp grensesnitt i services er beskrevet gjennom det forenklede flytskjemaet i figur 6.



Figur 6: Initialisering av grensesnitt i services

En mer detaljert beskrivelse av hvert steg som krevdes for utviklingen følger her:

- **Inkluder nødvendige biblioteker:**

  For å bruke CSP sin KISS-funksjonalitet må KISS-interface og CSP sin USART driver inkluderes.

```
#include "csp/interfaces/csp_if_kiss.h"
#include "csp/drivers/usart.h"
```

- **Del opp initialisering i ulike funksjoner:**

  For enkelhets skyld deles initialiseringen av de ulike grensesnittene opp i ulike funksjoner. CAN initialiseringen flyttes til en egen funksjon ved navn *init_can*. Denne funksjonen holdes så lik som mulig den eksisterende koden for å åpne CAN-grensesnittet, men med oppdateringen av libcsp oppdateres også funksjonen som åpner CAN-grensesnittet. Denne nye metoden er den anbefalte måten å starte CAN-grensesnittet. (kilde?)

```
void init_can(char* can_device, uint8_t address, uint8_t netmask)
{
    int bitrate;

    // Connect to CAN bus
    if (strstr(can_device, "vcan") != NULL)
    {
        // Virtual CAN bus, do not reset interface or set bitrate
        bitrate = 0;
    }
    else
    {
        // Real interface, reset and set bitrate.
        bitrate = HYPSO_CAN_BITRATE;
    }
    // Initialize can interface
    csp_iface_t* can_iface = NULL;
    int ret = csp_can_socketcan_open_and_add_interface(
        can_device, "CAN", bitrate, false, &can_iface);

    // Check if can initialized successfully
    verify_panicf(ret == CSP_ERR_NONE, "Failed to init CAN interface.");

    set_routing_table(can_iface, address, netmask);
}
```

- **Initialiser KISS:**

  KISS initialiseres også i en egen funksjon. Etter oppdateringen av libcsp var det mulig å bruke en oppdatert og mer modulær løsning for å åpne KISS-grensesnitt sammenliknet med den tidligere versjonen av libcsp. Denne ble tatt i bruk og initialisering ble da relativt lik som initialiseringen av CAN. Fordeler med initialiseringen av KISS etter oppdateringen av libcsp er muligheten til å åpne så mange KISS-grensesnitt en ønsker når programmet startes, og muligheten til å sjekke at KISS-grensesnittet blir åpnet på riktig måte. Ved den tidligere libcsp-versjonen måtte en definere hvor mange KISS-grensesnitt en ønsket å støtte globalt i koden, og der var ingen måte å håndtere feil under åpningen av et KISS-grensesnitt.

```
void init_usart(char* usart_device, uint8_t address, uint8_t netmask)
{
```

```
    // Configure usart
    csp_usart_conf_t conf;
    conf.device = usart_device;
    conf.baudrate = HYPSO_USART_BAUD;

    // Initialize KISS interface
    csp_iface_t* usart_iface = NULL;
    int ret =
        csp_usart_open_and_add_kiss_interface(&conf, "KISS", &usart_iface);

    // Check if usart initialized successfully
    verify_panicf(ret == CSP_ERR_NONE, "Failed to init USART interface.");

    set_routing_table(usart_iface, address, netmask);
}
```

- **Ruting:**

  Ruting betyr at CSP setter opp en tabell for hvilke grensesnitt som skal brukes for å sende meldinger til spesifikke adresser. Det lages en egen funksjon for oppsett rutingen av grensesnittene kalt *set_routing_table*. Tidligere ble den utgåtte funksjonen *csp_set_route* brukt for å rute grensesnitt, men det blir valgt å bruke den nye og anbefalte funksjonen *csp_rtable_set* istedet.

```
void set_routing_table(csp_iface_t* csp_iface, uint8_t address, uint8_t netmask)
{
    int ret = csp_rtable_set(address, netmask, csp_iface, CSP_NODE_MAC);
    verify_panicf(ret == CSP_ERR_NONE, "csp_route_set failed with ret: %d.",
                  ret);
}
```

- **Les flere grensesnitt fra argumenter:**

  For å kunne sette opp flere grensesnitt i programmet, må disse spesifiseres via kommandoen som starter programmet. Det er nødvendig for bakoverkompatibilitet med HYPSO-1 at syntaks for å starte opu-services ikke endres.

  Syntaksen for å starte opu-services spesifiserer ett argument for csp-grensesnitt. En måte å gi flere verdier til et argument er gjennom en kommaseparert liste. Dette ble vurdert som den enkleste måten å legge til muligheten for å støtte flere CSP-grensesnitt. Ulempen med dette er at det kun må brukes komma i argumentene, ikke mellomrom, og dette kan virke lite intuitivt for menneskelige operatører. Derfor har gruppen dokumentert og forklart dette nøye i intruksjonsteksten til programmet.

```
// csp_device: String containing each comma separated argument
// (ex: can0=0/0)
char* csp_device = strtok(csp_devices, ",");
// make sure any argument exists
verify_panicf(csp_device, "No valid argument: %s", csp_devices);

// device: String containing the device (ex: vcan0 or /dev/ttyUSB0)
char* device = strdup(csp_device);

// Parse comma-separated csp devices
while (csp_device)
{
    /*
    Initialization of interfaces happens here
```

```
    */
    csp_device = strtok(NULL, ","); // reset to parse next device
}
```

- **Åpne grensesnitt basert på argumenter:**

  For å åpne grensesnittene ble det skrevet kode som leser et argument for et grensesnitt og henter ut den nødvendige informasjonen (addresse, netmaske og enhetsnavn). Argumentene i services følger samme syntaks som argumentene i hypso-cli, så de leses på samme måte.

  Det legges til kode for å kunne opprette grensesnitt og ruting basert på syntaks og argumenter likt som i hypso-cli. I tillegg krever bakoverkompatibiliteten at det må være mulig å opprette grensesnitt og ruting basert på den enklere syntaksen brukt i HYPSO-1. Dette bruker et standard oppsett av adresser når det brukes ufullstendige argumenter. Opu-services tillater nå både syntaksen *opu-services 12 can0=0/1,/dev/ttyUSB0=16/1* og *opu-services 12 can0*. Når enhetsnavnet leses opprettes grensesnitt basert på dette. For at et CAN-grensesnitt skal initialiseres må enhetsnavnet inneholde "can", og for at KISS-grensesnitt skal initialiseres må enhetsnavnet inneholde "/dev" (seriell-porter ligger under /dev i Linux kilde?).

```
// Set address and netmask to default 0
uint8_t address = 0;
uint8_t netmask = 0;

// Extract device and routing (address/netmask) from
// comma-separated-argument
int n_parsed_args =
    sscanf(csp_device, "%[^=]=%hhu/%hhu", device, &address, &netmask);
if (n_parsed_args == 1)
{
    // When only device is given, only 1 value is read from csp-device
    debug_printf("Could not parse route from %s, default to "
                 "%s=%hhu/%hhu.",
                 csp_device, device, address, netmask);
}
else if (n_parsed_args != 3)
{
    // Error if values read are not 1 or 3
    // (only device or device and routing)
    verify_panicf(NULL, "Invalid initializer: %s", csp_device);
}
// Initialize CSP interface
if (strstr(device, "can") != NULL)
{
    init_can(device, address, netmask);
}
else if (strstr(device, "/dev") != NULL)
{
    init_usart(device, address, netmask);
}
else
{
    verify_panicf(NULL, "Device not recognized: %s", device);
}
```

Endringene gjort i opu-services og sdr-services kan oppsummeres med at det er blitt mulig å spesifisere flere CSP-grensesnitt med ruting. Initialiseringen av grensesnitt er delt mer opp sammenlinket med hvordan CAN tidligere ble initialisert. Funksjonen som initialiserer KISS er satt

opp slik at den blir behandlet på samme måte som CAN. Alle endringer som er gjort er bakoverkompatible med HYPSO-1.

Merk: i store deler av koden i hypso-SW er det skrevet USART i stedet for UART. USART og UART er to forskjellige typer hardware og forskjellen ligger i at USART kan sende både synkron og asynkron seriell kommunikasjon, mens UART bare kan sende asynkront. All kommunikasjon over KISS er asynkron kommunikasjon og det gir derfor mening å snakke om UART, men denne kommunikasjonen kan også sendes over en USART. I bachelorprosjektet brukes dermed disse to som synonymer, selv om de ikke nødvendigvis kan brukes om hverandre i andre settinger.

10 Oppdatering av Libcsp

10.1 Bakgrunn og motivasjon

SmallSat bruker CSP til kommunikasjon mellom enheter innad i stelitten. For å sende og motta CSP-kommunikasjon brukes biblioteket libcsp. SmallSat har brukt en egen fork av dette biblioteket siden 2019. I SmallSat sin fork har det vært gjort endringer og tillegg som er nødvendig for hypso-SW, men forken har ikke blitt vedlikeholdt og ingen endringer hadde blitt gjort siden 21 Mars 2020. Det offisielle libcsp biblioteket er blitt vedlikeholdt, og det er kommet store tillegg siden 2019 som ikke har blitt lagt inn i SmallSat sin fork, og dermed ikke en del av hypso-SW. Å ha en egen fork av libcsp er en fordel fordi Smallsat har kontroll over all oppdatering av kode. Ulempen er at det blir færre tilgjengelige mennesker til å legge merke til og fikse bugs og mangler i koden. Etter hvert blir SmallSat sitt libcsp-biblitek og det offisielle libcsp-bilioteket mer og mer forskjellige, og det blir mer utfordrende å kunne legge inn oppdateringer fra det offisielle biblioteket inn i SmallSat sin fork.

Fordi SmallSat sitt libcsp-bibliotek ikke har fått offisielle oppdateringer siden 2019, har det bygd seg opp forskjeller mellom de to versjonene. Blant disse forskjellene er noen tillegg i det offisielle libcsp som endrer hvordan grensesnitt åpnes. Gruppen ønsket å ta i bruk disse tilleggene i sin implementasjon av KISS i opu-services, og dette var gruppens motivasjon for å oppdatere SmallSat sin fork av libcsp-biblioteket (se vedlegg M, N og R).

10.2 Innhold i oppdateringen

Gruppen valgte å oppdatere SmallSat sin fork til den siste utgivelsen av den offisielle libcsp, nemlig versjon 1.6 (libcsp, 2020). Å oppdatere til denne versjonen har blant annet gitt følgende fordeler:

- En ny funksjon som kan brukes for å åpne KISS-grensesnitt. Denne funksjonen vil returnere en relevant feilkode om det oppstår problemer med initialiseringen av KISS-grensesnittet, og dette gir mulighet til å behandle feil i initialisering av grensesnitt på et senere tidspunkt i koden. Denne funksjonen gjør det også enklere å åpne flere KISS-grensesnitt, noe som kan være interessant for senere utvikling.
- En mer modulær og grundig måte å åpne flere CAN-grensesnitt, noe som utfaser en egen endring som tidligere har vært gjort i SmallSat sin libcsp-fork.
- Fiksing av bugs. For eksempel en bug relatert til KISS-kommunikasjon ble fikset. Denne bugen hadde gruppen tidligere fikset i SmallSat sin egen libcsp-fork, men det offisielle libcsp-biblioteket implementerte en mer robust fiks av denne bugen. (se vedlegg O)
- Oppdatert byggesystem og oppdaterte funksjoner som er vedlikeholdt av et utviklerteam. Se underkaptittel 10.3.
- Et enklere grunnlag for å oppdatere til en eventuell senere versjon av libcsp om det skulle være ønskelig.

- Det er en fordel å ha en fork av libcsp som er lik som den offisielle versjonen fordi dette gjør det lettere å samarbeide og få hjelp av andre utviklere som også arbeider med det offisielle libcsp-biblioteket. Da er det også mulig å merge oppdateringer og forbedringer direkte til den offisielle versjonen slik at andre utenfor SmallSat også kan benytte seg av oppdateringene.

10.3 Bygging og sikkerhet

Libcsp bruker Waf som byggesystem, og dette byggesystemet utvikles og kjøres med Python. SmallSat sitt gamle libcsp-bibliotek brukte Waf med Python 2, og dette er et problem fordi Python 2 er avviklet og ikke blir oppdatert etter januar 2020 (Python Software Foundation, 2019). Det betyr at Python 2 ikke oppdateres i det hele tatt og sikkerhetsproblemer og bugs ikke lengre blir fikset. Det er et stort sikkerhetshull i hypso-SW dersom et av de største bibliotekene har et byggesystem som kan falle fra hverandre når som helst på grunn av utdatert programvare. Det oppdaterte libcsp-biblioteket støtter Waf med Python 3, som fortsatt blir støttet og aktivt vedlikeholdt.

10.4 Testing av biblioteker

Oppdateringen av libcsp kommer også med ulemper fordi det krevde større endringer både i byggeprosessen til hypso-SW og i koden. Dette betyr at en oppdatering kan introdusere feil som er vanskelige å oppdage. I tillegg bruker Nano Avionics sin egen versjon av libcsp i sine komponenter på satellitten. Derfor må den oppdaterte versjonen også være kompatibel med Nano Avionics sin versjon. Gruppen vurderte derfor at oppdateringen av libcsp måtte testes grundig og det måtte være sikkert at all kommunikasjon var stabil i alle applikasjoner og mellom gamle og nye versjoner av libcsp.

Testingen har gått ut på å integrere den nye versjonen av libcsp i programvaren og kjøre programvaren på lidsat og en datamaskin koblet til lidsat. Deretter testes alle relevante kommandoer som involverer CSP for å sjekke at programvaren fungerer på target hardware. Deretter bygges hypso-cli med ny libcsp og opu-services med den gamle versjonen av libcsp. De samme kommandoene for CSP testes for å kontrollere at kommunikasjonen fungerer mellom programvare med forskjellige versjoner av biblioteket. Filopplasting og filnedlasting testes også. I tillegg testes kommunikasjonen med satellittens strømforskyning som opererer med Nano Avionics sitt libcsp bibliotek. Den oppdaterte libcsp-versjonen består alle testene. Dette bekrefter at den nye versjonen av libcsp både fungerer på target hardware, er kompatibel med Nano Avionics sine komponenter og er bakoverkompatibel med eldre versjoner av biblioteket.

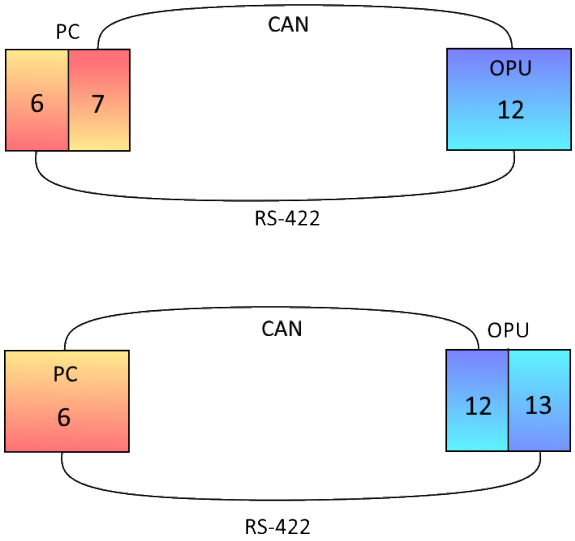
Testingen blir utført grundig og i samarbeid med andre i SmallSat teamet og opppdateringen av libcsp blir vurdert som stabil og pålitelig.

11 Integrasjon

Dette kapitlet forklarer hva andre som fortsetter arbeidet med seriell kommunikasjon innad i SmallSat må vite for å kunne integrere arbeidet fra bachelorprosjektet inn i HYPSO-2. Integrasjon er utfordrende i et prosjekt som SmallSat. Det er fort gjort å få folk til å sitte med utvikling av hvert sitt softwareprosjekt, men når prosjektene er ferdige, er det ikke like lett å "klikke" hver modul sammen. Det er noen utfordringer knyttet til dette bachelorprosjektet fordi det ikke er sikkert at de planlagte hardwareløsningene faktisk blir gjennomført. Det er mulighet for at det blir tatt beslutninger noen måneder frem i tid som gjør at tillegget i programvaren som er laget ikke lengre passer like godt inn med resten av programvaren eller hardwaren. Derfor er det viktig å detaljert forklare hvilke antakelser og forutsetninger gruppen har lagt til grunn i utviklingen av bachelorprosjektet, og hvordan gruppen forventer at bidraget fra bachelorprosjektet skal passe inn i det helhetlige systemet i HYPSO-2.

11.1 Adresser og applikasjoner

Dette underkapitlet beskriver konfigurasjonen av applikasjoner innad i satellitten med de endringene gjort i løpet av bachelorprosjektet. En applikasjon er i dette tilfellet programvare i en del av satellitten, for eksempel OPU-services og SDR-services. En applikasjon kan være sender og mottaker av data og hver applikasjon kan kun ha en CSP-adresse. Det settes opp ruting i det en applikasjon starter, og her bindes CSP-addresser til et eller flere ulike grensenitt. Hver CSP-adresse i systemet kan kun være bundet til et grensesnitt av gangen, om dette skal endres må programmet restartes. Når en starter OPU-services kan en velge mellom CAN eller KISS som grensesnitt for å sende data. Hvis en velger KISS, vil det dermed kun være mulig å sende data til PC over RS-422 mens applikasjonen kjører. Dersom det skal sendes data over RS-422 og CAN samtidig mellom OPU og PC, må det startes en ny applikasjon i enten OPU eller PC med en ny CSP-adresse. Det må enten være to OPU-services som kjører på OPU med hver sin adresse som er rutet til hvert sitt grensesnitt, eller to applikasjoner i PC med hver sin adresse rutet til hvert sitt interface. CSP kan ikke sende over to forskjellige grensesnitt som er bundet til samme adresse. Dersom man binder kommunikasjon mellom OPU og PC til KISS, vil det fortsatt være mulig for både OPU og PC å bruke CAN, men da kun for å kommunisere med andre applikasjoner, som sdr-services, ikke hverandre.



Figur 7: Oversikt over konfigurasjon av CSP-id mellom OPU og PC

Gruppen har forstått det slik at dette er et ønskelig scenario fordi alt OPUen foretar seg skal være basert på request/response. Det vil si at det ikke er ønskelig at OPU ruter kommunikasjonen selv, men kun gjør det den eksplisitt blir fortalt gjennom rutingen til grensesnittene. Dersom en ønsker å bruke et oppsett med både KISS og CAN som vist i figur 7, har det ikke noe å si for kommunikasjonsdelen om man bruker to opu-services eller to applikasjoner i PC. Dataen vil bli sendt over KISS og CAN like raskt og på samme måte. Det er lettere å starte to opu-services enn å stoppe og starte applikasjonen med en ny CSP-adresse. Da kan det også være mulig å implementere en løsning der de to OPU-programmene snakker med hverandre innad i OPUen.

I tidligere kommunikasjon med SmallSat (se vedlegg H) ble det tatt opp at det på et tidspunkt kunne vært ønskelig med en løsning der det blir satt opp to CSP-adresser innad i samme applikasjon. Med biblioteket libcsp er det per i dag ikke mulighet for å sette opp flere adresser i samme applikasjon. I dokumentasjon på GitHub til det offisielle libcsp-biblioteket er dette noe som er blitt nevnt som en ønsket utvidelse på et senere tidspunkt, men er ikke gjennomførbart slik libcsp er i dag (J. Christiansen, 2014, J. Christiansen, 2021). Derfor er CSP-adressene globale i koden og det kan bare settes en adresse per applikasjon.

11.2 Payload controller og operations

Det er ikke mulig for payload controlleren å laste opp og laste ned data samtidig i samme applikasjon. Gruppen har ikke gjort noe forsøk på å endre på dette. Det er fortsatt nødvendig å sette av tiden det tar å bufre filer mellom OPU og PC slik at det ikke gjøres andre oppgaver samtidig som dette skjer. Hele bildefilen må fortsatt bufres til PC før den kan lastes ned via S-band.

11.3 Baudrate

Standard baudrate for KISS er satt til 500 Kb/s. Dette må endres på når RS-422 blir implementert for å tillate kommunikasjonen å bli overført med en hastighet på 2-3 Mb/s. Endring av baudrate må endres inne i hypso-SW. Baudraten kan endres i filen *HYPSO.h* under navnet *HYPSO_USART_BAUD*. Programmet må bygges på nytt og lastes opp på target hardware på nytt hver gang baudrate endres. Det er viktig at denne baudraten blir satt basert på resultater fra testing på target hardware.

11.4 Integrasjonstesting

Det vil være nødvendig å teste kommunikasjon over KISS og CAN i parallell med hver sin adresse på target hardware. Det vil at konfigurasjonen vist i figur 7 må testes på target harware med alle kommandoer, samt filopplasting og filnedlasting.

12 Drøfting

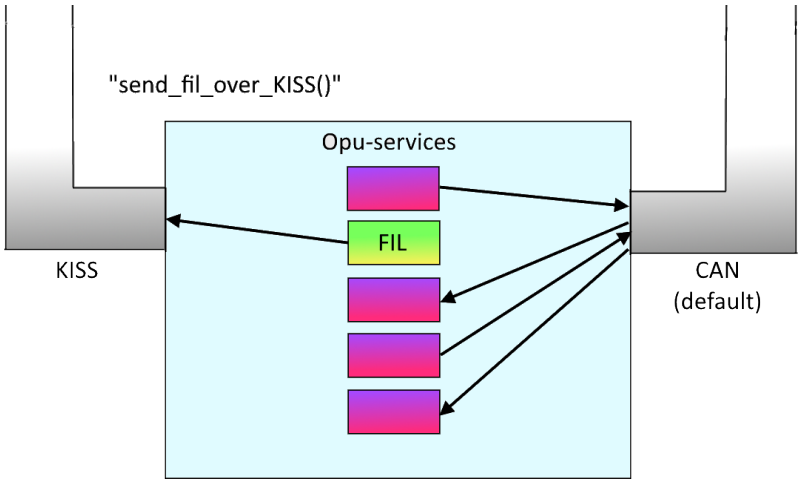
12.1 Alternative løsninger på oppgaven

Et alternativt oppsett for å øke overføringsraten med et KISS grensesnitt mellom OPU og PC er å bruke RS-422 og CAN parallellt mellom samme addresser i samme applikasjon. Dette er blant annet nevnt i GitHub issuet presentert i vedlegg G.

Den store **fordelen** med å implementere RS-422 og CAN parallellt ville vært å kunne kjøre på samme oppsettet som allerede eksisterer med de samme CSP-adressene. Det er mulig at det å sende informasjon over to grensesnitt parallelt kunne gjøres på en måte som går fortere enn å rute kommunikasjonen over grensesnittene enkeltvis.

Den største **ulempen** med å implementere RS-422 og CAN parallellt med de samme CSP-adressene er at implementeringen potensielt kan bli veldig komplisert, og videre skape større kompleksitet i det overordnede systemet på satellitten. Gruppen vurderte likevel to ulike mulige implementeringer:

1. En mulighet kan være å manuelt sende CSP pakker over RS-422 gjennom funksjonen *csp_send_direct*. Dette betyr at CSP vil sende alt av vanlig data gjennom de standard grensesnittene (som i praksis kun vil være CAN), men ved spesielle tilfeller kan en applikasjon rute spesifikke pakker over KISS-grensesnittet. Dette kan da skje i en egen funksjon som kalles spesifikt for å f.eks. sende filer over KISS. Denne funksjonen er nødt til å sette opp pakker manuelt og sørge for at disse blir sendt over riktig grensesnitt med funksjonen *csp_send_direct*. Dette gjør at det ikke trengs å settes opp ruting av CSP-adresser på KISS grensesnittet, siden all vanlig CSP data bør gå over CAN. En må forstatt åpne KISS-grensenittet i programvaren, men det vil ikke være nødvendig med spesifikk ruting til grensesnittet.



Figur 8: Forslag til alternativ funksjonalitet i opu-services. Se sammenlikning med figur 7.

Ulempen med denne løsningen er at det må spesifiseres hver gang en pakke skal sendes over KISS, og funksjoner som ønsker å sende meldinger over KISS må designes spesifikt for dette. Det kan være komplisert å implementere denne løsningen, fordi den krever at en setter opp pakker spesifikt for hvor de skal og sender de manuelt over riktig grensesnitt. Løsningen introduserer et nytt lag med kompleksitet i kommunikasjonen, og sender CSP kommunikasjon utenom den standard måten.

Gruppen vurderte den medfølgende kompleksiteten til denne løsningen stor, og valgte derfor å bruke løsningen med flere separate applikasjoner for å kunne kommunisere over CAN og KISS innad i satelitten (som beskrevet i kapittel 11.1). Gruppen vurderer likevel dette som en mulig fremtidig utvidelse, om det blir bestemt at separate CSP-IDer ikke er ønskelig og det er behov for et alternativt oppsett.

Om det er ønskelig å implementere denne løsningen kan programmet *packet_dropper* i HYPSO-SW være til god hjelp. *packet_dropper* sender pakker mellom to CAN-grensesnitt, på samme måte som vi foreslår i denne løsningen.

2. Den andre mulige implementasjonen er å sende meldinger over RS-422 uavhengig av CSP, og dermed aldri implementere KISS i hypso-SW i utgangspunktet. Potensielt kan pakker bli sendt spesifikt over RS-422 og mottatt gjennom RS-422 uavhengig av CSP, men denne løsningen er avhengig av å bruke et alternativt rammeverk og alternative protokoller for å sende og motta over RS-422. Denne løsningen ville gi stor valgfrihet og mye muligheter for optimalisering av kommunikasjonen mellom OPU og PC. Fordelen med en slik løsning er at overføringen kan optimaliseres for det systemet det utvikles for, og potensielt spare unødvendige begrensinger som for eksempel de beskrevet i kapittel 12.2.

   En ulempe med en slik løsning er at kommunikasjon over RS-422 i dette tilfellet ikke vil foregå på samme måte som kommunikasjonen over CAN i resten av sattelitten. Om løsningen optimaliseres for overføringen mellom OPU og PC vil den ikke lenger være standardisert på samme måte som CSP. Det kan tenkes at seriellkommunikasjonen mellom enheter innad i satelitten vil endres eller utvides, og det vil være mye vanskeligere om ikke all kommunikasjon følger samme standard. En annen ulempe med denne løsningen er at det rammeverket som ville bli valgt for å kommunisere over RS-422 kunne introdusert et unødvendig tillegg i kodebasen, både i form av plass og i form av kompleksitet. Utviklere av satelitten er avhengig av å være kjent med hvordan kommunikasjonen over alle grensesnittene fungerer for å holde dem vedlike, så om det introduseres et nytt rammeverk for RS-422 i tillegg til CSP som allerede er i bruk, vil dette gjøre det mer krevende å arbeide med programvaren.

   Løsningen med å lage et egen rammeverk for filoverføring over RS-422 kunne blitt utarbeidet både på en modulær og bakoverkompatibel måte, men dette er ikke nødvendigvis nok for å konkludere med at løsningen er praktisk eller fornuftig. Å bruke flere ulike rammeverk for kommunikasjon vil skape unødvendig kompleksitet og uoversiktlighet. Det er derfor mest hensiktsmessig å holde seg til CSP når det gjelder å sende og motta pakker mellom enheter

i sattelitten, selv om hastigheten på filoverføringen mellom enhetene ville kunne blitt mer effektiv.

12.2 Veien videre: Forbedring av overføringsrate fra et softwareperspektiv

En datapakke vil bestå av både overhead og selve dataen som blir sendt. CSP har 4 bytes med overhead (J. D. C. Christiansen, 2021), CAN har 6 bytes overhead (Corrigan, 2016) og KISS har en nibble (4 bits) overhead (Chepponis and Karn, 1987). CAN har bare plass til 8 bytes med data i en datapakke (Corrigan, 2016). Dette betyr at en CSP-datapakke som blir sendt over CAN blir først delt opp i 8 bytes store pakker som igjen blir "pakket inn" i en 6 bytes stor header. Dette utgjør dataoverføring med veldig mye overhead. Det ser ut som det er lite konkret en kan gjøre for å øke overføringsraten dersom man bruker CAN. Eventuell optimalisering av dataoverføring må i så fall bli gjort i de øvre protokoll-lagene og i filbehandlingen. KISS sin størrelse på datapakkene varierer avhengig av hardware og programvare. KISS kan sende pakker på opp til 1024 bytes, men størrelsen på datapakkene kan bestemmes av bufferminne tilgjengelig og eventuell programvare (Chepponis and Karn, 1987). Selv med datapakker på 8 bytes, samme størrelse som CAN, vil KISS ha høyere overføringsrate på grunn av overhead på bare 4-8 bits. Jo større datapakker på CSP og KISS, jo høyere vil netto overføringsrate bli fordi prosentvis overhead per datapakke blir mindre.

Maximum transmission unit (MTU) beskriver hvor mye data som maksimalt kan overføres i en datapakke i en nettverksprotokoll. Dette betyr at MTU for nettverksprotokollen brukes som referanse for de lavere protokollene (Cloudflare, 2022). I praksis betyr dette at MTU = data + overhead når det er snakk om nettverksprotokoll og MTU = data - overhead når det er snakk om datalinkprotokoller og fysiske protokoller. I hypso-SW er MTU for CSP hardkodet til å være 256 bytes. Dette er data inkludert overhead. KISS er begrenset til MTU for CSP, og dette betyr at KISS-pakken uten header er 256 bytes.

Det at CSP-pakkene kun kan være 256 bytes er bestemt av Nano Avionics som leverer og monterer store deler av hardwaren til HYPSO-1 og HYPSO-2 (NanoAvionics, 2021). MTU i hypso-SW er dermed en begrensning satt av en ekstern aktør og er utfordrende å gjøre noe med. Det kan være en fordel å ha standardiserte pakkestørrelser i hele satellitten for å gjøre integrasjon av både hardware og programvare enklere. Likevel ville det å øke MTU bidra til å gjøre overhead mindre og forbedre overføringsraten og effektiviteten til satellitten. Dersom det hadde gått an å finne en løsning der MTU kunne settes manuelt for komponentene som har med nyttelasten å gjøre, hadde dette kunne bidratt til å gjøre filoverføringen mer effektiv. Større pakkestørrelser er mer sårbare for feil i overføringen (Murray, 2012) og det hadde sannsynligvis måtte brukes en del tid på å finne pakkestørrelsen med best mulig resultat for både pakkeintegritet og mengden overhead.

Måten pakkestørrelsene er satt på er hardkodet mange steder i programvaren til hypso-SW. Det å gjøre pakkestørrelsen til en variabel som kan bestemmes av bruker hadde ikke hatt noe å si for bakoverkompatibilitet med HYPSO-1 dersom den gamle verdien fortsatt kan brukes. Det er også lite sannsynlig at økt MTU i CSP hadde påvirket overføringsraten over CAN. CAN vil uansett alltid dele CSP-pakken opp i 8 bytes. For CAN sin del vil en større CSP-pakke muligens påvirke buffer-kapasiteten. Dette betyr at CAN lagrer de ferdig oppdelte og innpakkede CSP-pakkene i en buffer og sender alle disse små pakkene etter hverandre (Glabbeek, 2017). Ingen andre typer pakker blir sendt over CAN før hele CSP-pakken er blitt sendt. En større CSP-pakke vil føre til lengre buffertid og kreve større bufferkapasitet. Hovedproblemet med å åpne for å endre MTU er at det er uoversiktlig i hypso-SW hva slags definert pakkestørrelse som gjelder for nyttelasten, hva som gjelder for Nano Avionics sine komponenter og hva som gjelder på begge. Det å åpne for å endre på MTU hadde trolig krevd uforholdsmessig mye arbeid for å forsikre om at forskjellig MTU i forskjellige deler av satellitten ikke ødelegger funksjonaliteten.

12.3 Veien videre: Softwarebasert testing

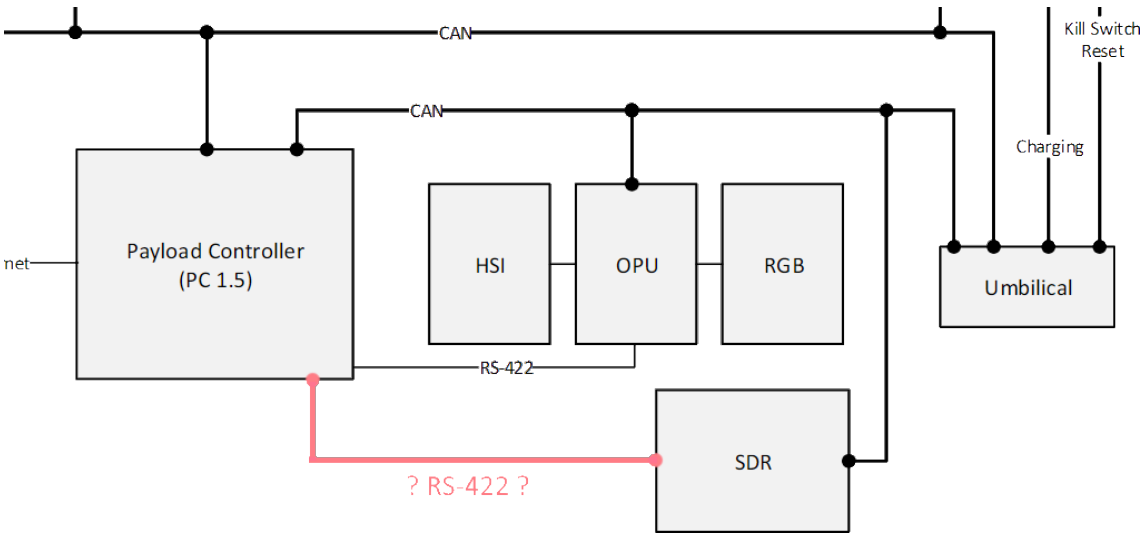
I tillegg til testscript som automatiserer de manuelle kommandoene som skrives inn i terminalvinduet, er det en annen metode for softwarebasert testing som er relevant for bachelorprosjektet. Unit-testing er et generelt begrep som ofte handler om å teste at funksjoner i en kodefil oppfører seg slik de skal. Unit-tester skrives oftest på samme språk som koden som skal testes. Unit-testing i hypso-SW skrives med Check, et rammeverk for unit testing i C. Dette rammeverket gjør det mulig å teste bugs og kodefeil som vanligvis er vanskelige å fikse. Et veldig enkelt eksempel på unit-testing kan være test av en funksjon som ganger to int-inputs med hverandre og returnerer resultatet. Da lages det et script som tester gangefunksjonen med å sette inn forskjellige inputs og se om funksjonen returnerer en forventet verdi. Check vil gjøre dette enda mer effektivt ved å kunne fange usynlige feil i kompilering. Ofte ønsker man å se på hva som skjer hvis man gir funksjonen dårlig input som f.eks. sette en float eller streng som input og se hvordan funksjonen og programmet responderer.

Det er i teorien lett å lage unit tester for funksjoner som har klart definert input og output. Problemet med koden i bachelorprosjektet er at funksjonene som har med seriell kommunikasjon å gjøre ikke har like tydelig definerte input og output fordi mye av denne koden er avhengig av spesifikk hardware. Dette betyr at uansett hvordan man skriver unit-tests, risikerer man at testen feiler fordi testen ikke er kompatibel med hardware i datamaskinen den kjøres på. Det er derfor lettere å skrive gode unit-tests for kode som bare er softwareavhengig fordi en kan sørge for at programvaren og test-scriptet bygges på eksakt lik måte hver gang testen kjøres. Gruppen har vurdert at det per i dag ikke er hensiktsmessig å designe unit-tester for koden som er laget i bacheloroppgaven ettersom dette hadde krevd uforholdsmessig mye ressurser og testene i beste fall bare kunne fungert på enkelte datamaskiner.

Det burde på et tidspunkt lages unit testing for funksjoner som dreier seg om KISS og UART og inkludere dette i GitHub Actions. Det er mest hensiktsmessig å automatisere unit-testene i GitHub Actions slik at testene kjøres hver gang en pull-request skal godkjennes i stedet for å manuelt måtte kjøre de unit-testene en regner med at er relevante for spesifikke kodeendringer. Formålet med unit-testene burde være en type testing som kan sjekke om ny eller endret kode som lastes opp til HYPSO-SW påvirker eller ødelegger funksjonaliteten til den serielle kommunikasjonen over KISS, enten dette er med vilje eller ikke. Dette må være kode som kommer med feilmeldinger, men ikke nødvendigvis forbyr endringer i funksjonaliteten til KISS. Dette kunne vært hensiktsmessig dersom en skulle lagt til endringer i KISS slik som er beskrevet i kapittel 12.2.

12.4 Veien videre: Hardware

Fordi SDR også er koblet til CAN-bussen, blir dataoverføring herfra også negativt påvirket av den trege overføringsraten. Dette utgjør likevel ikke et ekstremt stort problem fordi det er mye mindre data som overføres mellom SDR og PC sammenliknet med de hyperspektrale filene mellom OPU og PC. I tillegg er det lite praktisk og lite sannsnylig at SDR og OPU kommer til å være skrudd på og aktivt sende data samtidig, så det vil ikke oppstå en propp over CAN-bussen som resultat av dette. Likevel bruker SDR også CSP over CAN, som sannsynligvis fortsatt ligger på ca 60-70% tap i datarate, og dette burde ideelt sett gjøres noe med. Fordi det er vanskelig å gjøre noe direkte med CAN for å forbedre mengden overhead, kunne det vært mer hensiktsmessig å se på alternativer til CAN-buss i det fysiske laget.



Figur 9: Forslag til bedre kommunikasjon mellom SDR og PC. Bildet er redigert fra HYPSO-2-dokumentasjon internt i SmallSat

En kunne sett på muligheten for å legge til en annen seriell databuss i stedet for, eller i tillegg til CAN-bussen mellom SDR og PC. Dette hadde sannsynligvis resultert i relativt lite utbytte sammenliknet med arbeidet og ressursene det hadde krevd. Likevel har tillegget i koden i hypso-SW fra dette bachelorprosjektet åpnet for kommunikasjon over KISS fra SDR-services i tillegg til opu-services. Dette betyr at det bare mangler fysisk RS-422-grensesnitt på SDR og PC for å kunne la SDR kommunisere over RS-422 i stedet for CAN. Dette hadde resultert i raskere dataoverføring, men hadde risikert å gjøre hele CAN-bussen unødvendig. Hovedfordelen med å bruke CAN-buss i utgangspunktet er at det er en databuss der flere enheter kan motta og sende data mellom hverandre. Dette gjør at oppkoblingen mellom alle enhetene blir mye enklere og mindre kompleks enn hvis man skulle hatt et nettverk av toveis kommunikasjonsledninger.

Fordelen med å implementere RS-422 i SDR ville vært å hindre effekttapet som kommer av den store mengden overhead på CAN. Ulempen er at dette krever et mer komplisert oppsett og oppkobling av komponenter. Alvorligheten av effekttapet avhenger av mengden data som sendes over CAN. Gruppen foreslår som løsning at kommunikasjonen til og fra SDR over CAN testes på flatsat på laben på samme måte som er beskrevet mellom OPU og PC i 8.5, og som er beskrevet i test-oppskriften vedlagt i Hardware test guide. Herfra bør det være mulig å finne tall på netto datarate og hvor mye effekttap som skyldes overhead på CAN. Basert på dette kan det tas en vurdering på om det er verdt å gjøre noe med dataoverføringen til og fra SDR ved å implementere RS-422.

12.5 Veien videre: Prosjektering og programvareutvikling

Dette underkaptilet er ment som velmente og konstruktive tilbakemeldinger om prosjekteringen innad i SmallSat-teamet. I dag har HYPSO-1 og HYPSO-2 et felles repository på GitHub for programvare. Dette er en beslutning basert på kapasiteten til SmallSat-teamet og som trolig ikke vil bli endret på mens HYPSO-2 er under utvikling. På et eller annet tidspunkt vil man sannsynligvis måtte bryte bakoverkompatibiliteten og forlate HYPSO-1 i et eget repository. Frem til da vil det være spesifikke utfordringer knyttet til prosjekteringen i SW-teamet. Det er mye skriftlig dokumentasjon innad i SmallSat i forskjellige arkiver og plattformer. Utfordringen er at det er veldig vanskelig å finne fram i, og store deler av informasjonen bachelorgruppen har arbeidet med har blitt gitt muntlig fordi dette har vært den letteste måten å dele informasjon. I tillegg er det relativt mange beslutninger i arbeidet med HYPSO-2 som kun baserer seg på bakoverkompatibiliteten med HYPSO-1 og som ikke er spesielt intuituve å forstå som en utenforstående. Dersom dokumentasjonen for disse designvalgene er vanskelig å finne eller dårlig dokumentert, vil det gjøre det vanskelig for nye team-medlemmer som skal gjøre seg kjent med

HYPSO-prosjektet. Til nå har det vært forsøkt å få alle prosjektarbeiderne i SmallSat til å gjøre det de kan for å dokumentere og lage ovesiktlige arkiver med dokumentasjonen. Det har også vært enkelte team-medlemmer som har påtatt seg ekstra ansvar for å rydde i filsystemene og dokumentasjonen, men på et eller annet tidspunkt blir prosjektet så stort og komplisert at denne typen dugnadsarbeid ikke er tilstrekkelig for å holde arbeidet oversiktlig.

Gruppen foreslår dermed at SmallSat lyser ut enten en bacheloroppgave i studiet "Bachelor i arkiv- og samlingsforvaltning" (NTNU, 2022a) eller en masteroppgave i studiet "Master i arkiv og dokumentasjonsforvaltning" (NTNU, 2022b) som går ut på å lage et proffesjonelt og oversiktlig arkiv- og dokumenteringssystem i HYPSO-prosjektet. Denne typen dokumenteringssystem må bygges fra bunnen av med den hensikt å være lett å vedlikeholde, lett å finne fram i og med tanke på at brukerne av systemet byttes ut ofte. Det trenger ikke være nødvendig at teammedlemmer med ingeniørbakgrunn gjør dette oppryddingsarbeidet, så lenge studentene med ikke-ingeniørbakgrunn blir like godt integrert i SmallSat-teamet som bachelorgruppen har blitt i løpet av dette prosjektet.

13 Konklusjon

Bachelorgruppen anser det slik at alt som er mulig å gjøre med problemstillingen, gitt de begrensningene som eksisterer i dag, er fullført. Utviklingen og testingen av programvaren i opu-services kommer til et naturlig stopp når hardware ikke er tilgjengelig enda. Gruppen har forsøkt å legge godt til rette for at arbeidet med seriell kommunikasjon i HYPSO-2 lett skal kunne tas opp av andre som kommer inn i HYPSO-prosjektet ved et senere tidspunkt. Det er forklart nøye hvordan videre integrasjon og testing skal foregå når utviklingen av HYPSO-2 går videre inn i nye faser med riktig hardware.

Basert på testingene nevnt i kapittel 8.5.1 konkluderer gruppen med at det er trygt å anta at seriell kommunikasjon med CSP over UART fungerer etter hensikten, selv om det er nødvendig å få bekreftet dette 100% ved å utføre fullstendige tester på target hardware med fysiske RS-422-grensesnitt integrert på HYPSO-2 sin OPU. Endringene gruppen har gjort i hypso-SW tilfredsstiller kravene om bakoverkompatibilitet med HYPSO-1.

Gruppen har lagt et grunnlag i programvaren hypso-SW som åpner for kommunikasjon over RS-422. Selve databussen skal i seg selv sørge for forbedret overføringsrate, men resultatene fra testene utført i løpet av bachelorprosjektet indikerer også forbedret overføringsrate fra datalink-protokollen. Fordi gruppen anser testresultatene på tallverdiene av netto dataoverføring som upålitelige, må det utføres nøyere tester på target hardware når dette er tilgjengelig for å få konkrete og brukbare tall på optimaliseringen av overføringsraten i datalinklaget og de øvre protokollagene.

Gruppen anser oppdateringen av libcsp som et av de viktigste bidragene til hypso-SW som et resultat av bachelorarbeidet, selv om denne oppdateringen ikke var nødvendig for å løse den opprinnelige problemstillingen til oppgaven. Et av de største bibliotekene brukt i hypso-SW er nå oppdatert med forbedret og sikrere funksjonalitet og er lettere å vedlikeholde i fremtiden for SmallSat SW-team.

14 Ordliste

Baudrate	Refererer til hvor mange "symboler" som blir sendt per sekund, det vil si en hvilken som helst signalendring; binær eller ikke-binær.
Bitrate	Refererer til hvor mange bits som blir sendt per sekund.
Branch	Kopi av et repository som lager en ny versjon innad i repositoriet og tillater nye endringer, uten å skade hovedversjonen.
CAN	Både en signalstandard og en protokoll. (Datalink og Physical layer)
CSP	CubeSat Space Protocol. En modifisert TCP/IP modell med mindre overhead, laget for små nettverk med mindre ressurser.
Datarate	Datarate er total mengde data som kan bli sendt over en gitt kanal. Om en buss har høyere bitrate enn datarate, vil informasjon fortsatt bli sendt i hastigheten dataraten bestemmer.
Fork	En fullstendig kopi av et repository som lagres separat. En fork kopierer også alle eksisterende branches i repoen.
GitHub	Plattform for å distribuere og samarbeide på programvare.
Grensesnitt/ interface	Kontaktpunkt mellom to enheter/systemer.
Hypso-cli	HYPSO command line interface. Dette er kontrollpanelet en bruker for å operere og kommunisere med satelitten.
Hypso-SW	Software-repoen til HYPSO-prosjektet. All software for satelitten, testing og mer ligger her.
Issue	Kan sees på som en konkret arbeidsoppgave opprettet i GitHub.
KISS	"Keep it simple, stupid". En datalink-protokoll designet for å kunne enkelt implementeres i innvevde systemer.
Lidsat	Versjon av hardware likt som på HYPSO-1 laget for utvikling og testing på laben.
Merge	Når en branch slås sammen med en annen slik at endringer fra den ene branchen blir inkludert i den andre.
Nano Avionics	Litauisk selskap som leverer og monterer en stor andel av hardware til HYPSO-1 og HYPSO-2.
Overhead	Refererer til overflødig eller ikke-brukbar dataplass, som for eksempel header og adressefelt. Mye overhead vil føre til tregere dataoverføring.
OPU	On-board Processing Unit. Nyttelasten på satellitten som tar og håndterer hyperspektrale bilder.
Opu-services	Programvare som kjører på OPU og utfører oppgaver (services). Mottar kommandoer fra blant annet HYPSO-cli gjennom CSP-meldinger.

PC	Payload Controller. Alle steder i rapporten der PC er nevnt betyr, dette payload controller, aldri personal computer.
Pull request	En forespørsel i GitHub om å merge to brancher.
RS-422	En signaloverføringsstandard. Spesifiserer fysisk differensiell signalkarakteristikk.
Repository	Repository (repo) er et samlet fillagringssted, i oppgavens kontekst vil dette bli brukt for å referere til en samlet kodebase på GitHub.
socat	Program som oppretter virtuelle porter og kommunikasjon mellom disse.
Target hardware	Den relevante hardwaren som en spesifisert programvare skal kjøres på.
UART	Hardware som konverterer mellom parallell og asynkron seriell kommunikasjon. Gjerne en integrert enhet i innvenvde systemer.
Vcan	Virtuell CAN port.

Kilder

Birkeland R. Langer, D (2021). *Manual for Flatsat and LidSat*. Søkeord i SmallSat SharePoint: HYPSO-UM-004. (Visited on 18th May 2022).

Cavanaugh, John David (1994). *Protocol Overhead in IP/ATM Networks*. URL: http://www.sonic.net/support/docs/ip-atm.overhead.pdf (visited on 16th May 2022).

Chepponis, Mike and Phil Karn (1987). *The KISS TNC: A simple Host-to-TNC communications protocol*. URL: http://www.ax25.net/kiss.aspx (visited on 7th Apr. 2022).

Christiansen, J.D.C. (2014). *Add support for multiple host addresses #43*. Issue publisert på GitHub til libcsp. URL: https://github.com/libcsp/libcsp/issues/43 (visited on 19th May 2022).

— (2021). *Broadcast WIP #300*. Pull request publisert på GitHub til libcsp. URL: https://github.com/libcsp/libcsp/pull/300 (visited on 19th May 2022).

Christiansen, Johan De Claville (2021). *The CubeSat Space Protocol*. URL: https://github.com/libcsp/libcsp#readme (visited on 11th May 2022).

Cloudflare (2022). *What is MTU (maximum transmission unit)?* URL: https://www.cloudflare.com/learning/network-layer/what-is-mtu/ (visited on 17th May 2022).

Corrigan, Steve (2016). *Introduction to the Controller Area Network (CAN)*. URL: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf (visited on 17th May 2022).

GitHub (2022). *Understanding GitHub Actions*. URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions (visited on 17th May 2022).

Glabbeek R.J. Hofner, P. van (2017). *Split, Send, Reassemble: A Formal Specification of a CAN Bus Protocol Stack*. DOI: https://doi.org/10.48550/arXiv.1703.06569. (Visited on 17th May 2022).

Grøtte, M. E. et al (2021). *Ocean Color Hyperspectral Remote Sensing with High Resolution and Low Latency - the HYPSO-1 CubeSat Mission*. Transactions on Geoscience and Remote Sensing, volume 60. DOI: 10.1109/TGRS.2021.3080175. (Visited on 18th May 2022).

Kitware, Inc. (2022). *CMake Reference Documentation*. URL: https://cmake.org/cmake/help/latest/ (visited on 17th May 2022).

Kvaser (2022). *The CAN Bus Protocol Tutorial*. URL: https://www.kvaser.com/can-protocol-tutorial/ (visited on 12th May 2022).

libcsp (2020). *libcsp release v1.6*. URL: https://github.com/libcsp/libcsp/releases/tag/v1.6 (visited on 17th May 2022).

Luo, Luke (2014). *Linux Virtual Console(5): socat the bridging software*. URL: http://lukeluo.blogspot.com/2014/06/linux-virtual-console5-socat-bridging.html (visited on 12th May 2022).

Murray D., et al. (2012). *Large MTUs and internet performance*. IEEE Conference on High Performance Switching and Routing, volume 13. DOI: 10.1109/HPSR.2012.6260832. (Visited on 17th May 2022).

Nagy, Thomas (2021). *The Waf Book*. URL: https://waf.io/book/ (visited on 17th May 2022).

NanoAvionics (2018). *Payload Controller*. URL: https://nanoavionics.com/cubesat-components/payload-controller-1-5/ (visited on 17th May 2022).

— (2021). *Flight Computer / Interface Control Document / NA-FC-ICD-R4*. (Visited on 18th May 2022).

NovusAutomation (2022). *RS485 & RS422 Basics*. URL: https://www.novusautomation.com/downloads/Arquivos/rs485%5C%20%5C&%5C%20rs422%5C%20basics%5C%20-%5C%20english.pdf (visited on 18th May 2022).

NTNU, Institutt for lærerutdanning (2022a). *Bachelor i arkiv- og samlingsforvaltning*. URL: https://www.ntnu.no/studier/ltarkiv (visited on 17th May 2022).

— (2022b). *Master i arkiv og dokumentasjonsforvaltning*. URL: https://www.ntnu.no/studier/mad (visited on 17th May 2022).

Øverby, H. (2021). *OSI (datakommunikasjon)*. URL: https://snl.no/OSI_-_datakommunikasjon (visited on 17th May 2022).

Python Software Foundation (2019). *Sunsetting Python 2*. URL: https://www.python.org/doc/sunset-python-2/ (visited on 17th May 2022).

Rieger, Gerhard (2022). *socat - Multipurpose relay (SOcket CAT)*. URL: https://linux.die.net/man/1/socat (visited on 12th May 2022).

scrum.org (2022). *WHAT IS SCRUM?* URL: https://www.scrum.org/resources/what-is-scrum (visited on 11th May 2022).

Techopedia (2012). *Backward Compatible*. URL: https://www.techopedia.com/definition/4230/backward-compatible (visited on 17th May 2022).

Vedlegg

Implementering av CubeSat Space Protocol over UART i programvare for minisatellitt

Oppgaven

HYPSO-2 er en minisatellitt under utvikling av NTNU Small Satellite Lab (SmallSat) med mål om å skytes opp i verdensrommet i 2024, og ta hyperspektrale bilder av algekolonier i havet.

Oppgaven består av å utvide programvaren til satellitten til å kunne sende data over databussen RS-422. Dette vil forbedre den interne overføringsraten på HYPSO-2. For å implementere kommunikasjon over RS-422, brukes datalinkprotokollen «Keep it simple, stupid», KISS.

Målet med oppgaven er å kunne sende data over CubeSat Space Protocol (CSP) som er et nettverksprotokoll, over et fysisk UART/RS-422-grensesnitt ved å legge til en kompatibel datalinkprotokoll, KISS, i programvaren.

Filosofi

HYPSO-1 (allerede skutt opp i bane) og HYPSO-2 (ikke ferdigbygd) deler samme programvare, selv om de to satellittene skal ha forskjellig funksjonalitet. Oppgaven går ut på å implementere KISS-protokollen i HYPSO-2, men all utvikling må være kompatibel med begge satellittsystemene.

Programvaren skal også lastes opp på HYPSO-1 som allerede er i bane og som ikke skal ta i bruk endringene. Derfor er det kritisk at all programvare fungerer som forventet og at omfattende testing blir gjort på både hardware- og softwarenivå for å verifisere dette.

Prosess

Utviklingen av programvare ble utarbeidet i 5 steg:

1. Opprettet et virtuelt testmiljø for lokal utvikling og testing av kode.
2. Utvikling av programvare til bakkestasjonen for lettere utvikling og testing av kommunikasjon.
3. Implementering av KISS-protokollen i satellittens programvare.
4. Testing av programvaren med softwarebaserte tester.
5. Testing av programvaren på hardware.

Resultater

- Bacheloroppgaven har resultert i tillegg i programvaren til HYPSO-1 og HYPSO-2 som åpner for kommunikasjon med CSP-datapakker (nettverkslaget) over RS-422/UART (fysiske laget) ved å implementere KISS-protokollen (datalinklaget).
- Teori og foreløpige tester indikerer at denne endringen i programvare vil forbedre overføringsraten både i det fysiske laget og i datalinklaget.
- Forbedring i overføringsrate vil sørge for mindre effekttap når satellitten er operativ.
- Alle endringer gjort i programvaren for å forbedre HYPSO-2 er bakoverkompatible med HYPSO-1.

Bachelorgruppe E2219: Magnhild Eeg, Simen Løcka Eine, Ken Robin Dugan

NTNU

B Hardware test guide

For å følge testprosedyren er det nødvendig å følge instruksjonene i "Manual for Flatsat and LidSat". Dette dokumentet kan søkes opp i SmallSat SharePoint med søkeord HYPSO-UM-004. (Birkeland, 2021)

For å kunne teste kommunikasjon over KISS på RS-422, må en bruke en RS-422/USB-converter. Gruppen har lagt frem et forslag til converter som kan brukes for denne testen:

- USB/RS-422-converter: FTDI USB-RS422-WE-1800-BT

Sørg for at drivere for den converteren som er valgt er satt opp.

Disse converterne må kobles mellom RS-422-porten på OPU og USB-port på en datamaskin for å teste kommunikasjonen. Det anbefales å bruke en workstation koblet opp mot hardware på SmallSat-laben i stedet for sin egen laptop.

- Følg prosedyren i "Manual for Flatsat and LidSat" punkt 2.1 "First time setup" for å forsikre om at hardware er satt opp riktig.
- Sørg for at target hardware har strøm ved å sørge for at EPS kanal 10 er på. Dette kan gjøres gjennom kommandoene `epson`, `epsoff` og `epslist` som beskrevet i 2.1 i testmanualen.
- Logg inn på operativsystemet til target hardware og last opp riktig versjon av OPU-services.
- Start OPU-services med både et CAN og et KISS interface. Dette gjøres med kommandoen:

  ```
  opu-services 12 can0=0/1,/dev/ttyPS0=16/1
  ```

  5 `/dev/ttyPS0` er i dette eksemplet debug-porten på flatsat HYPSO-1. Denne adressen må sannsynligvis endres når det testes på HYPSO-2 hardware. Riktig seriell port burde ligge under /dev og kan sannsynligivs finnes med kommandoen:

  ```
  ls  /dev | grep tty
  ```

- Åpne to nye terminalvinduer for to hypso-cli. Start en hypso-cli med CAN interface med kommandoen:

  ```
  hypso-cli 10 -c can0=0/0
  ```

- Start en hypso-cli med KISS interface med kommandoen:

  ```
  hyspo-cli 20 -k /dev/ttyUSB0=0/0
  ```

  Det er ikke gitt at `/ttyUSB0=0/0` er riktig port. Dette skal være USB-porten som er påkoblet converteren som kommuniserer med RS-422 på target hardware.

- Ping fra begge hypso-cli til CSP-id 12 for å bekrefte kommunikasjon med OPU-services.

  ```
  csp ping 12
  ```

  Det må være laget et testfil for å sjekke at opplastning og nedlastning fungerer. Denne testfilen må være over 1KB og ligge i samme mappe som der en har åpnet hypso-cli. I eksempelkommandoene under er denne testfilen kalt "test-file.txt". Et eksempel på en mulig testfil er en tidligere hypso-cli log som en har endret navn på. Når det er bekreftet at kommunikasjon mellom hypso-cli og OPU-services fungerer, kjør følgende kommandoer fra hver hypso-cli.

  kommandoer fra virtuelt test script:

```
opu status
opu list
opu upload test-file.txt opu-test-file.txt
opu download opu-test-file.txt test-file.txt-dl
ft check local INTEGRITY test-file.txt-dl.download-format
shell remote oneshot 12 1 rm opu-test-file.txt*
shell rm test-file.txt-dl*
```

`opu status` og `opu` list sender data som viser at kommunikasjonen funker som den skal.

`opu upload` og `opu download` er for å få data om filoverføring og for å vise at dette fungerer som forventet. Disse gir tilbakemeldinger underveis i filoverføringene og vil gi feilmeldinger hvis det oppstår problemer.

`ft` check sjekker at det ikke er feil i filoverføringen. Resultat av feil i filoverføringen blir markert med X.

`shell remote` og `rm` er for å slette resultatfilene av opplastnigngen og nesdlastningen

`test-file.txt` er navnet på testfilen som blir lastet opp til OPU-services og lagres der under navnet `opu-test-file.txt`. Dette navnet brukes under nedlasting fra OPU-services og lagres tilbake på datamaskinen som `test-file.txt-dl`.

- For å skru av OPU brukes disse kommandoene i hypso-cli:

  ```
  opu shutdown
  ```

- Avslutt hypso-cli. Skru av stømforskyning til hardware med å kjøre kommandoen:

  ```
  epsoff 10
  ```

Merk at disse kommandoene og oppsettet ble testet på FlatSat tidlig 2022. Ta hensyn til mulige endringer i oppsett.

C Software testscript

Dette er et testscript (skrevet i bash) for å virtuelt teste kommunikasjon mellom opu-services og hypso-cli med CSP over KISS og CAN. Det vil si det ikke er nødvendig med tilgang til target hardware for å teste kommunikasjonen.

Sett opp testmiljø på Linux
For å sette opp virtuell CAN må vcan være tilgjengelig. For å sette opp vcan kjøres følgende kommandoer i terminalen:

```
sudo ip link add vcan0 type vcan
sudo ip link set vcan0 up
```

Etter dette skal vcan0 være tilgjengelig som en virtuell can. Dette kan verifiseres ved å se om vcan0 dukker opp etter kommandoen:
`ip link`.

Programmet Socat må være installert for å kunne åpne de virtuelle seriellportene som brukes i scriptet. I Linux-distribusjonen Ubuntu kan dette programmet installeres ved kommandoen:
`sudo apt install socat`

Sett opp testscriptet
For å sette opp testscriptet lages det en mappe i hypso-SW mappen. Denne mappen må inneholde filene `test-fil.txt`, `commands.txt` og `virtual-kiss-test.sh`.

`test-fil.txt` er en selvvalgt tekstfil på over 1 kb som brukes for test av filoverføring.
Innholdet i vedlegg D må lagres som filen `commands.txt`.
Innholdet i vedlegg E må lagdes som filen `virtual-kiss-test.sh`. I tillegg anbefales det å gjøre dette programmet kjørbart som en "executable" ved å kjøre kommandoen:
`chmod +x virtual.kiss-test.sh`

Sørg for at hypso-SW er bygget på standard måte og de kjørbare programmene ligger tilgjengelig i "build" mappa. Om det nødvendig kan variabler i filen `virtual-kiss-test.sh` konfigueres for eget oppsett.

Kjør testscript
For å kjøre testscriptet, sørg for å være i samme mappe som filene beskrevet over og kjør deretter kommandoen:
`bash ./virtual-kiss-test.sh`

Programmet burde da starte og gi tilbakemelding om framdriften og resultatene gjennom terminalen.

Kjør testscript i GitHub Actions
For å kunne kjøre dette test-scriptet i GitHub Actions er det satt opp en "workflow" som kan integreres i GitHub Actions i hypso-SW. For å kjøre denne workflowen må det legges inn en fil under mappa `.github/workflows`. Innholdet i denne filen må lagres som i vedlegg F men navnet på filen kan være egendefinert. Etter dette er gjort vil den nye relevante testen komme opp under "Actions"-fanen på GitHub-siden til hypso-SW og resultatene av testen kan vises der. Denne testen vil kjøres hver gang det lastes opp kode til en branch ved navn `kiss-test-framework` på hypso-SW GitHub repositorien.

D commands.txt

```
# hypso cli csp commands
csp ping 12
opu status
opu list
opu upload test-file.txt opu-test-file.txt
opu download opu-test-file.txt test-file.txt-dl
shell remote oneshot 12 1 rm opu-test-file.txt*
```

E virtual-kiss-test.sh

```
#!/bin/sh
BUILD_PATH=../build/x86
SERIAL_PORT_1=./dev_r
SERIAL_PORT_2=./dev_w
CAN_DEVICE=vcan0
DOWNLOAD_FORMAT="test-file.txt-dl.download-format"
OPU_LOG=opu-out.log

check_file_integrity () { # Check integrity of file download and print result
  if [ -f $DOWNLOAD_FORMAT ];
  then
    # Run check on local file and fail if it containns X (error in file)
    if [[ $(echo "ft check local INTEGRITY $DOWNLOAD_FORMAT" | \
            $BUILD_PATH/hypso-cli 10) == *X* ]];
    then
      echo "ft check failed: error in file"
    else
      echo "ft check success"
    fi
  else
    echo "ft check failed: $DOWNLOAD_FORMAT does not exist"
  fi
}
clean_directory() { # Make sure to delete files before test
  echo "Remove download files"
  rm ./*.txt-dl*
}

echo "Remove hypso-cli logs"
rm ./*hypso-cli.log

echo "Set up virtual serial ports"
socat -d -d pty,rawer,link=./dev_r pty,rawer,link=./dev_w &

echo "Start opu-services and log stdout to $OPU_LOG"
$BUILD_PATH/opu-services 12 $CAN_DEVICE=0/1,$SERIAL_PORT_1=16/1 > $OPU_LOG &

echo "Wait for opu-services to start"; sleep 10

echo "Test with KISS"
clean_directory
# Run commands in hypso-cli
cat ./commands.txt | $BUILD_PATH/hypso-cli 20 -k $SERIAL_PORT_2=0/0
check_file_integrity # Do a check of the downloaded file

echo "Test with CAN"
clean_directory
# Run commands in hypso-cli
cat ./commands.txt | $BUILD_PATH/hypso-cli 10 -c $CAN_DEVICE=0/0
check_file_integrity # Do a check of the downloaded file

# Kill all subprocesses from in this script
echo "Test finished"
pkill -P $$
```

F kiss-workflow.yml

```yaml
name: Demonstration of virtual KISS test in GitHub Actions
on:
  push:
    branches: [kiss-test-framework]

jobs:
  CI_test_job:
    runs-on: [self-hosted, lidsat]
    name: A job to do CI tests
    steps:
      - name: Checkout branch
        uses: actions/checkout@v2
        with:
          submodules: recursive
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      - name: CI test runner
        uses: NTNU-SmallSat-Lab/hypso-sw-build-check@main
        env:
          GITHUB_TOKEN: ${{ secrets.G_ACCESS_TOKEN }}
      - name: Run script
        # adding repository to get output saved in stdout for id: run_script
        uses: mathiasvr/command-output@v1
        id: run_script
        with:
          run: | # run script with correct relative path
            cd workflow-test
            bash ./virtual-kiss-test.sh
            cd ..

      - name: Test opu-services over virtual KISS and CAN
        if: |
              true == contains(steps.run_script.outputs.stdout, 'fail')
        uses: actions/github-script@v3
        with:
              script: |
                    core.setFailed('Test failed')
```

Buffer to PC (and read from PC?) via RS-422 #165

rogerbirkeland commented on Mar 25, 2020

> NanoAvionics are investigating use of RS-422 as a parallel interface to CAN in order to quicker transfer payload data. Target speed is > 1 Mbps usable rate.
>
> Please do a short study on if this is feasible to implement in software.
>
> @JoarGjersund: RS-422 drivers and pin configuration (should be possible to use free GPIO already available from BoB)
> @magnudan and @magne-hov @DennisNTNU: KISS over RS-422. Is it possible?
>
> I forsee a `ft buffer`-422 function in parallel with the `ft buffer` that use CAN.

DennisNTNU commented on Mar 25, 2020

> @magne-hov is it possible to configure csp to so that we can choose between two different interfaces for the same csp address?
>
> I'm sure nanoAvionics know their stuff and wouldn't consider two parallel interfaces between the same nodes in a csp network if the routing were a problem.

rogerbirkelan commented on Mar 25, 2020

> Questions sent to NA:
>
> > • I assume CAN will be kept parallel and un-changed
> > • How should we route these CSP-packets? Will the PC_RS422-interface get its own CSP-address, or do we need to access it any other way?
> > • Is the RS-422-support in libcsp OK?
>
> (The last point should have read "Is the KISS RS-422-support in libcsp OK?")

rogerbirkeland commented on Mar 25, 2020

> From Rimantas:
>
> > Yes, CAN stays without any modifications.
> > We would need to assign separate CSP ID to communicate over RS422. Other option – you could stream data over RS422 and get acks over CAN. In this case nothing changes.
> > Yes, we would use standard CSP over RS422.
>
> I think I would prefer a separate CSP ID for RS422?

magne-hov commented on Mar 25, 2020

> Separate CSP ID for RS422 is definitely easiest and is trivial to configure. RS422 KISS driver should also be trivial to implement in userspace as long as the correct serial kernel drivers are installed in `opu-system`.
>
> Regarding the transfer speed for buffered files --- I don't know how much testing you've done @rogerbirkeland , but I imagine it needs some measuring. In contrast to FT download, each buffered packet currently waits for an acknowledgement packet before transmitting the next buffered packet. I have not measured the impact of this RTT interleaved with the packet transfers. The ACKS are short (1 byte payload), however, and increased bandwidth in the critical direction (OPU → PC) will definitely increase the overall buffer bandwidth.

evelynlimore commented on Apr 2, 2020

> How do we know when the study is done @rogerbirkeland ?

evelynlimore commented on Apr 16, 2020

> Missing:
>
> 1. HW-implementation of UART peripheral
> 2. SW of using this
> 3. wiring and PCB layout

sivertba commented on Sep 17, 2020

> No longer relevant

sivertba closed this on Sep 17, 2020

Investigate how to accommodate for multiple CSP-interfaces for OPU-services (CAN and RS-422) #624

rogerbirkeland commented on Oct 29, 2021

> **Is your feature request related to a problem? Please describe.**
> For HYPSO-2, we are planning to use both CAN and RS-422. For this to work, `opu-services` must launch two CSP-interfaces mapped to two different hardware layers. Might also need to do something with the routing table.
>
> **Describe the solution you'd like**
> When starting `opu-services`, we could perhaps to something like this:
>
> ```
> opu-services <firstID> <first_interface> <secondID> <second_interface> <routing_opitions>
> ```
>
> It must be possible to start with just one interface (either CAN or RS-422)
>
> This issue represents the needed software changes related to #165 and #482 (both closed) and NTNU-SmallSat-Lab/opu-system#195 (open at time of creation).

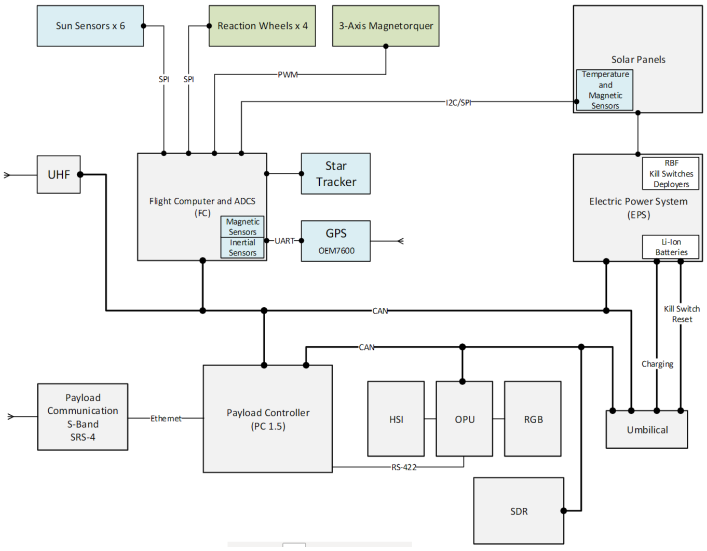
sivertba commented on Nov 1, 2021

> - Use virtual can to propagate over two CSP addresses into one (bridge(s))
> - Do we want separate CSP IDs for the two interfaces?
> - Look into CSP submodule from Magne (https://github.com/NTNU-SmallSat-Lab/libcsp)

rogerbirkeland commented on Nov 1, 2021

> Possibly a good task for Simen Løcka Eine and Magnhild Eeg?

rogerbirkeland commented on Jan 20

> Suggest to implement this so you can run opu-services on `x86` and use either virtual `COM`-port or physical `COM`-port.
>
> Some suggested tasks:
>
> - Figure out how to make (virtual) COM-interface (See
> - Figure out how to make (virtual) COM-interface #666)
> - Familarize with building `opu-services` for `x86` and testing that
> - Read up on CSP over serial (`KISS`) - and make test implemetation (See
> - Read up on CSP over serial (`KISS`) and propose an implementation in documentation #667)
> - Understand the payload-controller (*link to payload-controller*)
> - Read: HYPSO-UM-004 LidSat and FlatSat manual. HYPSO-UM-006 HYPSO-1 user manual
>
> HYPSO-2 architecture:
>
> 
>

rogerbirkeland commented on 20 Apr

> Since this issue says "investigate", it may almost be closed, at lesat by #667=

schimen closed this in #711 on 20 Apr

Figure out how to make (virtual) COM-interface #666

rogerbirkeland commented on Jan 20

> **Is your feature request related to a problem? Please describe.**
> We need to be able to test if CSP ID gets directed to CAN or RS-422 (#624). Therefore we need to make a system where communication between HYPSO-cli and OPU services can be tested virtually on the same computer. (preferably without physical connections).
>
> **Describe the solution you'd like**
> We would like to natively send and receive serial communication between HYPSO-cli and OPU services on the same computer.
>
> **Describe alternatives you've considered**
> The only alternative to virtual COM interface seems to be physical COM ports between two computers. This is a hassle and we don't want to do it.

schimen commented on Jan 27

> We have found a working solution for emulating com ports on Linux, and here is a small guide on how to use it 🤠.
> If this guide is good, is it relevant to put in the wiki somewhere?
>
> # Making a virtual COM interface
>
> Making a virtual com interface in Linux is possible in multiple different ways.
> The two options that seems most reasonable is:
>
> The kernel module tty0tty
>
> - This option aims to emulate a null modem, so it should do the job very well.
> - This option requires to build and load a kernel module, which makes the process of installing slightly complicated (although the guide for installing is very detailed)
>
> The linux utility socat
>
> - This program is a popular linux tool for sending data between two independent channels (it can be regarded as an advanced version of netcat with more functionality). A popular use for this program is to send data between two pseudo-terminals. This should emulate a null-modem and should work like previous option.
> - This option is very easy to install, as it is a common tool for linux, and can simply be installed as any other program via the package manager.
>
> Since both of these options should provide the functionality we are looking for, we decided to go for **socat**, as it is easier to install and it is a more popular option.
>
> ## Install
>
> - In Ubuntu, run the command `sudo apt install socat`
>   (In other distributions, it should be in their respective package manager.)
> - If you want to build from source, here is a nice mirror.
>
> ## Usage
>
> To open two connected virtual serial ports, simply run the command
> `socat -d -d pty,rawer pty,rawer`
>
> The terminal will print the two connected ports, and they will communicate.
>
> **Options**
>
> Socat is called like this `socat [options] <address> <address>`
>
> - In socat, the `-d` parameter determines how much it should print. Two times (`-d -d`) will include notice messages, and therefore tell us which ports it opened.
>   The two adresses can have different options in them, separated with a comma (and no space)
>
> Relevant options for us can be:
>
> - `pty`: This option generates pseudo-terminals. This is what we use for our virtual ports
> - `b9600`: this option chooses baudrate. Different possible baudrates can be listed with `socat -hh | grep " b[0-9]"`
> - `rawer`: passes input and output almost unprocessed and turns of echo.
> - `echo=1`: option to turn on local echo
> - `link=/dev/ttyS0`: makes a symbolic link, useful for creating pseudo-terminals with predictable names (data can be written to the link)
>
> More options can be found in the manual (open with the command `man socat`)
>
> Here is a short blog and a short tutorial on using socat to make virtual serial ports.

rogerbirkeland commented on Jan 27

> This looks promising. Then I suggest that you move forward with running a CSP KISS-protocol ontop of this. I'm not sure if it is easier to go ahead and modify `hypso-cli`-code right away, or if it is better to make a smaller stand-alone test service first?

DennisNTNU commented on Jan 28

> I'm not sure if it is easier to go ahead and modify hypso-cli-code right away, or if it is better to make a smaller stand-alone test service first?
>
> This should be in the end in opu-services not hypso-cli right? But I agree, a smaller test program would be simpler to start with. You can create one by copying the hypso-cli main file in `apps/hypso-cli.c` to maybe `apps/csp-kiss-test.c` and removing a bunch of stuff that you don't need. Then you need to edit the `CMakeLists.txt` file by making a copy of the block of lines starting from and including line 54 (`add_executable(hypso-cli`) until including line 112 (`target_compile_options(hypso-cli PRIVATE -Werror -Wall -fopenmp)`) right below it. Also you need to changing the name from `hypso-cli` on lines 54, 110, 111 and 112 to something else, e.g. `kiss-csp-test`. Optionally you can remove any unnecessary `.c` files that you don't need.

> This might be a bit tricky to do, but if you build the software, you should be getting the program `kiss-csp-test` in `build/x86` that you can run. You can ask me on slack for clarifications or if you are stuck.

rogerbirkeland commented on Jan 28

> Good point @DennisNTNU, should be `opu-services`. But I was also thinking that if you enable `KISS` over serial for both, it might be easier to test.

sivertba commented on Feb 3

> Implementation pending, but the core task of the issue is now closed.

sivertba closed this on Feb 3

Test virtual COM ports using CSP #676

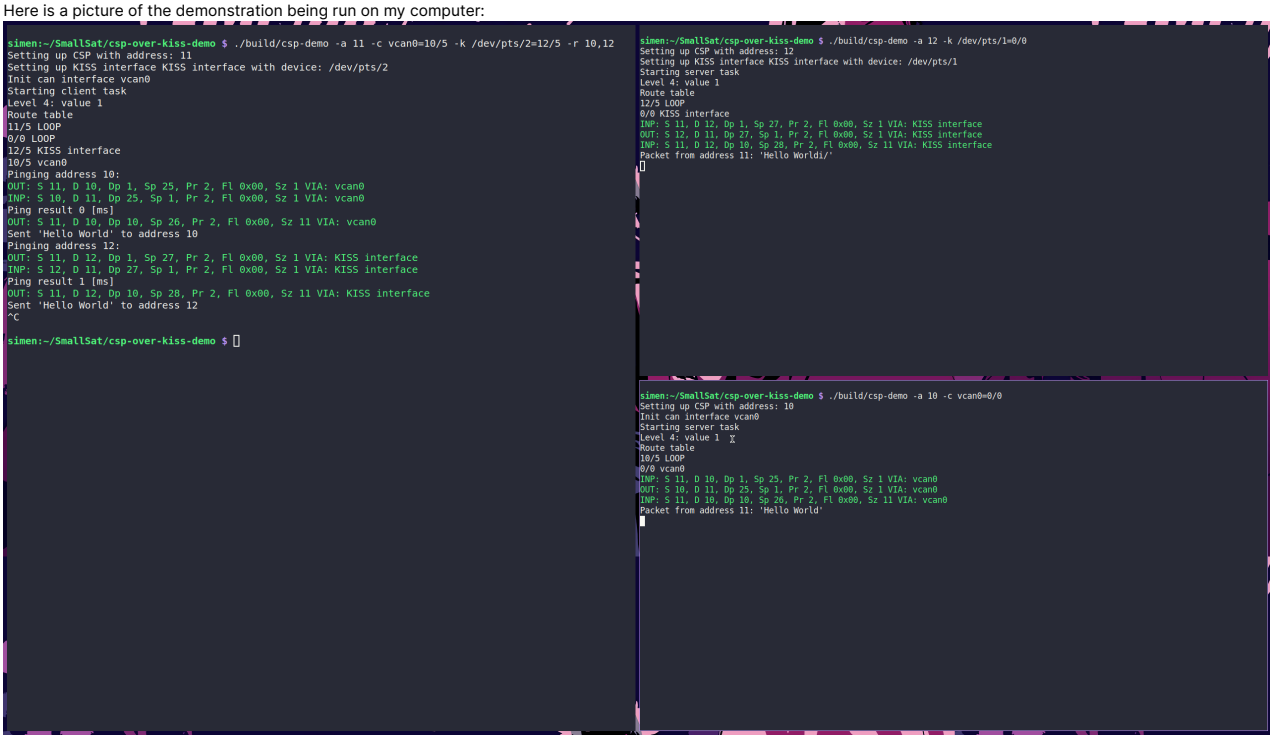
MagnhildEeg commented on Feb 3

> **Is your feature request related to a problem? Please describe.**
> We need to test if virtual serial COM ports can send CSP protocol. We will make a test program and see if the solution of #666 works. We also need to familiarize ourselves with using CSP over serial.
>
> **Describe the solution you'd like**
> We want to make an example program that sends seriall communication over the virtual ports, reads the content, and replies with a relevant answer.

DennisNTNU commented on Feb 3

> > an example program that sends [...] reads the content, and replies
>
> I have a hunch that a single program that sends, reads and replies over csp→kiss→uart may be not so easy to make, maybe its easier starting with one program that can send, then another program that can read and reply, and then see if the first program can read the reply of the second program.
>
> Relevant code in this repository that can serve as a reference or starting point:
>
> - `extern/libcsp/src/drivers/usart/usart_linux.c`
> - `extern/libcsp/src/interfaces/csp_if_kiss.c`
> - `apps/hypso_cli.c`, in the main function after the line `/* Initialise CSP */`
> - `src/cli/cli_csp.c`, the functions `cli_csp_init_can()` and `cli_csp_init_nng()`
> - Maybe also `apps/opu_services.c`, in the main function after the line `/* CSP init */`

schimen commented on Feb 16

> We managed to get a working program that is able to send messages with libcsp both over kiss and csp interface. These interfaces are routed to different addresses.
> This demonstration seems to work reliably over virtual can and virtual serial.
> We get two unwanted characters (`ï¿½`) in the end of the message sent via KISS, and we will look into that.
>
> The demonstration is located in https://github.com/schimen/csp-over-kiss-demo.
> Here is a picture of the demonstration being run on my computer:
>
> 
>
>
> We were able to run this demonstration on Linux, but not on WSL. We we did not manage to get virtual can set up there, but we will look into that.

rogerbirkeland commented on Feb 18

> Since this is a "training"-issue, we call it done.

rogerbirkeland closed this on Feb 18

Make HYPSO cli and OPU services communicate over virtual COM ports #677

MagnhildEeg commented on Feb 3

> **Is your feature request related to a problem? Please describe.**
> We want to make HYPSO cli and OPU services communicate serially over `x86`. We will use the virtual ports from #666 and possibly the results from #676 to test this. This is a step in the development of #624
>
> **Describe the solution you'd like**
> Ideally, we would like to be able to run HYPSO cli and OPU services locally while communicating over serial. In order to do this, we need to develop HYPSO cli and OPU services to support serial.

MagnhildEeg commented on Feb 16

> This issue is bigger and a bit more complicated than we first assumed. We have decided to focus on developing hypso-cli first, then later develop OPU-services. A new issue for developing hypso-cli has been made, #685. That issue consists of what we thought would be the first half of this issue.

schimen closed this in #711 on Apr 20

Make hypso-cli reliably communicate with test program over KISS #685

MagnhildEeg commented on Feb 16

> **Is your feature request related to a problem? Please describe.**
> First step of solving #677. A solution to include KISS in the code for hypso-cli might already exist. We need to look into the code for hypso-cli and make it possible to choose a CSP port to interface with KISS when using hypso-cli.
>
> **Describe the solution you'd like**
>
> 1. Read up on code for hypso-cli.
> 2. Make changes to accomodate KISS.
> 3. Make hypso-cli communicate over KISS with test program from issue #676.
>
> **Describe alternatives you've considered**
> We will be focusing on integrating KISS in hypso-cli first, then later integrate KISS in OPU services. The alternative would be to work on both programs in parallell. We believe that would be much more time consuming and difficult. We also believe that hypso-cli should be the easiest of the two programs to familiarize ourselves with.

MagnhildEeg commented on Feb 17

> We will primarily look at `hypso_cli.c`, `cli_csp.c`, `cli_csp.h`.
>
> We need to edit the function `cli_csp_init_usart` in `cli_csp.c` in several ways:
>
> - Need to change routing function from `csp_route_set` to `csp_rtable_set`.
> - Must be able to choose address and submask.
> - Must be able to choose serial port (and possibly baud rate?).
> - Must add a reaction if serial port is invalid/does not exist.
>
> The edited function must be added to a header file.
> We also need to edit the arguments and commands in `hypso_cli.c` and the command line in `cli_csp.c`.

rogerbirkeland commented on Feb 18

> Solving previous issue (#676) was a big step on the way.
> Need to refactore/re-write old code that is "hidden" in hypso-sw.
> Testing is important!

MagnhildEeg closed this in #695 on Feb 28

Check if usart device exists #697

schimen commented on Mar 1

> **Is your feature request related to a problem? Please describe.**
> When using kiss over usart in hypso-cli (as implemented in #695), you can choose a usart device that does not exist. Hypso-cli will try to open this, but it will not return an error when the device can not be opened, it will just complain to stdout and start a non working kiss interface.
>
> **Describe the solution you'd like**
> Usart is initialized in the `usart_init` function in usart_linux in libcsp. This libcsp is old, and newer versions of libcsp like the standard libcsp and Gomspaces libcsp returns an error when usart is initialized with a non existing device.
>
> We would like to update our `usart_init` function so that it also returns an error code when the usart device does not exist, similar to how it is done in `csp_usart_open` in Gomspace usart linux.
> We would then check for this in our `cli_csp_init_usart` function.
>
> **Describe alternatives you've considered**
>
> Not check if usart device exists
>
> - This option is problematic, because this can cause errors in communication, when we would think it should work.
> - This option is the simplest because we ignore the problem
>
> Update libcsp library to newer version
>
> - This option might be the most difficult and it can possible mean that we have to refactor a lot of code and bring some of our own changes to the updated library
> - This option could be the best solution as it might help with more than just checking for usart devices. It will incorporate bug fixes and features to libcsp from the last years and this could be a good thing
> - We decided this option is too complicated for our issue, but if is decided to update libcsp our issue will probably be fixed

rogerbirkeland commented on Mar 9

> One outcome *may* be that this is done, despite that this issue also will be fixed if the libcsp fork is possible.

schimen closed this in #721 on May 13

Research an update in libcsp fork #702

schimen commented on Mar 8

> **Is your feature request related to a problem? Please describe.**
> This feature request is related to our ongoing usart over csp implementation (#624).
> We have managed to open usart interfaces and communicate over these. Our problem now is that the csp code in our fork of libcsp is outdated and we are missing functionality as described in #697.
> Our motivation for updating libcsp initially was to be able to check for existing usart devices existence before opening an interface with them.
>
> *We would like to research if it is feasible and necessary to update the current libcsp fork to a newer version of libcsp.*
>
> **Describe the solution you'd like**
> We would like a discussion where we find out if it is relevant for the hypso project to update the libcsp fork.
> We would like to weigh pros and cons regarding benefits of updated code, possible unstability and amount of work put in this project.
>
> **Describe alternatives you've considered**
> To fix our speficic problem in #697, we considered to edit the current existing libcsp fork for our own needs. This would be sufficient for our single problem and would most likely be an easier solution.
> This issue addresses a broader need for updating libcsp so that we might benefit over new code changes made since last time it was updated.

schimen commented on Mar 8

> If we would choose to update the libcsp fork, we imagine it must consist of these steps:
>
> - [ ] Find the best version of libcsp to use
> - [ ] Test this version of libcsp with the rest of the project (in an isolated branch)
> - [ ] Make the new libcsp version work with the rest of the project (we will probably have to change build process and semantics among other things)
> - [ ] Thoroughly test hypso-sw with the updated libcsp

schimen commented on Mar 31

> We have started the research of an update in libcsp and we have found some important information:
>
> ### Experiences
>
> - Merging libcsp is not very complicated, as most of the additions to libcsp are done in seperate files. I have made an attempt of merging them in this fork and I only had merge conflicts in wscript and can_socketcan.c.
> - Updating libcsp will require changes in the hypso-additions to libcsp, in build process (CMake and Docker) and in relevant source files of this repository. It is difficult to say how much work this will require, but I think this should be possible.
> - I am not quite sure what the differences in the versions of libcsp are, but I imagine the latest release is a good place to start. Updating after this should be trivial as long as there are no new conflicts
>
> ### Benefits
>
> - An update of libcsp would include new developments in the libcsp project, including big refactoring of can and kiss.
> - It would be a lot easier to include potential future developments in libcsp
>
> ### Problems
>
> - I am very unsure about the amount of work and testing this update requires
> - We might loose improvements on our own fork when merging. Specifically due to merge conflicts in can_socketcan.c.
>
> ### Conclusion
>
> I will work on this for now, and see how it works out. I am working at this branch

schimen commented on Apr 4

> I have researched the update in the libcsp fork and I have managed to build hypso-sw with an updated libcsp in this branch.
> I encountered merge conflicts in wscript and can_socketcan.c. I believe the conflicts in wscripts were solved without problems, but the conflicts in can_socketcan.c are more difficult. I ended up rejecting the changes in two conflicts and:
>
> - added support for multiple socketcan interfaces
>   - I believe the Refactored all interfaces commit implemented support for multiple socketcan interfaces, and that this should be chosen. I need to test this first to verify.
> - prevent long usleep from slowing down socketcan
>   - this change limited the delay when retrying sending messages over can from 10ms to 1us. The code for this changes in later versions of libcsp, and it now waits 5ms. I believe it should be simple to reimplement the shorter wait time, and I plan to do this.
>
> I initially encountered problems in csp_if_nng.c when building libcsp, but I believe I was able to fix these issues in this commit: Update csp_if_nng.c to work with new libcsp.
>
> I also encountered different problems when building hypso-sw with the new libcsp, specifically:
>
> - Needed to add pkg-config because libcsp build system now relies on it. I added pkg-config to Docker, included the pkg-config file for libsocketcan in the libsocketcan install and added the path for this to libcsp when building.
> - There was some problem with having the install and build folder of libcsp the same. This was fixed by creating separate build and install folders for libcsp, see Isolate build and install folder for libcsp.
> - The csp_init function now also initializes buffers. I therefore moved all buffer settings to the csp_conf struct and let these be initialized in csp_init function.
> - The old way of initializing kiss is not valid anymore, and I used the new convenience function `csp_usart_open_and_add_kiss_interface` to open kiss. This enables us to check if the serial port we open kiss over exists, and call an error if it does not.
> - I hardcoded `256` to the define `HYPSO_CSP_MTU`. `HYPSO_CSP_MTU` used to be set as the value in `CSP_CAN_MTU`, but this define does not exist anymore. I don't really like this solution, but I also don't really want to mess too much with this value.

> These changes are implemented in Update hypso-sw to new libcsp
>
> Python 3 is now supported when building libcsp, maybe it is a good idea to update python when updating libcsp?
>
> What to do before making pull request:
>
> - Test that multiple socketcan interfaces works
> - Make the retry delay shorter
> - Do some more tests to see that everything actually works

schimen closed this in #721 on May 13

Add csp buffer packet overhead to kiss overflow check #3

schimen commented on 7 Apr

> **Summary of the pull request**
> This pull request adds the csp overhed to the kiss mtu when checking for overflow in KISS.
> This is because now, packets sent over KISS (including their overhead) are not allowed to be larger than the MTU.
>
> This pull request fixes an immediate problem, but this problem has been fixed in the the official libcsp.
> I am working on updating this fork to libcsp v1.6, and this would fix the issue.
>
> **What issues are related to this pull request?**
> This pull request is for making the kiss integration in opu-sevices possible.
>
> **Describe potential caveats of the pull request, if any**
> I believe the changes should be stable and not bring any problems.
> A potential caveat is the conflict between this fork and the official libcsp, I will have to remove this change when updating to the new libcsp.

DennisNTNU approved these changes on Apr 10

> Does indeed fix the overflow errors! Managed to transfer multiple files over physical UARTs.

DennisNTNU merged commit `9559848` into `NTNU-SmallSat-Lab:hypso` on Apr 10

Cli kiss #695

MagnhildEeg commented on Feb 24

> Semantic Versioning Change Type: idk
>
> **Summary of the pull request for the changelog**
> We have
>
> - updated the code in init_usart function to take device as argument and use correct routing function.
> - made changes to hypso_cli so user can choose KISS over usart as optional interface.
> - included init_usart function declaration in hypso_cli.h
>
> **What issues are related to this pull request?**
> This pull request is related to issue #685
>
> **What does your pull request address?**
> We have made changes to several files in order to implement KISS over usart in hypso-cli. The result is added functionality in hypso-cli with all previous functionality preserved. The user can now choose between can, nng and kiss from the command options.
>
> **Describe potential caveats of the pull request, if any**
> We have not updated the function generate_command_prompt to include KISS. We do not know if this is required or necessary because the program still works as intended. The way we see it, the generate_command_promot function hardcodes CSP-addresses to certain interfaces and destinations and we are unsure whether or not KISS should have a hardcoded address.
>
> We have not implemented a response for invalid usart device. The new and updated version of libcsp has updated code that has error handling for this event.
> We suggest either to edit the code in SmallSat libcsp, or update our libcsp library to a newer version from GomSpace.
> Right now, hypso-cli will open a KISS interface even if the given usart device does not exist.

schimen commented on Feb 25

> I tested hypso cli over physical uart now, and it works as expected 🤠

DennisNTNU approved these changes on Feb 25

> I did the same, using two UAB-UART dongles. The two hypso-cli instances can ping each other! Code looks good too.

MagnhildEeg merged commit `49ac5dc` into master on Feb 28

Add KISS interface to services #711

schimen commented on Mar 25

Semantic Versioning Change Type: Minor (https://semver.org/)

Summary of the pull request for the changelog
This pull request makes it possible to use a KISS interface over csp in opu-services.

What issues are related to this pull request?
This pull request implements KISS interface in opu-services, as part of #677.
KISS is already implemented in hypso-cli, so this is the last part of the issue.

What does your pull request address?
This pull request adds the possibility for opu-services (and sdr-services) to communicate over usart-interfaces via csp.
The usart interface is chosen in the argument `<CSP-DEVICE>`.
If this argument contains "can", can interface is chosen, else if it contains "/dev", usart interface is chosen.

We have also split up the initialization for can and usart into functions.
This is more readable and easier to expand on later.

Describe potential caveats of the pull request, if any

- When parsing the device, any devices that do not contain "can" or "/dev" will not open an interface and the program will fail. If there is a can device that does not contain "can" or a kiss device that does not contain "/dev", it will not work.
- Increased complexity with routing of multiple interfaces might be a problem.
  We have tried to combat this problem by defaulting to `0/0` when no routing is specified
- We chose to use comma-separated arguments (example `./opu-services 21 vcan0=0/1,/dev/pts/3=16/1`) for backwards compatibility. This might be a strange way to interface with the program and might lead to errors (for example if there are spaces between the comma)

How to test the pull request

To test this pull-request, you need to

1. build hypso-sw.
2. make sure you have two connected serial ports available.
   This can be done via virtual serial ports. To set these up, follow this comment and open the virtual serial ports with the command `socat -d -d pty,rawer pty,rawer &`.
   (remember the paths of the new serial ports)
3. make sure you have a can device to test. An easy way would be to use virtual can. Do this by running the commands:
   `sudo ip link add vcan0 type vcan && sudo ip link set vcan0 up`
4. open opu-services with can and one of the new serial ports, for example like this
   `./build/x86/opu-services 15 /dev/pts/1=0/1,vcan0=16/1`
5. open one hypso-cli with the other new serial port: `./build/x86/hypso-cli 10 -k >/dev/pts/2=0/0`
   and another hypso-cli over can: `./build/x86/hypso-cli 20 -c vcan0=0/0`
6. now you can ping opu services from hypso-cli: `csp ping 15` with both interfaces.

schimen reviewed on Mar 27 *src/services/services_csp.c*

```
+csp_iface_t* init_usart(char* usart_device)
+{
+    // Allocate memory for usart driver and interface
+    static csp_kiss_handle_t usart_driver;
+    static csp_iface_t usart_iface;
+
+    // Usart callback function
+    void my_usart_rx(uint8_t * buf, int len, void* pxTaskWoken)
+    {
+        csp_kiss_rx(&usart_iface, buf, len, pxTaskWoken);
+    }
```

schimen on Mar 27

> After some investigation we realized that nested functions are not supported in standard C, but it is supported in GCC (source).
>
> > If you try to call the nested function through its address after the containing function exits, all hell breaks loose.
> > If you try to call it after a containing scope level exits, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk.
> > If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.
>
> Our nested function `my_usart_rx` is called after its containing scope exits, but it only refers to one local variable, and it is a static variable.
> Since the static variable should be kept after the parent function exits, we believe it is safe to use a nested function here.

schimen on Mar 29

> After a good discussion with @DennisNTNU we found that a global calback function is better

MagnhildEeg reviewed on Apr 4 *cmake/libcsp.cmake*

```
list(APPEND LIBCSP_OPTIONS "--install-csp")
list(APPEND LIBCSP_OPTIONS "--with-rtable=cidr")
list(APPEND LIBCSP_OPTIONS "--enable-init-shutdown")
```

```
+list(APPEND LIBCSP_OPTIONS "--enable-if-kiss")
+list(APPEND LIBCSP_OPTIONS "--with-driver-usart=linux")
...
 if(${BUILD} MATCHES ".*build/x86$")
-  list(APPEND LIBCSP_OPTIONS "--enable-if-kiss")
-  list(APPEND LIBCSP_OPTIONS "--with-driver-usart=linux")
```

MagnhildEeg on Apr 4

> Made changes in cmake/libcsp.cmake in order to include KISS and UART for all architectures.

DennisNTNU reviewed on Apr 5

During testing I keep getting the following errors when doing an `opu download` over KISS:

```
KISS RX overflow
: ▯▯▯gKISS RX overflow
:
\dhKISS RX overflow
ou▯▯!(pKISS RX overflow
:0▯d▯KISS RX overflow
td▯▯KISS RX overflow
```

The characters before `KISS RX overflow` are deterministic for the same hypso-cli command. The command I am doing is

```
opu download 220404T205332_hypso-cli.log 220404T205332_hypso-cli.log-dl
```

which is just downloading a random log file that I had was lying around, which is `1472` bytes large, corresponging to 7 entries.

Doing an `ft check local` on the temporary download file, reveals a hint of what may be wrong:

```
(hypso-cli-CAN) ft check local ALL 220404T205332_hypso-cli.log-dl.download-format
Start entry ID: 1
Last entry ID:  7
Total entries:  7
XXXXXX.
File has 6 bad entries.
```

Only the last entry was transferred. Since pinging works, I have a hunch that packets that are too long, the KISS protocol can't handle. Maybe this is related to the 256 Byte MTU limitation? Doing the same transfer over a virtual CAN, reaveals that the packets are 262 Bytes large, here is a sample of a packet:

```
18 FD CA 00 01 00 02 00 04 00 00 00 00 00 F6 AC
E9 FD 35 F0 00 74 3A 20 28 68 79 70 73 6F 2D 63
6C 69 2D 43 41 4E 29 20 6F 70 75 20 74 6D 6C 6F
67 20 6F 66 66 0A 32 30 32 32 2D 30 34 2D 30 34
54 32 30 3A 35 35 3A 30 32 5A 20 73 74 64 6F 75
74 3A 20 53 65 6E 64 69 6E 67 20 74 65 6C 65 6D
65 74 72 79 20 6C 6F 67 20 72 65 71 75 65 73 74
0A 32 30 32 32 2D 30 34 2D 30 34 54 32 30 3A 35
35 3A 30 32 5A 20 73 74 64 6F 75 74 3A 20 3C 2D
2D 20 4C 6F 67 67 69 6E 67 20 73 74 61 74 75 73
3A 20 30 2C 20 4C 6F 67 67 69 6E 67 20 69 6E 74
65 72 76 61 6C 3A 20 33 30 0A 32 30 32 32 2D 30
34 2D 30 34 54 32 30 3A 35 35 3A 30 32 5A 20 73
74 64 6F 75 74 3A 20 28 68 79 70 73 6F 2D 63 6C
69 2D 43 41 4E 29 20 63 73 70 20 70 69 6E 67 20
31 32 0A 32 30 32 32 2D 30 34 2D 30 34 54 32 30
3A 35 35 3A 30 00
```

I am starting hypso-cli as follows

```
rlwrap ./hypso-cli 15 -c vcan0=16/1 -k /dev/ttyUSB0=0/1
```

and `opu-services` as

```
./opu-services 12 /dev/ttyUSB1
```

And they communicate over two physical USB-UART converters.

schimen commented on Apr 6

Thank you Dennis, I was able to reproduce this (over virtual serial).
I successfully downloaded an 8 byte file with `opu download`, but I got the same errors as you when I tried with a 500 byte file. With the 500 byte file, only the last packet was successfull.

I also I got overflow when running the `opu status` and `opu list` commands.

We will look into this 🤠

schimen commented on Apr 6

> After some discussion with Dennis I believe the problem is that when messages are sent over KISS, they check the maximum length of the message is the same as the kiss interface mtu variable. This is not the case as this length includes the header and id, so the length must be larger than the MTU.
> I tried to fix this by increasing the MTU in the KISS interface, but i was not able to modify this, as it is set to `KISS_MTU` in the init function, with no option for other MTU 🤔
> I don't have a good way of solving this in hypso software at the moment, and I believe we will have to edit libcsp to make it work properly.
> My proposal is to edit the define `KISS_MTU` in csp_if_kiss.c to `#define KISS_MTU CSP_BUFFER_PACKET_OVERHEAD + 256`. This will add the extra overhead bytes and avoid the current overflow issue.
> I have tried this and it works.
>
> I want to add that this has been fixed in libcsp, and the implementation for this check in libcsp v1.6 is more robust. If #702 works out and is implemented, the new libcsp will cover this and won't need to edit KISS_MTU anymore.

schimen commented on Apr 7

> I just made a pull request on libcsp with my preferred solution, please review @rogerbirkeland or @DennisNTNU😇

DennisNTNU approved these changes on Apr 10

> Working well now! I managed to run `opu-services` with both a CAN and an UART interface, and communicate with it over both interfaces with two `hypso-cli`s.
>
> Then I did some speed benchmarking using `opu download` and various 'period' values, here are the results transferring a file of size `1681440 B` ~ `1.6 MB`:
>
> Period [ms]
>
> | Period [ms] | Duration | Effective Speed [KB/s] | Effective speed Kb/s |
> |---|---|---|---|
> | 6 | 00:47 | 35.1146 | 280.9169 |
> | 5 | 00:40 | 41.1601 | 329.2808 |
> | 4 | 00:37 | 44.6928 | 357.5425 |
> | 3 | 00:37 | 44.6758 | 357.4067 |
> | 2 | 00:37 | 44.6750 | 357.4000 |
> | 1 | 00:37 | 44.6657 | 357.3259 |
>
> After changing bitrate from 500k to 1M (HYPSO_USART_BAUD in HYPSO.h)
>
> | Period [ms] | Duration | Effective Speed [KB/s] | Effective speed Kb/s |
> |---|---|---|---|
> | 6 | 00:47 | 35.1146 | 280.9169 |
> | 5 | 00:40 | 41.1601 | 329.2808 |
> | 4 | 00:37 | 44.6928 | 357.5425 |
> | 3 | 00:37 | 44.6758 | 357.4067 |
> | 2 | 00:37 | 44.6750 | 357.4000 |
> | 1 | 00:37 | 44.6657 | 357.3259 |
>
> So at 500Kb/s we only effectively get 357Kb/S. And at 1000Kb/s we only effectively get 659Kb/s. So I am puzzled and a bit disappointed that there is like a 30%-35% overhead, even though the csp and file transfer overheads are supposed to be much smaller. But possibly improving the speed is for a future issue.
>
> Also, when testing this, I discovered that what is called a the `period` argument in the `opu download` command, is actually a delay between packets, not technically a period.

schimen merged commit `88492d1` into master on Apr 20

Update libcsp #721

schimen commented on Apr 21

Semantic Versioning Change Type: Minor (https://semver.org/)

Summary of the pull request for the changelog
Update the libcsp library to v1.6:

- Update build process of libcsp
- Move from old libcsp functions to new equivalent functions where it is relevant

This gives us benefits of new additions from upstream libcsp.

What issues are related to this pull request?
This pull request resolves #697 because usart device check is a feature of the updated libcsp version.
This pull request will also make the creation of can and kiss interfaces easier and better.

This pull request depends on this pull request in our libcsp fork. The pull request in the libcsp fork needs to be merged before this pull request can be merged, because we need to update the git submodule.

What does your pull request address?
This pull request updates our libcsp fork that has not been updated for quite a while. The official libcsp repository is actively maintained now, and we can benefit on including this development in our own libcsp fork.

- Updating libcsp now will make it easier to eventually update to newer versions of libcsp in the future
- We can move from deprecated `csp_can_socketcan_init` and use newer init functions for can and kiss. This means that there is no limit on the amount of can and kiss interfaces we use.
- We can check if kiss was initialized correctly as described in #697
- We can move to python3 for building libcsp

Describe potential caveats of the pull request, if any

To update libcsp I had to resolve two merge conflicts

- `wscript`: I had to accept incoming changes in `wscript`, but I edited it to not include nng drivers when not specified. This will most likely be changed in the next libcsp version.
- `can_socketcan.c`: I had to accept incoming changes, but I believe the incoming changes solves the problems our changes have tried to aaddress better.
- There might be funny behaviour or errors I have not discovered, because some fundamental changes in libcsp was made. This pull request requires testing all apps that use csp.

How to test the pull request
To test the pull request, make sure the git submodule for libcsp is updated by running `git submodule update`.
Then rebuild hypso-sw and test the apps in the way you usually would run them.

DennisNTNU approved these changes on May 6

Compiled and tested both on WS2 and lidsat. To compile, the docker image needs to be rebuild, since the `dockerfile` was updated. Compilation went without issues.

On WS2, tested new `m6p-time-sync` and `opu-services` with a new `hypso-cli`. Commands that I tested were:

`csp buffers`, `csp hello`, `csp mem`, `csp ping`, `csp reboot`, `csp shutdown`, `csp uptime`, `opu download`, `opu upload`, `shell remote`, `shell remote oneshot`, `opu restart`, `opu caminfo`, `hsi capture`, `opu git`, `opu lastcmd`, `opu telemetry`, `opu status`

and all of them worked as expected, except `csp reboot` and `csp shutdown`, which prints an error that this option is not enabled. So to make them work, a compiler flag would probably need to be set somewhere. I think its OK without those, or it can be a future issue to enable them. Maybe other people in the team has other opinions.

What also still worked was interfacing with the EPS on WS2, both from `hypso-cli` and from `opu-services`, meaning `opu-services` is still able to control the EPS power output states on boot, capture and shutdown.

On lidsat I tested a new `hypso-cli` against an old `opu-services` and no issues were observed there. I did `csp ping`, `opu telemetry`, `opu status`, `opu settime`, `shell remote oneshot` and `opu upload`. What I uploaded was a new `m6p-time-sync` and a new `opu-services` with the new version of libcsp, and `opu restart`ed into it. `m6p-time-sync` worked on the lidsat as well.

Then I did an `ft buffer file` which worked flawlessly as well, and I downloaded the same file from the PC with a new `hypso-cli` and the downloaded file was identical to the original.

So with all these tests, I am convinced that the new libcsp version is

1. Working on target hardware
2. Still compatible with the nanoAvionics' components
3. `hypso-cli` backwards compatible to older `opu-services`

I tested the main functionality we need (remote shell, status & telemetry, upload and download, capture and buffering), observed no issues, and thus I think this is good to merge.

Schimen commented on May 6

Thank you for testing! Great to hear that this works ❤

Regarding:

> and all of them worked as expected, except csp reboot and csp shutdown, which prints an error that this option is not enabled.

> The option `--enable-init-shutdown` was removed in this pull request. I will have a look into it when I have time, I believe there are changes for csp reboot and csp shutdown we have to look into.
> Or maybe we just have to enable another build-flag, which would be pretty great 😄

schimen commented on May 8

> I investigated the reboot and shutdown functions and I found that we need to specify the functions that the system uses to reboot and shutdown.
> I added the posix functions to services using the standard system commands in 689b319.
> This didn't really work as I expected on my personal laptop, but I tested it on flatsat with WS2 over can0 and both reboot and shutdown works as expected here.
>
> @DennisNTNU Is this ok to merge?

schimen commented on May 9

> I switched from python2 to python3 in cmake for building libcsp. Python 3.8.10 is available for me in Docker so I figured it should be fine to ask for at least Python 3.8

DennisNTNU commented on May 13

> OK to merge! `csp shutdown` and `csp reboot` work on target hardware now. Need to build from clean (e.g. after `rm -r build`) for it to work due to the python version change. If coming from an earlier branch, also a rebuild of the docker container may be necessary.
>
> Testing with the SDR system will be a future issue.

schimen merged commit `a571c51` into master May 13

NTNU

Kunnskap for en bedre verden