POLITECNICO DI TORINO

III Facoltà di Ingegneria

Corso di laurea in Ingegneria delle Telecomunicazioni

Tesi di Laurea Specialistica

Sottosistema di telecomunicazione UHF e protocollo di comunicazione per nanosatelliti modulari

Relatori:

Prof. Leonardo Reyneri

Prof. Claudio Sansoè

Candidato:

Antonio Aru

Luglio 2012

Ai miei genitori…

Sommario

Nel corso degli ultimi anni l'opportunità rappresentata da vettori lanciatori a basso costo ha reso maggiormente accessibile l'ambiente spaziale anche a organizzazioni non dotate di elevati budget, quali le piccole e medie imprese e le università. Ciò ha portato nel 1999 alla definizione dello standard CubeSat. Il Politecnico di Torino nel 2006 ha terminato lo sviluppo di un proprio nanosatellite (PiCPoT), di dimensioni poco più grandi dello standard CubeSat, il cui principale obiettivo era la valutazione dell'impiego dei componenti COTS (componenti commerciali). Il progetto successivo a PiCPoT è il progetto AraMiS, nato nel 2006 e attualmente attivo, all'interno del quale si è integrato il presente lavoro di tesi. Il principale obiettivo del progetto AraMiS è l'applicazione del concetto di architettura modulare che porta vantaggi quali tempi di realizzazione rapidi e costi limitati. AraMiS consiste di un insieme di moduli, chiamati *tile*, pre-assemblati e pre-testati che hanno la caratteristica di riusabilità a seconda delle necessità della specifica missione. Tra i diversi moduli, in particolare, vi è il sottosistema di telecomunicazione che ha il compito di ricevere comandi in uplink e trasmettere dati in downlink, tenendo in considerazione la limitata disponibilità energetica a bordo e il ridotto periodo di visibilità tipico delle orbite LEO (minore di 20 minuti). Per superare quest'ultima limitazione, si è deciso di partecipare al sistema GENSO, un progetto dell'European Space Agency (ESA). Il suo scopo è quello di utilizzare le ground station radioamatoriali, distribuite per tutta la Terra e raggiungibili via Internet, come se queste fossero una interfaccia radio remota. Un sistema di questo tipo permetterebbe di estendere il periodo di visibilità dei satelliti al proprio Mission Controller teoricamente fino alle 24 ore.

Il sottosistema di comunicazione di AraMiS utilizza due differenti canali a banda stretta, completamente indipendenti e ridondanti sulle bande radioamatoriali UHF (435 MHz) e Banda S (2440 MHz). In questo lavoro di tesi si è considerato il modulo di comunicazione satellitare in banda UHF, con una maggiore concentrazione sul protocollo di comunicazione. L'attività è partita dall'analisi del contesto radioamatoriale per capire le caratteristiche dei sistemi a cui rendersi compatibili (come si vedrà,

modulazione FSK e alcune funzioni dei livelli inferiori del protocollo AX.25 con data rate a 9600 bps). Successivamente si è passati ad una attività di laboratorio nella quale si è presa visione di un prototipo precedentemente sviluppato per AraMiS (abbastanza rappresentativo di quella che potrà essere la versione definitiva), e si è costruito un secondo prototipo. Questi prototipi sono stati programmati e sono stati eseguiti dei test di trasmissione e ricezione di frame AX.25 (livello data-link del modello OSI) tra un prototipo e l'altro e tra un prototipo ed una stazione radioamatoriale allestita in laboratorio. Sulla base della documentazione iniziale, dell'esperienza ricavata dalle sperimentazioni effettuate, delle riunioni periodiche con gli altri membri del team e dell'utilizzo dell'ambiente di sviluppo UML *Visual Paradigm*, si è avviato un processo di progetto di un protocollo ad-hoc che si poggiasse su quello di livello data-link per conferire efficienza e affidabilità al sistema di comunicazione. La definizione di questo protocollo rappresenta il risultato principale dell'attività. A conclusione del lavoro svolto si sono valutate analiticamente le prestazioni del protocollo proposto.

Il documento è suddiviso nei seguenti capitoli:

- Capitolo 1: fornisce alcuni elementi a riguardo dello standard CubeSat, dei componenti COTS e del concetto di modularità. Successivamente viene fornita una descrizione generale del progetto AraMiS.
- Capitolo 2: descrive il progetto GENSO, la sua architettura, le sue potenzialità secondo lo stato di sviluppo alla data attuale.
- Capitolo 3: contiene una ricerca sull'attività satellitare dei radioamatori, volta ad identificare vincoli e gradi di libertà per il progetto del nostro sottosistema di comunicazione affinché sia a loro compatibile.
- Capitolo 4: documenta la scheda utilizzata e fornisce il link budget della missione tipo.
- Capitolo 5: descrive in dettaglio il protocollo AX.25.
- Capitolo 6: descrive le funzioni svolte dal firmware e le prove eseguite.
- Capitolo 7: descrive il protocollo proposto.
- Capitolo 8: riporta l'analisi delle prestazioni del protocollo proposto e fornisce ulteriori spunti per il proseguimento del lavoro.

Indice

Sommario **I**

1 Introduzione **1**

1.1 CubeSat, componenti COTS e modularità ... 1

1.2 AraMiS ... 3

2 Progetto GENSO **7**

2.1 Progetto GENSO ... 7

2.2 Architettura e funzionamento del sistema GENSO ... 8

2.2.1 Authentication Server (AUS) ... 9

2.2.2 Ground Station Server (GSS) ... 10

2.2.3 Mission Control Client (MCC) ... 10

3 Protocolli radioamatoriali **13**

3.1 Introduzione ... 13

3.2 Definizioni preliminari ... 14

3.3 Satelliti radioamatoriali ... 15

3.4 Satelliti radioamatoriali universitari ... 24

3.5 Contesto odierno ... 26

4 Modulo di comunicazione a 437 MHz del satellite **29**

4.1 Componenti utilizzati (COTS) ... 29

4.1.1 Transceiver: TI CC1020 ... 29

4.1.2 Microcontrollore: TI MSP430 ... 32

4.2 Interfaccia di configurazione e generalità sul processo di configurazione ... 34

4.3 Interfaccia dati ... 37

4.4 Prototipo della On-Board Radio Frequency module ... 38

4.5 Link-budget ... 41

5 Il protocollo di comunicazione AX.25 **45**

5.1 Introduzione ... 45

5.2 Frame in uso nel protocollo AX.25 ... 46

Campo Flag ... 46

Campo Addess 47
Campo Control 48
Campo PID (solo I o UI frame) 48
Campo Information (solo I o UI frame) 49
Campo FCS 49
Bit stuffing 49
6 Firmware e test dell'On-Board Radio Frequency module 51
6.1 Descrizione del programma 51
6.1.1 main.c 52
6.1.2 ax25.c 60
6.1.3 cc1020.c 60
6.1.4 SPI.c 61
6.1.5 uart.c 62
6.1.6 timer.c 62
6.2 Collaudi e test 63
7 Il protocollo elaborato 65
7.1 Introduzione 65
7.1 Definizione del formato dei pacchetti 68
7.1.1 Pacchetti in uplink 69
7.1.2 Pacchetti in downlink 72
7.2 Funzionamento del protocollo 76
Basic Protocol - Short command 77
Basic Protocol - Short command with loss in uplink 79
Basic Protocol - Short command with loss in downlink 80
Basic Protocol - Long command with subsequent request command 81
Basic Protocol - Long command with ACK loss in downlink 83
Basic Protocol - Combined operations 84
Basic Protocol - Download with Fragmentation 85
7.3 Funzioni definite 86
7.4 Cenni sulla gestione della memoria 88

8 Analisi delle prestazioni **89**
8.1 RDT per i comandi SHORT 89
9 Conclusioni **95**
Appendice A Codici sorgente **97**
A.1 main.c 97
A.2 main.h 99
A.3 ax.25.c 100
A.4. ax25.h 106
A.5. cc1020.c 107
A.6. cc1020.h 118
A.7. SPI.c 122
A.8. SPI.h 123
A.9. uart.c 124
A.10. uart.h 125
A.11. timer.c 126
A.12. timer.h 127
Appendice B Tabella dei comandi e dei dati **129**
Appendice C Calcolo della durata del periodo di visibilità **131**

Elenco delle abbreviazioni

AraMiS	Modular Architecture for Satellites
BER	Bit Error Rate
COTS	Commercial Off The Shelf
CRC	Cyclic Redundancy Check
FCS	Frame Check Sequence
FSK	Frequency Shift Keyed
GENSO	Global Educational Network for Satellite Operations
GFSK	Gaussian FSK
GSS	Ground Station Server
HP	Housekeeping Processor
JTAG	Joint Test Action Group
KISS	Keep It Simple, Stupid
LEO	Low Earth Orbit
MCC	Mission Control Client
MCU	Micro-Control Unit
NRZ	Non-Return to Zero
NRZI	Non-Return to Zero Inverse
OBC	On-Board Computer
OBRF	On-Board Radio Frequency (module)
PA	Power Amplifier
PCB	Printed Circuit Board
PiCPoT	Piccolo Cubo del Politecnico di Torino
SPI	Serial Peripheral Interface Bus
TCP	Telemetry/Command Processor
TNC	Terminal Node Controller
UHF	Ultra High Frequency
USART	Universal synchronous and asynchronous receiver/transmitter

Capitolo 1

Introduzione

L'opportunità rappresentata da vettori lanciatori a basso costo negli ultimi anni ha reso maggiormente accessibile l'ambiente spaziale anche a organizzazioni non dotate di elevati budget, quali le piccole e medie imprese e le università. Ciò ha contribuito alla nascita dell'interesse per lo sviluppo di piccoli ed economici satelliti che ha poi portato alla ricerca di una metodologia di base per l'approccio alla progettazione di tali sistemi che ne limitasse i costi.

1.1 CubeSat, componenti COTS e modularità

Un primo passo di questo processo di ricerca è stato il progetto *CubeSat* [1], nato nelle California Polytechnic State University e Stanford University. Nel 1999 il CubeSat è diventato uno standard di satellite che definisce la loro forma e dimensione (cubica con lato di 10 cm) e massa (≤1.33 Kg) e loro multipli (Figura 1.1). La definizione di questo standard permette di accorpare più satelliti all'interno della stessa interfaccia di lancio con conseguente ulteriore riduzione dei costi di lancio, riduzione del tempo di sviluppo, aumento della disponibilità dei lanci. In Figura 1.2 è mostrato un contenitore per CubeSat che può contenerne tre CubeSat 1U, oppure un 1U più un 2U, oppure un CubeSat 3U. Le sue funzioni sono quelle di proteggere le altre apparecchiature trasportate sullo stesso veicolo di lancio, presumibilmente molto più costose dei CubeSat, e di spingere i piccoli satelliti verso l'esterno del lanciatore al momento dell'estrazione. La molla visibile frontalmente viene utilizzata a tal fine.

Figura 1.1 Chassis di CubeSat nel formato 1U e nel formato 3U

Figura 1.2. Contenitore P-POD per CubeSat

Un secondo passo è stato l'impiego di dispositivi *COTS* (Commercial Off The Shelf). Con tale denominazione ci si riferisce a componenti disponibili sul mercato per l'acquisto da parte di aziende di sviluppo interessate a utilizzarli nei loro progetti. Tali componenti sono tipicamente dotati di funzionalità già attive e ottimizzate. Possono essere assemblati all'interno di specifici sottosistemi permettendo la realizzazione di

soluzioni ad-hoc e al contempo contenendo i costi di sviluppo e, dove previsto, di manutenzione. L'impiego di componenti COTS per i satelliti low-cost è ancora in via di sperimentazione. Lo studio di fattibilità è stato proprio uno degli obiettivi del nanosatellite PiCPoT, sviluppato dal Politecnico di Torino a partire dal 2004 [2].

Un ulteriore passo di questo processo di ricerca verso la riduzione dei costi è l'impiego di una architettura modulare, in particolare in riferimento ai sottosistemi elettronici del satellite. I primi vantaggi si riflettono sui costi di progetto, realizzazione e test che, nell'ottica del riuso, possono approfittare di una economia di scala. D'altra parte, fare uso degli stessi moduli in più missioni consente perfezionamenti che portano a livelli più elevati di efficienza ed affidabilità. Ciò in parte compensa le conseguenze di alcune delle caratteristiche tecniche non elevate dei componenti COTS. Altri vantaggi derivano dal ridotto *time-to-launch*, cioè il tempo impiegato a terminare la preparazione della missione dal momento in cui questa viene specificata dal committente. Tanto più il time-to-lunch è breve, tante più saranno le possibilità di accedere ai cosiddetti *piggyback payloads*. Con questo termine ci si riferisce alla capacità che risulta ancora inutilizzata all'interno dei veicoli di lancio al termine dell'assemblaggio del carico primario. Usufruire dei piggyback payloads risulta economico, ma tipicamente comporta tempi di preavviso brevi. Lo studio dell'impiego della modularità è uno degli obiettivi dei nanosatelliti AraMiS in via di sviluppo al Politecnico di Torino dal 2006.

1.2 AraMiS

I satelliti di classe AraMiS (acronimo di Architetture Modulari per Satelliti) [3] adottano una vera architettura modulare i cui elementi, chiamati *tile*, utilizzano componenti COTS. Pur trattandosi di nano e micro-satelliti, AraMiS non rispetta lo standard CubeSat in quanto le possibili applicazioni che vuole coprire richiedono prestazioni più elevate di quelle che al momento si possono ottenere con tale standard. Sono quindi previste diverse forme (cubo, parallelepipedo, prisma esagonale o ottagonale) e dimensioni. Per quelli di dimensione più piccola, l'idea di base è quella di attribuire alle tile progettate per essere collocate nelle facce esterne, anche un ruolo

meccanico, oltreché funzionale. Questo permette di ridurre la massa complessiva e quindi i costi, e di semplificare l'assemblaggio e il testing del satellite.

I moduli esterni possono essere di due tipi:

- Power Management

  Provvede a gestire l'alimentazione dell'elettronica di bordo e il controllo d'assetto. In esso sono incorporati i pannelli solari, la batteria ricaricabile, il caricatore della batteria, un microcontrollore che tiene traccia delle misure di tensione, corrente e temperatura del tile stesso, e il dispositivo di controllo attivo dell'assetto del satellite. Il pre-design modulare in questo caso consiste nella possibilità di posizionare queste tile come facce esterne del satellite in un appropriato numero, tale da soddisfare i requisiti sulla potenza della specifica missione.

- Telecommunication

  Provvede a gestire il protocollo usato per la comunicazione dei comandi e dei dati da e per la Terra. È costituita da due sistemi di trasmissione a banda stretta, completamente indipendenti e ridondanti al fine di aumentare l'affidabilità. Il primo è basato sul transceiver TI-CC1020 controllato da un microprocessore esterno TI-MSP430. Questo sistema che permette la comunicazione a 9.6 kbps nella banda a 437MHz ed è pensato per essere GENSO-compatibile. È quello oggetto di approfondimento in questa tesi. Il secondo sistema è basato sul transceiver TI-CC2510 con microprocessore integrato della famiglia 8051, che permette la comunicazione a 500 kbps nella banda a 2.4 GHz. Ciascuno dei due sistemi è completato del rispettivo amplificatore di potenza in trasmissione, del LNA in ricezione e della propria antenna, rispettivamente una a doppia elica e una Planar Inverter-f Antenna (PIFA). Durante l'esercizio del satellite, la faccia occupata da questa tile deve chiaramente risultare orientata verso Terra.

La parte interna del satellite è tipicamente lasciata vuota (ad eccezione dell'On-Board Computer e dei tile di supporto al payload), in modo da essere riempita dal payload definito dal committente, che è la sola parte che deve essere progettata e fabbricata ad-hoc per ogni missione. Questo conferisce al satellite AraMiS la caratteristica di essere

una piattaforma configurabile. Il sottosistema “On-Board Processor and Payload Support” provvede all’interfacciamento con i moduli Payload definiti dall’utilizzatore, all’esecuzione degli algoritmi di gestione dell’assetto, alla gestione di tutti i dati di housekeeping, ad alcune funzionalità del protocollo e all’interpretazione ed esecuzione dei comandi. Questa tile dispone anche di alcune CPU e memorie che, pur non essendo state sviluppate a tal fine, permettono di eseguire alcune funzionalità definite dall’utilizzatore, risparmiando l’utilizzo di ulteriori dispositivi dedicati.

Capitolo 2

Progetto GENSO

Caratteristica dei satelliti in orbita LEO è di essere visibili da una stessa ground station per un periodo molto limitato, cosicché la generica università è in grado di comunicare col proprio satellite in orbita LEO per un tempo generalmente non superiore ai 15 minuti per passaggio (come mostrato in Appendice C). Nel caso di CubeSat, a questa limitazione si aggiunge quella relativa al basso bitrate che tipicamente caratterizza i canali di uplink e downlink di questo tipo di satelliti.

Il progetto GENSO parte dalle seguenti considerazioni:

1) ogni ground station universitaria, quando non utilizzata con il proprio satellite (circa il 97% del tempo), potrebbe essere messa a disposizione come stazione remota ad altre organizzazioni per consentire l'accesso al proprio satellite quando non lo hanno in vista.

2) esiste la possibilità di coinvolgere la comunità radioamatoriale, capillarmente diffusa su tutta la Terra.

Il presente capitolo illustra il funzionamento di base del progetto GENSO, tenendo conto che si tratta di un progetto ancora in via di sviluppo nel momento in cui si scrive.

2.1 Progetto GENSO

Nell'ottica di specificare un sistema comune che massimizzi i vantaggi della collaborazione di cui si è parlato sopra, nel 2006 è nato il progetto GENSO (*Global Educational Network for Satellite Operations*) [4] coordinato dall'*Education Office* dell'ESA (*European Space Agency*).

GENSO ha l'obiettivo di creare un network di stazioni di radioamatore e università posizionate in punti diversi della Terra, basato sulla rete Internet, il cui principale scopo è quello di estendere alle università l'accesso al proprio satellite scientifico anche

quando esso non è a loro direttamente visibile. Il coinvolgimento di stazioni remote riguarda primariamente la ricezione dei dati di telemetria, ma le università responsabili di missione hanno la possibilità di utilizzarne alcune di fiducia anche per il remote control, secondo il sistema descritto nel prossimo paragrafo.

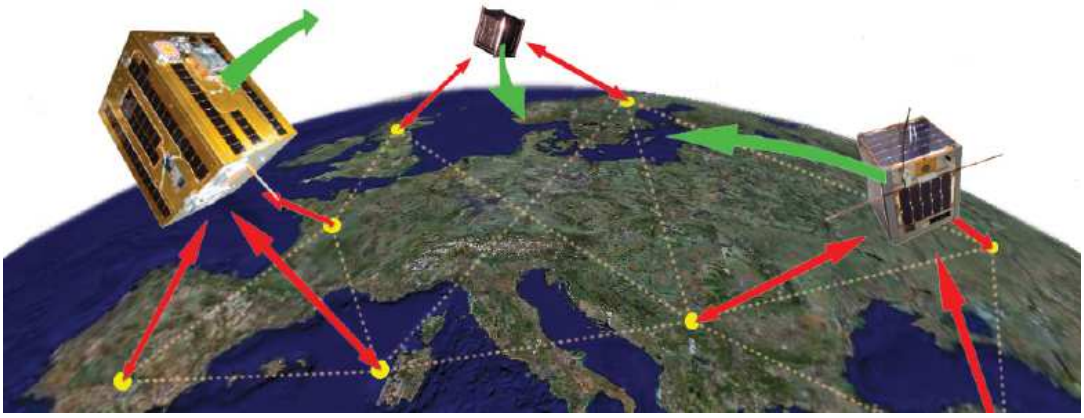


Figura 2.1. Rappresentazione del sistema GENSO

2.2 Architettura e funzionamento del sistema GENSO

Il sistema GENSO consiste di tre distinte componenti [5]:

1) Authentication Server (AUS)
2) Ground Station Server (GSS)
3) Mission Control Client (MCC)

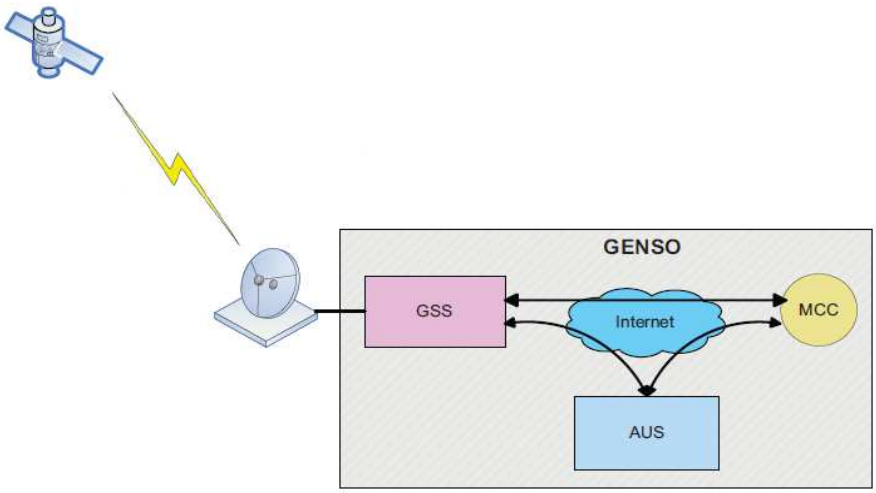


Figura 2.2. Componenti del sistema GENSO

2.2.1 Authentication Server (AUS)

Gestito dagli amministratori di GENSO, il ruolo dell'AUS è di intermediario tra la generica MCC che vuole usufruire dei servizi offerti dal sistema GENSO e un certo insieme di GSS. Durante la fase di sviluppo della rete GENSO si sono prese in considerazione diverse architetture, da quella fortemente centralizzata a quella peer-to-peer. Entrambe le architetture presentano vantaggi e svantaggi. Nel primo caso si considerava un Central Server che si occupasse quasi di tutto: calcolare l'istante dei passaggi, prenotare l'utilizzo delle ground station, memorizzare i dati per usi futuri, interpretare i dati ricevuti e renderli disponibili in maniera organizzata. Questa topologia risultava semplice da costruire, ma con diversi svantaggi che la rendevano inadatta per GENSO. Il principale era la mancanza di scalabilità. Un server centrale, infatti, può accettare solo un numero limitato di connessioni da altri computer. Tra gli obiettivi di GENSO, invece, vi è la possibilità di consentire la partecipazione ad un numero arbitrariamente grande di organizzazioni. In un futuro, questo potrebbe voler dire centinaia di satelliti e migliaia di ground station. Chiaramente un unico server centrale non può trattare la quantità di dati generata da migliaia di stazioni, e in ogni caso rappresenterebbe un punto critico in caso di problemi o manutenzione.

Nel secondo caso, l'autorità centrale ha un ruolo solo durante la fase di registrazione e accesso dei vari attori al sistema, mentre nell'esercizio del servizio questi lavorano in regime peer-to-peer. I vantaggi sono una limitata dipendenza da un server centrale. Gli svantaggi sono quelli derivanti dallo scarso controllo che può avere l'autorità centrale che gestisce GENSO su ciò che succede all'interno del sistema.

Si è quindi adottata una soluzione intermedia tra le precedenti, cioè disporre di alcuni server le cui funzioni sono di autenticare i nodi della rete, eseguire la crittografia, mantenere una lista aggiornata dei satelliti e delle ground station autorizzate e distribuire queste informazioni quando richieste rispettivamente dalle GSS e dalle MCC, cioè funzioni che non richiedono un intensivo utilizzo della banda disponibile.

2.2.2 Ground Station Server (GSS)

È l'applicazione in esecuzione in ogni stazione di radioamatore che intende offrire il proprio servizio al sistema GENSO. Permette al computer di prendere controllo del rotore d'antenna, del modem TNC e del transceiver per consentire alla Mission Control Station di comunicare col proprio satellite. Per interfacciarsi con l'hardware, il GSS utilizza appositi driver (Hamlib). Appena attivato, per mezzo del Controller si connette all'AUS sia per fornire le caratteristiche della stazione (frequenze e modulazioni operabili, posizione geografica, eccetera) ed essere aggiunto alla lista delle ground station disponibili (GSSL - Ground Station Server List) mantenuta dalla AUS e sia per ricevere la lista dettagliata dei satelliti GENSO (PSL – Participating Spacecraft List), come si può vedere nelle frecce 1 e 1.2 nel diagramma di Figura 2.3. Da quel momento in poi la GSS può essere prenotata da un MCC per collegare il proprio satellite (Scheduler).

2.2.3 Mission Control Client (MCC)

È l'applicazione in esecuzione al centro di controllo del satellite (Mission controller). Mediante una interfaccia grafica, permette all'operatore responsabile della missione di utilizzare i servizi GENSO. Quando attivato, in fase di autenticazione, fornisce all'AUS le caratteristiche del proprio satellite (frequenza, modulazione, eccetera) e riceve l'elenco aggiornato di tutte le ground station disponibili (frecce 2 e 2.2 nel diagramma di Figura 2.3). A questo punto, l'MCC può predire sopra quale ground station e in quale periodo passerà il proprio satellite, quindi può contattare il relativo GSS per prenotarne l'utilizzo (frecce da 3 a 3.1.1.1 nel diagramma di Figura 2.3). Sono previsti modi diversi di comunicare, noi consideriamo quello secondo cui durante un periodo prenotato, la comunicazione tra MCC e satellite avviene in maniera trasparente, bidirezionale e in real-time (frecce 4 e 4.1 nel diagramma di Figura 2.3).

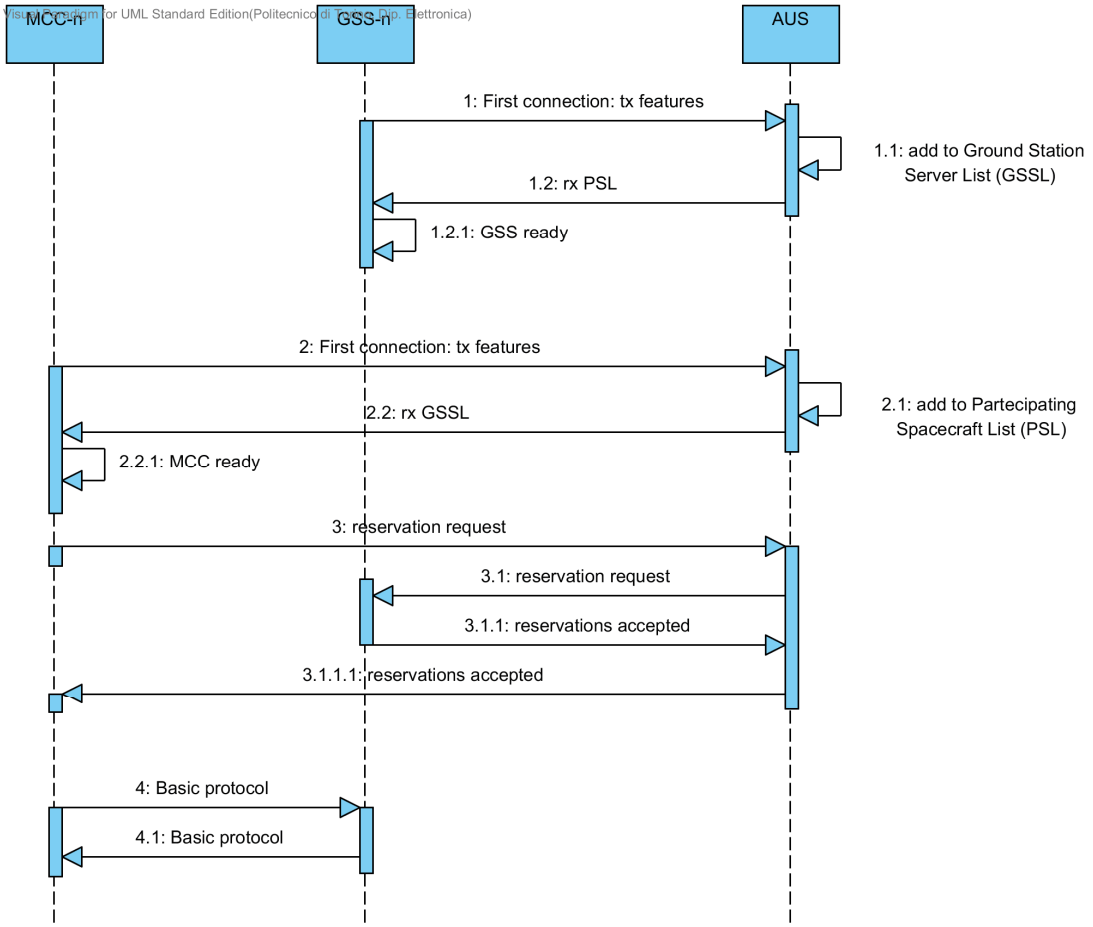


Figura 2.3. Diagramma di sequenza del funzionamento del sistema GENSO

Capitolo 3

Protocolli radioamatoriali

Nell'ottica di rendere il nanosatellite AraMiS compatibile al sistema GENSO, si vuole far si che il nostro sottosistema di comunicazione possa interagire col maggior numero possibile di stazioni di radioamatore sparse nelle diverse regioni della Terra. Quindi nel sviluppare il protocollo di comunicazione di AraMiS è necessario identificare le modalità di comunicazione satellitari attualmente più diffuse tra i radioamatori, scopo della ricerca riportata in questo capitolo.

3.1 Introduzione

Come spesso accade nei processi di innovazione tecnologica, anche nell'ambito delle comunicazioni satellitari amatoriali il quadro attuale è il risultato dell'affermarsi di un disegno dominante. A volte singoli episodi, una serie di coincidente, ragioni di ordine pratico, o delle scelte giuste innescano dei circoli virtuosi che tendono a rafforzare delle modalità piuttosto che altre, indipendentemente dal suo grado di superiorità o inferiorità rispetto alle modalità alternative. Così è stato, ad esempio, nel processo di affermazione della AFSK che si è affermata grazie alla possibilità di realizzarla semplicemente collegando un modem all'ingresso e all'uscita dello stadio audio del generico transceiver commerciale, già presente nella maggior parte delle stazioni di radioamatore. Per comprendere le ragioni e le caratteristiche dei sistemi utilizzati nelle comunicazioni satellitari radioamatoriali è quindi conveniente farne una analisi lungo la linea dell'evoluzione tecnologica. Il paragrafo che segue contiene una sintesi di tale analisi, limitatamente alle modalità di trasmissione, prendendo in considerazione in ordine cronologico alcune tra le missioni maggiormente significative per aver contribuito a definire il quadro attuale.

Si arriverà alla fine del capitolo ad una identificazione di una modalità di comunicazione da prendere come riferimento e la cui compatibilità costituirà uno dei requisiti del progetto del protocollo AraMiS.

3.2 Definizioni preliminari

Segnale di beacon

In questo contesto, con il termine beacon si intende un segnale che nella forma più elementare viene trasmesso automaticamente a specifici intervalli di tempo con lo scopo di segnalare la presenza del satellite nel canale. Il segnale di beacon deve rendere identificabile il satellite da cui è trasmesso (requisito di legge). La maggior parte delle volte il segnale di beacon trasporta anche dati di telemetria. Per i satelliti in orbita elevata che furono lanciati dai radioamatori nel periodo noto come "fase 3" si usarono due categorie di beacon, denominate come [6]:

- engineering beacon: un beacon trasmesso a potenza relativamente elevata, usato tipicamente in elevata altitudine con una antenna omnidirezionale durante le prime fasi di trasferimento in orbita o in caso di emergenza;
- general beacon: trasmesso continuativamente ad intervalli di tempo regolari.

Nomenclatura di uso comune per le modalità di funzionamento dei transponder

Per i transponder (ripetitori nei quali il segnale in ingresso viene trasposto in uscita su un'altra banda) sono state tradizionalmente definite le seguenti modalità di funzionamento in base alle frequenze di uplink e downlink:

Modo A	uplink a 145 MHz e downlink a 29 MHz
Modo B	uplink a 435 MHz e downlink a 145 MHz
Modo J	uplink a 145 MHz e downlink a 435 MHz
Modo K	uplink a 21 MHz e downlink a 29 MHz
Mode L	uplink a 1269 MHz e downlink a 435 MHz
Modo S	uplink a 435 MHz (o 1260 MHz) e downlink a 2.4 GHz

Tabella 3.1. Definizioni dei modi di funzionamento dei transponder

3.3 Satelliti radioamatoriali

Nella descrizione che segue per ogni missione citata utilizziamo il formato:

Beacon (Telemetria)

Telecomunicazioni

Telecomando

1962 OSCAR 1 e OSCAR 2

Beacon (Telemetria): il segnale di beacon integrava la telemetria. Il metodo di codifica con cui esso trasportava la telemetria consisteva nella ripetizione del messaggio "HI" in codice Morse, la cui velocità era controllata da un sensore interno di temperatura.

Telecomunicazioni: a parte il downlink del beacon, il satellite non aveva capacità di comunicazione con le ground station.

Telecomando: il satellite non aveva capacità di telecomando.

1962 OSCAR III

Beacon (Telemetria): aveva due segnali di beacon. Uno consisteva di una semplice portante non modulata utile per il tracking (determinazione in tempo reale della velocità e della posizione del satellite) e studi sulla propagazione, mentre l'altro integrava la telemetria. Questa trasportava informazioni in formato analogico sullo stato della batteria primaria (temperatura e tensione) e sul dispositivo amplificatore (temperatura).

Telecomunicazioni: è il primo satellite amatoriale a fornire un servizio di collegamento tra radioamatori. Esso era infatti equipaggiato di un transponder lineare con larghezza di banda di 50 kHz non invertente (riceveva a 146 MHz e ritrasmetteva a 144 MHz).

Telecomando: il satellite non aveva capacità di telecomando.

1970 Australis-OSCAR 5

Beacon (Telemetria): il segnale di beacon integrava la telemetria e veniva trasmesso su due bande, 29 MHz e 144 MHz (prima telemetria ridondata).

Telecomunicazioni: non includeva funzioni di comunicazione (diventerà una caratteristica abbastanza comune per i satelliti di origine universitaria in cui la missione primaria era quella di condurre esperimenti).
Telecomando: è il primo satellite amatoriale che includeva funzioni di telecomando. Consisteva nella semplice capacità di spegnere e riaccendere il beacon.

1972 AMSAT-OSCAR 6

Beacon (Telemetria): il segnale di beacon integrava la telemetria e veniva trasmesso su due bande 29 MHz e 435 MHz. Si basava su un particolare sistema di elaborazione che convertiva informazioni analogiche quali tensione, corrente e temperatura come numeri trasmessi in codice Morse su modulazione CW. Tale modalità anche all'epoca non era considerata chiaramente la soluzione più efficiente, ma fu scelta perché permetteva a qualunque stazione senza particolari equipaggiamenti (nessun modem), ma solo con un operatore che conoscesse la telegrafia, di decodificare i dati.
Telecomunicazioni: è stato equipaggiato con un transponder lineare non invertente a elevatissima sensibilità, con larghezza di banda di 100 kHz e modo A. Inoltre introduceva per la prima volta un primitivo sistema di store-and-forward di messaggi, chiamato *Codestore* che permetteva a una stazione opportunamente equipaggiata di caricare messaggi teletype fino a 256 caratteri in codice Morse, i quali venivano successivamente ritrasmessi verso Terra in broadcast.
Telecomando: il sistema di telecomando si basava su decoder ridondati.

1974 AMSAT-OSCAR 7

Beacon (Telemetria): il segnale di beacon integrava la telemetria e veniva trasmesso su quattro bande: 29 MHz, 145 MHz, 435 MHz e 2304 MHz. La telemetria era trasmessa in due modi. Uno con codifica Morse (come in AO-6), capace di trasmettere informazioni su 24 parametri analogici. Il secondo, sperimentale, si basava su un sistema RTTY a modulazione AFSK con shift 850 Hz, capace di trasmettere a una velocità di 60 wpm informazioni relative a 60 parametri analogiche e 20 parametri di stato.

Telecomunicazioni: è stato equipaggiato di due transponder lineari che operavano in maniera mutuamente esclusiva (temporizzata o comandata da Terra), uno non invertente con modo A (come quello di AO-6) e l'altro invertente con modo B, entrambi per modulazioni analogiche CW e SSB. Osserviamo che due dei quattro beacon sono posizionati alla estremità superiore delle bande di trasmissione dei due transponder. Era inoltre equipaggiato con lo stesso sistema *Codestore* di AO-6.

Telecomando: derivato da quello collaudato con successo su AO-6, il sistema di telecomando si basava su decoder ridondati (29 MHz e 145 MHz). Il satellite era progettato per operare in uno dei quattro modi mostrati in Figura 3.1. Il decoder era in grado di decodificare 35 comandi (Figura 3.2).

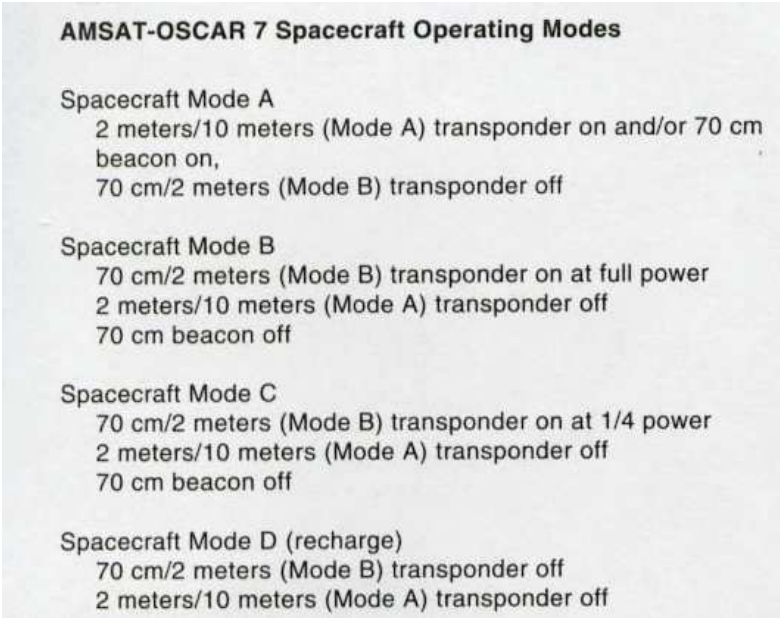


Figura 3.1. Modi di funzionamento di AMSAT-OSCAR 7

AMSAT-OSCAR 7 Command States

Command #	*Action*
1	Select Spacecraft Mode A
2	Select Spacecraft Mode B
3	70 cm Beacon on (Spacecraft Mode A or D only)
4	70 cm Beacon off
5	Run Codestore
6	Load Codestore
7	Morse code TTY (20 WPM)
8	Morse code TTY (10 WPM)
9	2/10 transponder – full sensitivity
10	2/10 transponder – reduced sensitivity (-14dB)
11	Route Codestore to 70 cm beacon
12	Route Morse code TTY to 70 cm beacon
13	Route RTTY to 70 cm beacon
14	Route Morse code TTY to 10 meter and 2 meter beacons
15	Route Codestore to 10 meter and 2 meter beacons
16	Route RTTY to 10 meter and 2 meter beacons
17	Reset 24 clock
18	Select Spacecraft Mode C
19	Select Spacecraft Mode D
20	Selcct BCR 1
21	Select BCR 2
22	Set RTTY to dwell mode
23	Set RTTY to run mode
24	2304 MHz beacon on (14 minutes only)
25	2304 MHz beacon off
26	Set 2304 MHz beacon to internal keying
27	Route Morse code TTY to 2304 MHz beacon
28	Set RTTY to FSK mode
29	Set RTTY to AFSK mode
30	Same as 1
31	Same as 2
32	Same as 3
33	Same as 4
34	Same as 18
35	Same as 19

Figura 3.2. Comandi di AMSAT-OSCAR 7

1983 AMSAT-OSCAR 10 (Phase 3B)

Beacon (Telemetria): primo satellite di origine radioamatoriale ad adottare le trasmissioni digitali computer-to-computer, almeno per quanto riguarda la telemetria. Utilizzava infatti la codifica ASCII in modulazione BPSK a 400 bps in banda 145 MHz. Telecomunicazioni: era equipaggiato di due transponder lineari invertenti, il primo di modo B e il secondo di modo L.

1984 UoSat OSCAR 11 (UoSAT 2)

Beacon (Telemetria): i beacon erano tre e trasportavano i dati di telemetria con codifica ASCII: 145MHz e 435 MHz in AFSK 1200 bps (compatibile con lo standard *Bell 202*) e 2401 MHz in PSK 4800 bps. Trasmetteva informazioni su 60 parametri analogici e 96 parametri di stato.

Telecomunicazioni: non includeva funzioni di comunicazione ad accesso aperto, ma includeva cosiddetti Digital Communication Experiment (DCE), autorizzando ad un numero limitato di ground station l'accesso ad un transponder digitale AFSK a 1200 bps, secondo un meccanismo di store-and-forward, precursore degli attuali open access packet sistems.

Telecomando: il sistema di telecomando si basava su tre decoder ridondati, operanti nelle bande amatoriali 145 MHz, 435 MHz e 1.2 GHz.

1986 Fuji-OSCAR 12 (JAS 1)

Beacon (Telemetria): il segnale di beacon integrava la telemetria e veniva trasmesso sulla bande 435 MHz nei modi SSB e CW.

Telecomunicazioni: è stato equipaggiato di un transponder lineare invertente di modo J per le modulazioni SSB e CW. È stato inoltre il primo satellite ad offrire un servizio ad accesso aperto per comunicazioni digitali tra radioamatori secondo il concetto PACSAT. PACSAT si applica a satelliti in orbita LEO per consentire comunicazioni non real-time, ma store-and-forward tra stazioni diverse che vedono il satellite in tempi diversi. Utilizzava i canali uplink nella banda a 145 MHz con modulazione MFSK a 1200 bps e downlink nella banda a 435 MHz con modulazione PSK a 1200 bps. Questo link era incompatibile sia con i modem usati nelle comunicazioni terrestri, sia con il Bell 202 e necessitava quindi di modem progettati ad-hoc. Il modem più diffuso era quello progettato da G3RUH, che necessitava un accesso diretto allo stadio (de)modulatore del transceiver, bypassando lo stadio audio.

1988 AMSAT-OSCAR 13 (Phase 3C)

Beacon (Telemetria): i beacon erano quattro e integravano la telemetria con codifica RTTY e ASCII in BPSK 400 bps: due erano nella banda 145MHz (general beacon e engineering beacon), uno nella banda a 435 MHz (general beacon) e infine uno nella banda a 2.4 GHz (engineering beacon).

Telecomunicazioni: era equipaggiato con due transponder lineari, uno di modo B e l'altro di modo S per le modulazioni SSB e CW. Inoltre era dotato di un particolare transponder digitale (digipeater), chiamato *Rudak-1*, un sistema store-and-forward di messaggi con funzione di mailbox, con uplink a a 435 MHz 2400 bps e downlink a 146 MHz 400 bps (intenzionalmente la stessa velocità della telemetria, per permetterne la ricezione senza ulteriori modifiche a chi era già equipaggiato a ricevere la telemetria standard *phase 3*, in vigore dalla fine degli anni '70).

1990 AMSAT OSCAR 16 (PacSAT)

Beacon (Telemetria): con l'introduzione della classe MicroSat, di cui AO-16 ne è il primo esempio, viene superato il concetto di trasmissione della telemetria a frame fisso a favore di quella a pacchetto. La trasmissione a pacchetto esiste sia in modalità connectionless, e in tal caso la ground station riceve una telemetria basica, oppure in modalità connected, e in tal caso la ground station potrà inviare richieste di informazioni di telemetria addizionali.

Telecomunicazioni: AO-16 inaugura il *PACSAT Protocol Suite* per la classe Microsat, che include il protocollo AX.25 e che definisce un modo per il trasferimento di file store-and-forward attraverso microsatelliti in orbita LEO che si è molto diffuso nei satelliti lanciati gli anni successivi. Il servizio svolto da questo protocollo si chiama *PacSAT BBS* (Bulletin Board System) e permette ad una stazione di Terra di caricare sulla memoria del satellite dei bollettini indirizzati a specifiche stazioni o a tutte (broadcast). Le stazioni destinatarie trasmettono quindi richieste di ricezione di bollettini ad esse indirizzati. Quando il satellite è in vista da parte delle stazioni richiedenti, raccoglie i comandi di richiesta e se ha dei bollettini per esse, li trasmette e quindi li rimuove dalla memoria. Per quanto riguarda l'*AO-16* questo servizio si

poggiava su un singolo canale di downlink con modulazione BPSK a 1200 bps e quattro canali di uplink con modulazione MFSK a 1200 bpps.

1990 Fuji-OSCAR 20

Beacon (Telemetria): ha due beacon che integrano la telemetria, il primo in CW a 20 wpm e il secondo in BPSK a 1200 bps. Entrambi sulla banda a 435 MHz. La telemetria in CW trasporta informazioni relative a 12 misure analogiche e 33 stati. Per quanto riguarda la telemetria in BPSK, questa è trasmessa in pacchetti secondo il protocollo AX.25. Sono usati tre tipi di frame: real time telemetry (ASCII o binario), stored telemetry (ASCII o binario), messages (numerati da 0 a 9).

I frame delle prime due categorie contengono informazioni sulle misure analogiche raccolte in 29 punti del satellite (tensioni, correnti e temperature) e sullo stato di 33 parametri. Le informazioni sono rappresentate in formato binario o più frequentemente in formato ASCII.

Telecomunicazioni: era equipaggiato con un transponder lineare invertente di modo J con larghezza di banda 100 KHz. Inoltre inlcudeva un transponder digitale per il servizio PacSAT BBS con 4 uplink in banda 145 MHz in modulazione Mancherster 1200 bps e un downlink in banda 435 MHz in modulazione BPSK 1200 bps. Differentemente dai Microsat come AO-16, permetteva di accedere al servizio senza necessitare di uno speciale Microsat terminal software. Ogni computer o terminale che poteva essere usato per accedere al servizio Bulletin Board Systems (BBSs) terrestre era idoneo per accedere al servizio di mailbox offerto da FO-20.

Telecomando: il sistema di telecomando riconosce 5 modi operativi ed è caratterizzato da un proprio sottoinsieme di sottosistemi attivi in ordine di consumo crescente:

Modo D: tutti i carichi sono spenti eccetto il ricevitore dei comandi (2.5W)

Modo JA (5.3W)

Modo JD[1], … , Modo JD[4]: banchi di memoria da 256 KB da 1 a 4 attivati (da 6.2 W a 7.3 W)

Modo DI: tutti i carichi spenti eccetto il ricevitore dei comandi, la CPU e le memorie

Modo JAD: equivale al modo JA e JD congiunto (10.2 W)

1996 Fuji-OSCAR 29

Beacon (Telemetria): come per FO-20.

Telecomunicazioni: come per FO-20, ma in più è equipaggiato di un secondo transponder digitale per il servizio PacSAT BBS con un canale di uplink in banda 145 MHz downlink in banda 435 MHz compatibile con il packet radio a 9600 bps.

Telecomando: vale quanto detto per FO-20. La Tabella 3.2 riassume i comandi utente per il servizio PacSAT BBS.

FO-29 Digital Transponder User Commands and TNC Settings

Command	*Description*
A	User's command
B	Bulletin; lists file headers addressed to all.
F	Lists the latest file header and those in reverse order of time.
FDD <MM/DD>	Lists file headers in reverse order of time from MM/DD.
H	Help, shows list of commands and their explanation.
K <nnn>	Deletes file, possible by posted and addressed station only.
M	Lists file headers addressed to user.
R <nnn>	Shows message of file.
U	Lists current users.
W	Writes messages, sending is requested address and title of message, after that messages are sent, and termination of message should be pressed 14, return at the line head.
Y	Requests more display for B and F commands.
Q	Requests disconnection of current link.

FO-29 TNC Settings

Item	*Setting*
FULLDUP	ON
MAXFRAMES	4
PACLEN	128 or less

If there is a *SOFTDCD* command, turn it OFF.

Tabella 3.2. Comandi utente per il servizio PacSAT BBS.

1998 Thai – Microsatellite – OSCAR 31

Beacon (Telemetria): ha quattro beacon, tutti in banda 436 MHz in modulazione FSK e 9600 bps e 38400 bps. Questa seconda velocità era stata utilizzata per trasmettere verso Terra immagini acquisite a bordo (missione primaria).

I dati di telemetria consistevano in 70 misure analogiche e 16 misure di stato. Per decodificarli le ground station dovevano utilizzare un programma chiamato DTLM.

Telecomunicazioni: la missione secondaria forniva ai radioamatori il servizio Pacsat con uplink in banda a 145 e downlonk in banda 436 MHz, protocollo AX.25, velocità 9600 bps.

1999 UoSAT-OSCAR 36 (uosat 12)

Beacon (Telemetria): ha due beacon. Il primo sulla banda 437 MHz con modulazione FSK a 9600 bps, 38400 bps (primario) e 76800 bps (sperimentale). Il secondo sulla banda a 2,401 GHz con modulazione BPSK a 1Mbps con codifica di Viterbi

Telecomunicazioni: è equipaggiato con due transponder digitali. Il primo con funzioni di store-and-forward con uplink in banda 145 MHz e velocità 9600 bps e downlink in banda 437 MHz con velocità come per il beacon nella stessa banda. Il secondo transponder, sviluppato dalla Nanyang Technical University (Singapore) e dalla SSTL MErlion Transponder, con uplink in banda 1.2 GHz e downlink in 2.4 GHz, supporta trasmissioni analogiche e digitali fino a 1Mbps, utile per il downlink di immagini e altri file di grandi dimensioni. Inoltre include la possibilità di trasmettere a basso rate in spread spectrum.

2000 AMSAT-OSCAR 40 (Phase 3D)

Beacon (Telemetria): ha diversi beacon, nelle bande a 145, 435 e 2400 MHz. La telemetria è integrata nei beacon, tipicamente modulate in BPSK a 400 bps, e via RUDAK-U. Il satellite è progettato per consentire un un'ampia varietà, flessibilità e ridondanza nel sistema di telemetria.

Telecomunicazioni: anche il sistema di comunicazione offre molta varietà di utilizzo. Il satellite è equipaggiato di un set di transponder che possono essere collegati assieme

attraverso una configurazione dinamica in modo da poter utilizzare più uplink e più downlink secondo diverse combinazioni. È equipaggiato inoltre di un complesso transponder digitale, chiamato RUDAK-U.

3.4 Satelliti radioamatoriali universitari

Negli ultimi anni sono stati sviluppati decine di satelliti universitari nello standard CubeSat basati su bande e modi di trasmissione radioamatoriali. La Tabella 3.3 riassume le caratteristiche principali di alcuni di essi. Si osserva come, nonostante tutte le trasmissioni avvengono non codificate, le organizzazioni oltre il tipo di frame usato, tendono a non divulgare eventuali protocolli studiati ad-hoc per i comandi.

Satellite	Size	Radio	Frequency	License	Power	TNC	Protocol	Baud Rate/Modulation	Lifetime	Antenna
AAU1 CubeSat	1U	Wood & Douglas SX450	437.475 MHz	amateur	500 mW	MX909	AX.25, Mobitex	9600 baud GMSK	3 months	dipole
DTUsat-1	1U	RPMD RF2905	437.475 MHz	amateur	400 mW		AX.25	2400 baud FSK	0 days	cauted turnstile
CanX-1	1U	Melexis	437.880 MHz	amateur	500 mW		Custom	1200 baud MSK	0 days	crossed dipoles
Cute-1 (CO-55)	1U	Maki Denki (Beacon) Alinco DJ-C4 (Data)	436.8375 MHz 437.470 MHz	amateur amateur	100 mW 350 mW	PIC16LC73A MX614	CW AX.25	50 WPM 1200 baud AFSK	65+ months	monopole monopole
QuakeSat-1	3U	Tekk KS-960	436.675 MHz	amateur	2 W	BayPac BP-96A	AX.25[2]	9600 baud FSK	7 months	turnstile
XI-IV (CO-57)	1U	Nishi RF Lab (Beacon) Nishi RF Lab (Data)	436.8475 MHz 437.490 MHz	amateur amateur	80 mW 1 W	PIC16C716 PIC16C622	CW AX.25	50 WPM 1200 baud AFSK	65+ months	dipole dipole
XI-V (CO-58)	1U	Nishi RF Lab (Beacon) Nishi RF Lab (Data)	437.465 MHz 437.345 MHz	amateur amateur	80 mW 1 W	PIC16C716 PIC16C622	CW AX.25	50 WPM 1200 baud AFSK	36+ months	dipole dipole
NCube-2	1U		437.505 MHz	amateur			AX.25	1200 baud AFSK	0 days	monopole
UWE-1	1U	PR430	437.505 MHz	amateur	1 W	H8S/2674R[4]	AX.25	1200/9600 baud AFSK	0.75 months	end-fed dipole
Cute-1.7+APD (CO-56)	2U	Telemetry Beacon Alinco DJ-C5	437.385 MHz 437.505 MHz	amateur amateur	100 mW 300 mW	H8S/2328[4] CMX589A	CW AX.25/SRLL	50 WPM 1200 AFSK/9600 GMSK	2.5 months	dipole dipole
GeneSat-1	3U+	Atmel ATA8402 (Beacon) Microhard MHX-2400	437.067 MHz 2.4 GHz	amateur ISM	500 mW 1 W	PIC12C617 Integrated[5]	AX.25 Proprietary	1200 baud AFSK	3 months	monopole patch
CSTD1	1U	Commercial[6]	400.0375 MHz	Experimental	<1 W	PIC	Proprietary	1200 baud AFSK	19+ months	dipole
AeroCube-2	1U	Commercial[6]	902-928 MHz	ISM	2 W	Integrated[5]	Proprietary	38.4 kbaud	0.25 months	patch
CP4	1U	TI CC1000	437.325 MHz	amateur	1 W	PIC18LF6720	AX.25	1200 baud FSK	2 months	dipole
Libertad-1	1U	Stensat	437.405 MHz	amateur	400 mW		AX.25	1200 baud AFSK	1 month	monopole
CAPE1	1U	TI CC1020	435.245 MHz	amateur	1 W	PIC16LF452	AX.25	9600 baud FSK	4 months	dipole
CP3	1U	TI CC1000	436.845 MHz	Experimental	1 W	PIC18LF6720	AX.25	1200 baud FSK	19+ months	dipole
MAST[10]	3U	Microhard MHX-2400	2.4 GHz	ISM	1 W	Intgrated[5]	Proprietary	15 kbps	0.75 months	monopole
Delfi-C3 (DO-64)	3U	Custom Beacon Custom Transponder	145.870 MHz 145.9-435.55 MHz	amateur amateur	400 mW 200 mW	PIC18LF4680 N/A	AX.25 Linear	1200 baud BPSK 40 kHz wide	7+ months	turnstile turnstile
Seeds-2 (CO-66)	1U	Musashino Electric (Beacon) Musashino Electric (Data)	437.485 MHz 437.485 MHz	amateur amateur	90 mW 450 mW		CW AX.25	1200 baud AFSK	7+ months	monopole monopole
CanX-2	3U	Custom S-Band	2.2 GHz	Space Research[12]	500 mW	Integrated	NSP	16kbps-256kbps BPSK	7+ months	patch
AAUSAT-II	1U	Holger Eckhardt (DF2FQ)	437.425 MHz	amateur	610 mW	PIC18LF6680	AX.25	1200 baud MSK	7+ months	dipole
Cute 1.7+APD II (CO-65)	3U+[14]	Invax (Beacon) Alinco DJ-C5 (Data)	437.275 MHz 437.475 MHz	amateur amateur	100 mW 300 mW	H8S/2328 H8S/2328, CMX589A	CW AX.25/SRLL	50 WPM 1200 AFSK/9600 GMSK	7+ months	monopole monopole
Compass-1	1U	BC549 (Beacon) Holger Eckhardt (Data)	437.275 MHz 437.405 MHz	amateur amateur	200 mW 300 mW	PIC12F629 C8051F123, FX614	CW AX.25	15 WPM 1200 baud AFSK/MSK	7+ months	dipole dipole

Tabella 3.3. Principali satelliti universitari

3.5 Contesto odierno

Dall'analisi svolta emergono diversi momenti importanti, ma probabilmente quello fondamentale nel contribuire a definire il contesto odierno delle comunicazioni satellitari amatoriali lo si ha a cavallo del 1990 e riguarda i satelliti in orbita LEO.

Nel 1986 appare su FO-12 il primo servizio BBS open access secondo il concetto PACSAT, col quale si intende un servizio per satelliti LEO basato sulla memorizzazione di dati a bordo da parte di una stazione e sul successivo recupero da parte di un'altra stazione. Questa modalità consente a due stazioni che non sono contemporaneamente in vista del satellite LEO di poter comunicare. Quella su FO-12 è però una realizzazione rudimentale e inefficiente, dove l'accesso al satellite e il trasferimento dei dati sono gestiti manualmente.

Nel 1990, in seguito alla *9th ARRL Computer Networking Conference* viene pubblicato lo *UoSat/MicroSat File Transfer Protocol Suite*, più conosciuto come *PACSAT suite*. L'obiettivo di questa suite di protocolli è quella di sfruttare al meglio le potenzialità di un satellite radioamatoriale in orbita LEO con una larghezza di banda limitata, tenendo conto di un bacino di utenza eterogeneo e mondiale. Facendo riferimento al 2° livello del modello OSI, PACSAT suite utilizza il protocollo AX.25. Nei livelli superiori includeva una serie di funzioni che ne fanno un vero e proprio file server ottimizzato per la specifica situazione. Per operare con i satelliti che avevano adottato PACSAT Suite, le stazioni di Terra utilizzavano due software: *PG* per la trasmissione e *PB* per la ricezione.

Durante i primi anni '90 tutti satelliti che furono lanciati usavano *PACSAT suite* in modulazione BPSK a 1200 bps e in modulazione FSK a 9600 bps e nel 1992 *PG* e *PB* vennero rimpiazzati da *WiSP* che li integrava entrambi, oltre ad aggiungere diverse altre funzionalità. Il 1992 può essere considerato l'anno in cui le comunicazioni satellitari amatoriali passano da una fase sperimentale a una fase operativa. Un gran numero di radioamatori iniziano ad equipaggiarsi per poter operare nelle modalità sopra descritte. In questi anni si ha quindi un processo di consolidamento di queste tecnologie. Nel 1996 un sondaggio rivelava che del totale dei radioamatori attivi in comunicazioni satellitari, il 25% era equipaggiato per comunicazioni digitali e di questo il 18% era operativo in

FSK a 9600 bps e il 7% in BPSK a 1200 bps [6]. Il protocollo adottato risulta essere l'AX.25, lo stesso usato per il packet radioamatoriale terrestre. Di conseguenza anche la tipologia di dispositivo è la stessa di quella usata per il packet radioamatoriale terrestre. Tali dispositivi prendono il nome di TNC (*Terminal Node Controller*), costituiscono l'interfaccia tra PC e transceiver e realizzano sia il protocollo che le funzioni di modem (Figura 3.3). Il PC potrà comunicare con il TNC attraverso la porta seriale RS-232 usando un emulatore di terminale quale RealTerm o Yat32 oppure software con maggiori funzionalità. Il TCN potrà interfacciarsi al transceiver attraverso l'uscita e l'ingresso analogico del segnale audio vocale. Nei transceiver più moderni sono inoltre predisposti appositi ingressi che permettono la (de)modulazione del segnale a bassa frequenza bypassando i filtri dedicati ai segnali vocali (Figura 3.4). Differentemente dalle trasmissione a 1200 bps, questo è indispensabile quando si vuole operare a 9600 bps.

Figura 3.3. Transceiver radioamatoriale collegato a un TNC

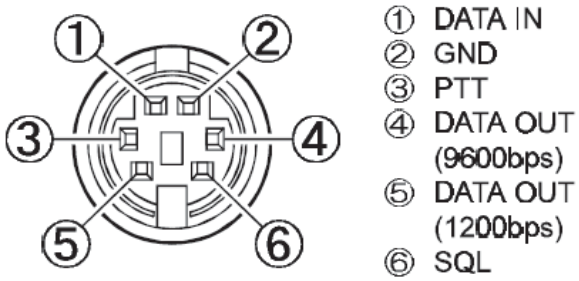


Figura 3.4. Ingresso del transceiver dedicato alla connessione con un TNC

In conclusione, dall'analisi svolta si identifica come protocollo radioamatoriale a cui rendersi compatibili, l'AX-25 con schema di modulazione FSK e data-rate 9600 bps nella banda 437 MHz. Per questo motivo il capitolo 5 sarà dedicato alla descrizione del protocollo AX.25.

Capitolo 4

Modulo di comunicazione a 437 MHz del satellite

Questo capitolo discute il modulo di comunicazione a 437 MHz del satellite, con particolare riferimento al prototipo presente in laboratorio e usato per realizzare il livello fisico e data-link del protocollo AX.25. Successivamente saranno presentati i nuovi moduli (437 MHz e 2.4 GHz) in via di sviluppo. Il capitolo si chiude con il calcolo del link-budget per una missione tipo.

4.1 Componenti utilizzati (COTS)

4.1.1 Transceiver: TI CC1020

Dalla analisi conoscitiva del contesto radioamatoriale precedentemente sviluppata, per ciò che riguarda la scelta del transceiver sono stati fissati i seguenti requisiti:

- banda utilizzata: radioamatoriale a 437 MHz
- schema di modulazione: FSK
- data-rate: 9600 bps.

D'altra parte, considerazioni energetiche relative al sistema in cui deve lavorare il modulo di comunicazione hanno indirizzato la ricerca verso un componente caratterizzato da:

- minima tensione di alimentazione
- minimo consumo di corrente
- massima potenza in uscita
- possibilità di interfacciamento con PA esterno.

Durante la fase di ricerca, eseguita da un precedente tesista, sono stati confrontati tra loro diversi componenti COTS. L'idea iniziale era quella di utilizzare un dispositivo che al suo interno integrasse anche il microcontrollore, ma con questa caratteristica non ne sono stati trovati per la frequenza desiderata. Si è quindi optato per il TI CC1020 [10], un dispositivo pensato per scopi ISM (Industrial, Scientific and Medical) e SRD (Short Range Device) che richiede di essere gestito da un microprocessore esterno attraverso un'interfaccia che discuteremo di seguito. Il CC1020 è un chip che racchiude tutte le funzionalità di un transceiver a banda stretta (12.5 o 25 kHz) su due bande UHF (402÷470 MHz e 804÷960 Mhz). Le modulazioni numeriche supportate sono la OOK, la FSK e la GFSK con data-rate fino a 153.6 kbps. Il CC1020 si presta bene ai nostri scopi per le sue buone prestazioni (sensitività: -118dBm per un canale a larghezza di banda 12.5kHz), perché è piccolo (32 pin), perché necessita di pochi altri componenti esterni, e soprattutto perché, rispetto a dispositivi simili, funziona con tensione di alimentazione bassa (2.3÷3.6 V) e assorbe poca corrente (19.9 mA in RX).

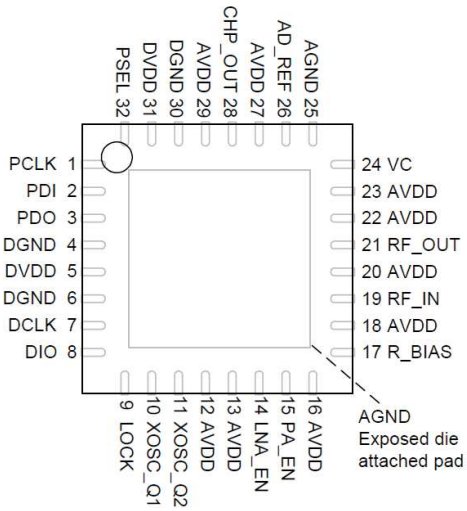


Figura 4.1. Package del CC1020

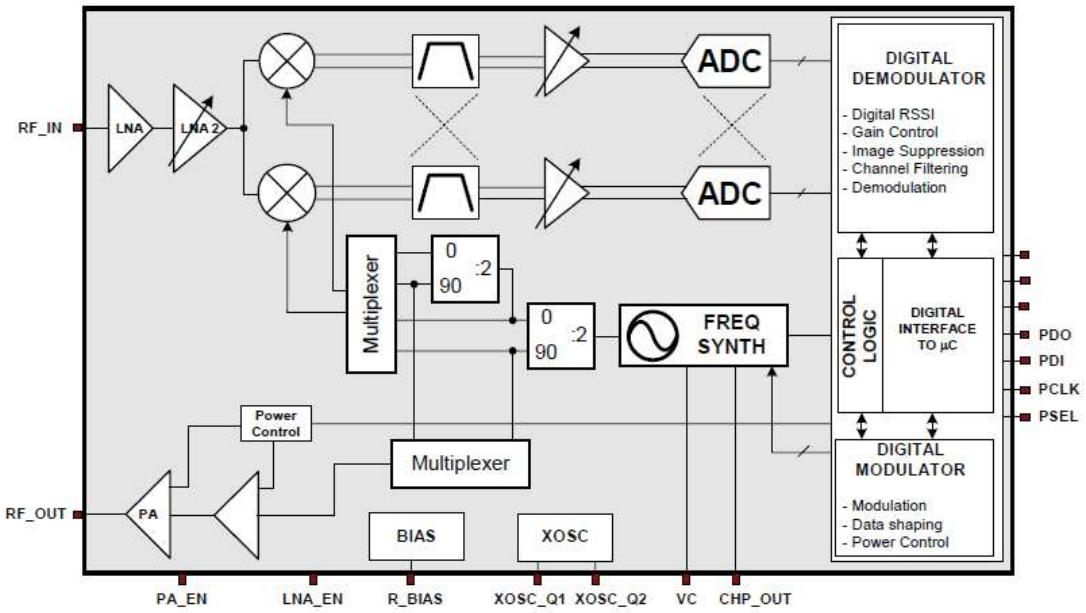


Figura 4.2. Schema a blocchi semplificato del CC1020

Dallo schema in Figura 4.2, osserviamo come in ricezione il segnale RF proveniente dall'antenna viene posto in ingresso al CC1020 per mezzo del pin RF_IN. Questo segnale viene amplificato da due LNA in serie (il primo a guadagno fisso, il secondo a guadagno variabile a seconda della potenza del segnale d'ingresso), poi convertito a IF nei due rami I e Q. I segnali in entrambi i rami vengono filtrati e amplificati, quindi digitalizzati. Controllo automatico del guadagno, filtraggio fine del canale, demodulazione e sincronizzazione dei bit vengono tutti eseguiti digitalmente. L'uscita dello stream di bit dei dati demodulati è sul pin DIO. In trasmissione l'ingresso dello stream dei bit di dati da modulare è ancora il pin DIO. Questo stream viene modulato in FSK o opzionalmente in GFSK, portato in un PA e infine in uscita sul pin RF_OUT, diretto verso l'antenna. I pin RF_IN e RF_OUT possono essere uniti in un'unica linea adattata verso l'antenna, oppure, come nel nostro caso in cui abbiamo un amplificatore di potenza a valle di RF_OUT, utilizzare uno switch esterno.

4.1.2 Microcontrollore: TI MSP430

Come detto, il CC1020 è un dispositivo che necessita di essere gestito esternamente da una MCU. Durante l'esercizio di una comunicazione secondo un qualsiasi protocollo, la MCU ha il compito di configurare il modo di funzionamento (ricezione, trasmissione, power down) e i relativi parametri (primariamente frequenza, tipo di modulazione, potenza di trasmissione) e gestire lo stream che si sta ricevendo o trasmettendo. Similmente al transceiver, anche la scelta della MCU è stata condotta ricercando caratteristiche quali dimensioni ridotte, consumi bassi e necessità di pochi componenti esterni. La scelta è ricaduta sui microcontrollori della Texas Instruments e in particolare della famiglia MSP430 [11]. Il dispositivo testato nel prototipo usato per questa tesi è il MSP430F149 (Figura 4.3), ma, come si vedrà nel paragrafo 4.4, è previsto in futuro l'utilizzo di uno della stessa famiglia ma con prestazioni più elevate.

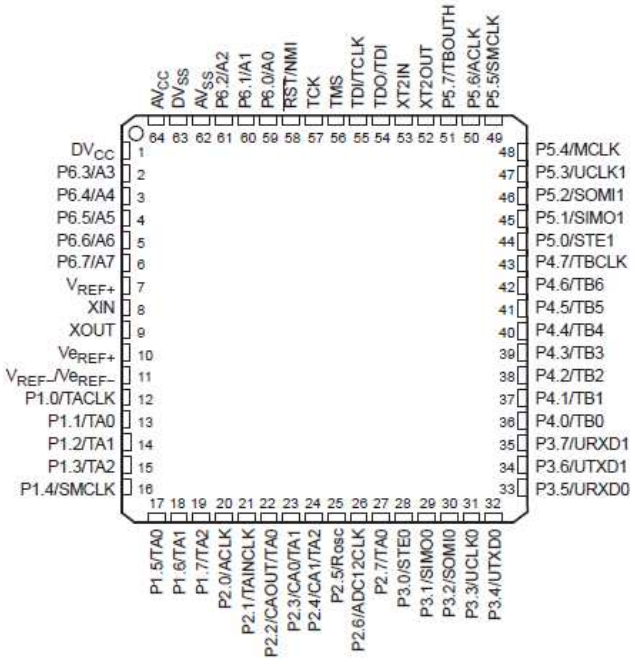


Figura 4.3. Package del MSP439F149

La serie MSP430F1xx è caratterizzata da una CPU a 16 bit con registri a 16 bit e 48 pin di I/O. Le applicazioni tipiche, includono sistemi che acquisiscono i segnali analogici, li convertono in segnali digitali e poi li processano al fine di poterli visualizzare o

trasmettere ad un sistema host. Le caratteristiche principali del microcontrollore MSP430F149, sono le seguenti:

- bassa tensione di alimentazione 1.8-3.6 V;
- bassi consumi di alimentazione:
    - Modalità attivo: 280µA a 1MHz, 2.2V;
    - Modalità standby: 1.6µA;
    - Modalità spento (con mantenimento dei dati in RAM): 0.1µA;
- passaggio dallo stato standby allo stato attivo, in meno di 6µs;
- architettura a 16 bit, con tempo di esecuzione per istruzione di 125 ns;

Tutte le operazioni sono effettuate come funzioni di registro, insieme ai sette modi d'indirizzamento per l'operando di sorgente e ai quattro modi d'indirizzamento per l'operando di destinazione. La CPU è integrata con 16 registri che provvedono a ridurre il tempo di esecuzione delle istruzioni. Il tempo di esecuzione delle operazioni tra registri è di un ciclo di clock della CPU. Quattro dei registri, da R0 a R3, sono dedicati al program counter, allo stack pointer e allo stato dei registri. Tutti i registri rimanenti, sono destinati per tutti gli usi.

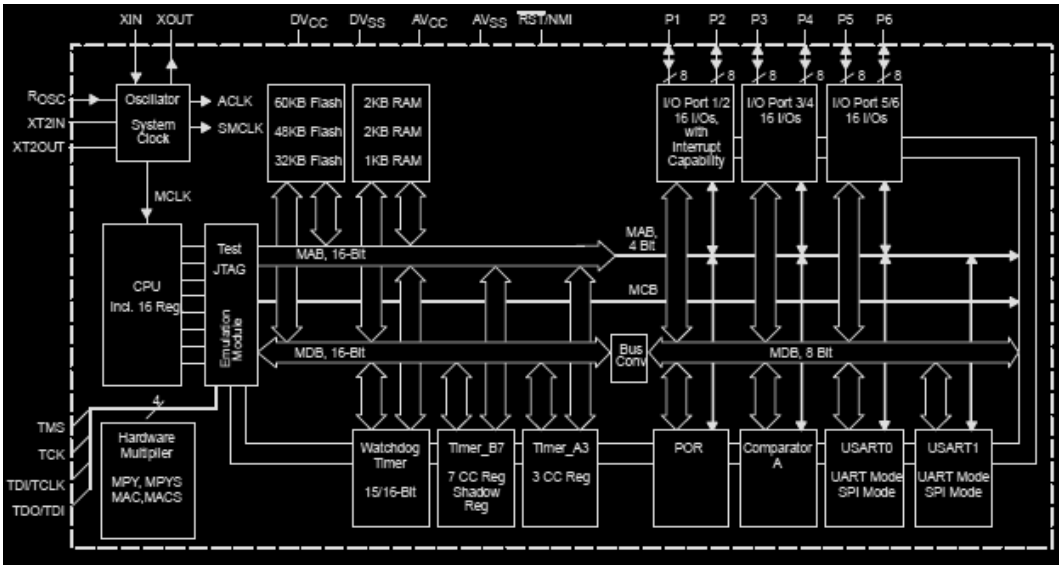


Figura 4.4. Schema a blocchi del microcontrollore MSP430F149

4.2 Interfaccia di configurazione e generalità sul processo di configurazione

Durante il funzionamento del protocollo il microprocessore configura il CC1020 scrivendo nei suoi registri interni determinati valori esadecimali cui corrispondono i modi/parametri di funzionamento desiderati. In ausilio al programmatore, la corrispondenza modo/parametro e valore esadecimale viene generata dal software *SmartRFStudio* fornito dalla casa. Fisicamente queste operazioni richiedono un collegamento tra microprocessore e transceiver denominato *Configure interface*, SPI-compatibile sui seguenti quattro pin del CC1020 (Figura 4.5):

- PSEL (pin 32): Programming chip select, è un ingresso collegato internamente a una resistenza di pull-up e quando posto ad un livello basso abilita le operazioni sui registri del chip, mentre tra un'operazione e l'altra viene lasciato libero (alto).
- PCLK (pin 1): Programming clock, è un ingresso su cui viene posto il clock per lo stream di configurazione.
- PDI (pin 2): Programming data input, è l'ingresso dei bit di configurazione dei registri.
- PDO (pin 3): Programming data output, è l'uscita dei bit di lettura del contenuto dei registri.

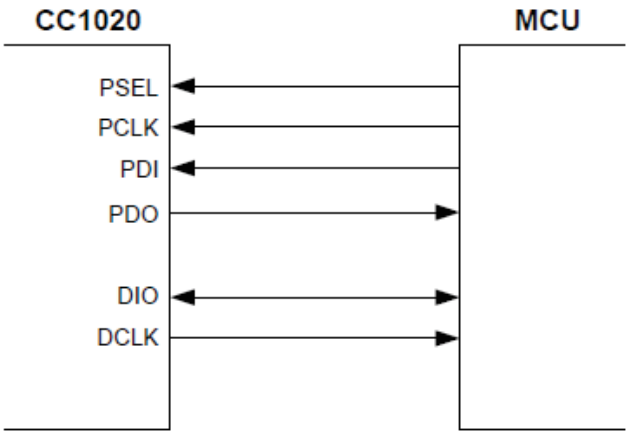


Figura 4.5. Interfacce hardware tra il CC1020 e la MCU

Mediante questa interfaccia è possibile scrivere o leggere i registri del CC1020. Operare su un registro richiede 7 bit per l'indirizzamento, più un 1 bit per specificare il modo d'accesso (lettura/scrittura), più 8 bit del dato scritto o letto. In totale abbiamo un frame di 16 bit per ogni operazione su un registro. Una configurazione/lettura completa agisce su 33 registri, quindi richiede il trasferimento di 528 bit. La velocità con cui questi bit vengono trasmessi dipende dalla frequenza del clock sul PCLK. Se questa è di 10 MHz (il massimo sostenibile dal chip), una configurazione piena richiederà meno di 53µs. Il dispositivo offre la possibilità di essere settato in *power down mode* (PD). A differenza di uno spegnimento, per il quale si ha la perdita della configurazione dei registri, il riavvio da PD richiederà l'invio di un solo frame, cioè un tempo inferiore a 2µs.

Nelle Figura 4.6 e Figura 4.7 si possono osservare rispettivamente le operazioni di scrittura e di lettura di un registro. Durante un ciclo di lettura/scrittura, come detto, vengono trasferiti 16 bit. I primi sono i 7 bit dell'indirizzo, da A6 fino a A0. Poi viene trasmesso il bit R/W, (1 per la scrittura, 0 per la lettura). Infine gli 8 bit del dato, da D6 fino a D0. Osserviamo che nel caso di scrittura, la trasmissione fluisce esclusivamente in ingresso al CC1020, cioè sempre sulla linea PDI, mentre la linea PDO non viene mai interessata. Invece in una operazione di lettura, dapprima viene interessata la linea d'ingresso PDI per trasmettere verso il CC1020 l'indirizzo e il tipo di operazione, dopodiché viene interessata la linea d'uscita PDO per ricevere dal CC1020 il valore contenuto nel registro interrogato.

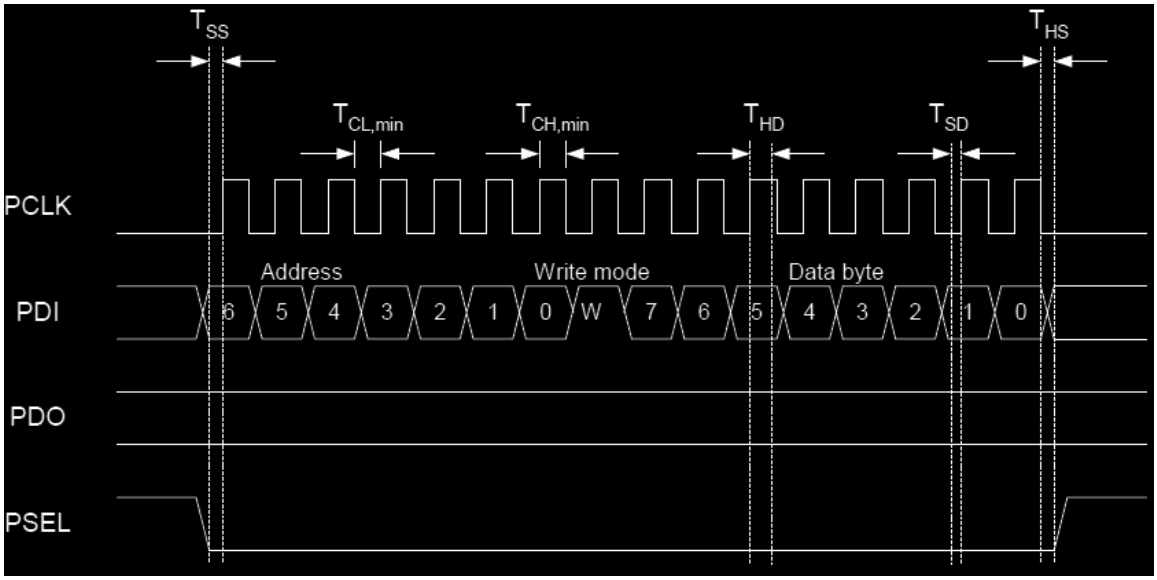


Figura 4.6. Operazione di scrittura di un registro

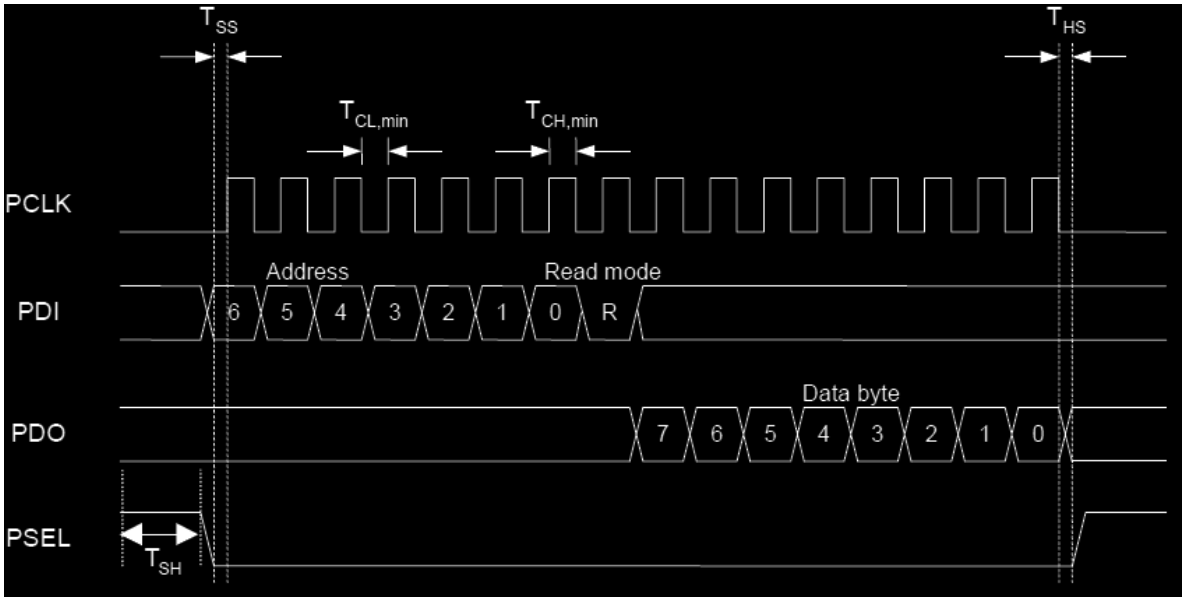


Figura 4.7. Operazione di lettura di un registro

Osserviamo che la lettura dei bit in ingresso sul PDI da parte del CC1020 e dei bit in uscita sul PDO da parte della MCU avviene sempre sul fronte di salita del clock. La commutazione dei bit in uscita sul PDO da parte del CC1020 avviene quindi in

corrispondenza dei fronti di discesa del PCLK. Quindi la lettura di questi bit dal PDO da parte del microcontrollore dovrebbe essere fatta in corrispondenza del fronte di salita del PCLK.

4.3 Interfaccia dati

La *data interface* è quella mediante cui il CC1020 fornisce lo stream di dati alla MCU quando in ricezione o accetta lo stream di dati dalla MCU quando in trasmissione.
A seconda della configurazione del registro INTERFACE, essa è costituita da due o tre pin (Figura 4.5):

- DIO (pin 8): è l'ingresso dei dati quando in trasmissione, ed eventualmente l'uscita dei dati quando in ricezione.
- LOCK (pin 9): è l'uscita dei dati quando non si vuole utilizzare il precedente pin in modo bidirezionale.
- DCLK (pin 7): è il pin in cui il CC1020 fornisce sia in modalità ricezione che in modalità trasmissione il clock da usare per i dati in ingresso o in uscita sul pin DIO in caso di formati sincroni.

Il prototipo utilizzato in laboratorio non utilizza il pin LOCK, ma solo il pin DIO in modalità bidirezionale.

Il CC1020 permette tre possibili formati con cui i dati possono essere scambiati con la MCU attraverso la Data Interface:

1) Synchronous NRZ mode
2) Synchronous Machester encoded mode
3) Transparent Asynchronous UART mode

La nostra realizzazione utilizza la prima modalità. In trasmissione, i dati vengono portati al pin DIO dove vengono letti in corrispondenza del fronte di salita del clock sul pin DCLK. In ricezione, i dati vengono forniti ancora sul pin DIO dove la MCU dovrebbe leggerli in corrispondenza del fronte di salita del clock sul pin DCLK (Figura 4.8).

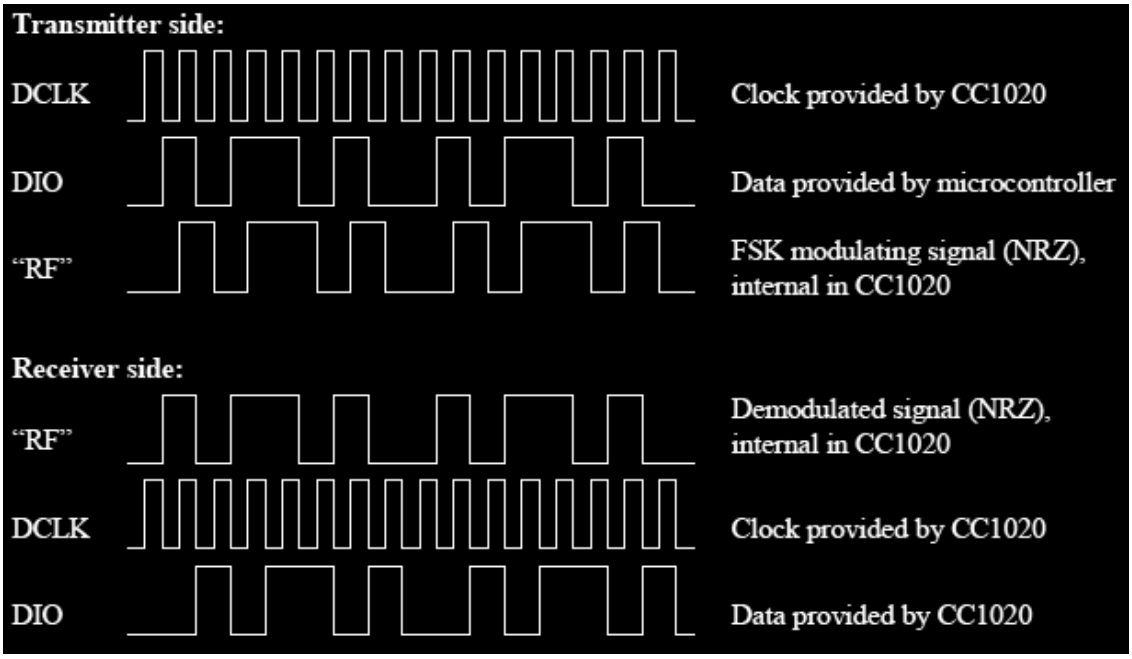


Figura 4.8. Synchronous NRZ mode

4.4 Prototipo della On-Board Radio Frequency module

Lo sviluppo dello schema elettrico e la realizzazione di un primo prototipo del modulo di comunicazione utilizzato in laboratorio è stata eseguita da un precedente tesista, di cui viene riportato in Figura 4.9 lo schema a blocchi. Durante il presente lavoro di tesi si è fabbricato un secondo prototipo identico al primo, al fine di poter eseguire successivamente test di comunicazione tra l'uno e l'altro (Figura 4.10).

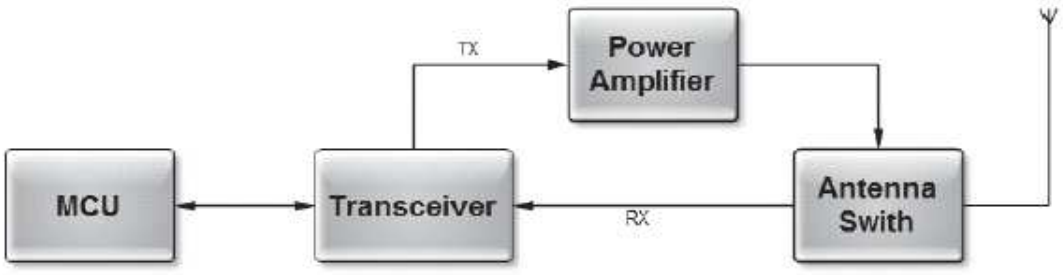


Figura 4.9. Schema a blocchi del modulo di comunicazione

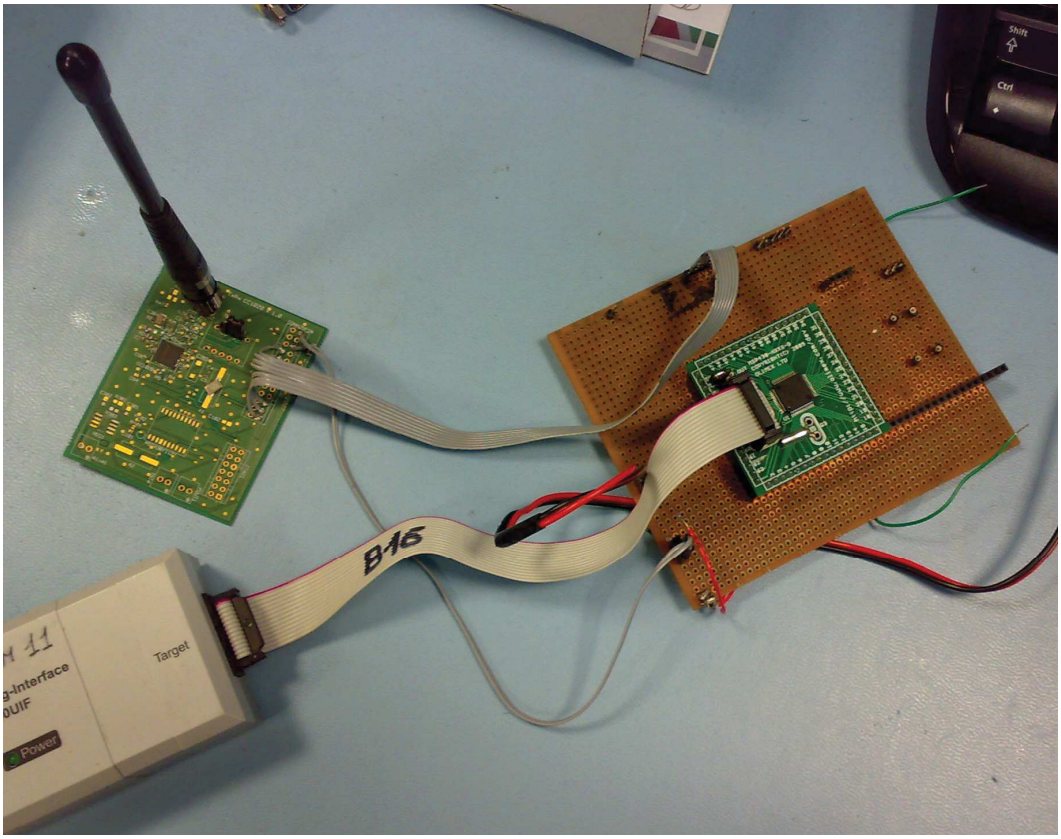


Figura 4.10. Prototipo del modulo di comunicazione utilizzato in laboratorio

Come detto in precedenza, sono in via di sviluppo i due nuovi moduli di comunicazione per i canali a 437 MHz e 2.4 GHz. La principale differenza del nuovo modulo a 437 MHz rispetto al vecchio, è l'utilizzo di un MCU più performante, cioè l'MSP430F5437 al posto dell'MSP430F149. Nella Figura 4.11 e Figura 4.12 è mostrato il loro diagramma delle classi in UML.

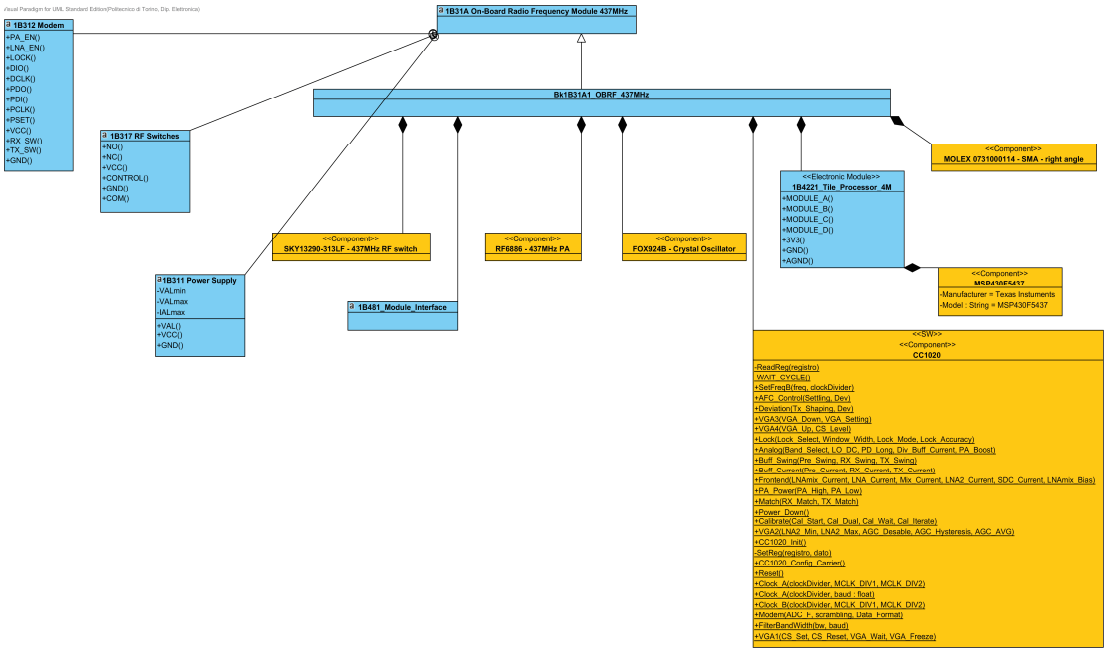


Figura 4.11. Diagramma UML per il nuovo modulo a 437 MHz

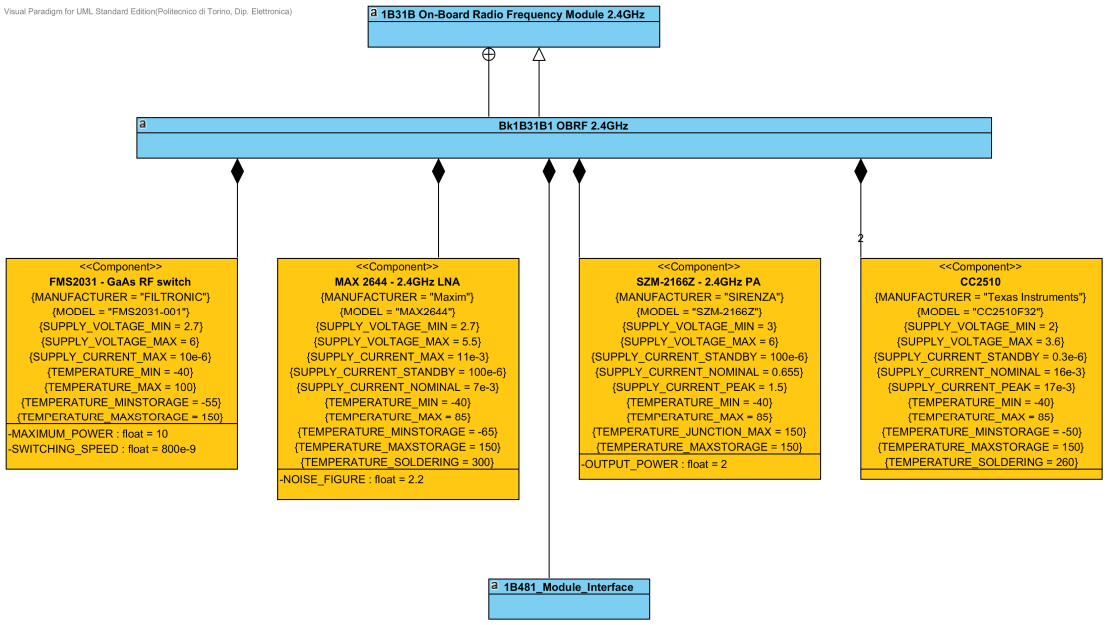


Figura 4.12. Diagramma UML per il nuovo modulo a 2.4 GHz

Le principali differenze del modulo a 2.4 GHz rispetto a quello a 437 sono l'utilizzo di un transceiver che al proprio interno ingloba anche il microcontrollore (CC2510) e l'utilizzo di uno switch d'antenna allo stato solido nel modulo a 2.4 GHz ed

elettromeccanico nel modulo a 437 MHz. L'utilizzo dello switch si rende necessario in entrambi i moduli per via della presenza del PA.

4.5 Link-budget

In questo paragrafo si calcola il link budget in uplink e in downlink.al fine di caratterizzare in termini del BER la qualità della trasmissione.

Il calcolo del link budget si basa sulla equazione della trasmissione:

$$P_R = P_T G_T G_R \frac{1}{\alpha_0} \frac{1}{\alpha_i} \frac{1}{\alpha_m} \tag{4.1}$$

dove:

- P_R e P_T sono rispettivamente la potenza in ricezione e in trasmissione
- G_R e G_T sono rispettivamente i guadagni delle antenne in ricezione e in trasmissione
- $\alpha_0 = \left(\frac{4\pi R}{\lambda}\right)^2$ è l'attenuazione di propagazione nello spazio libero
- α_i è la perdita di implementazione
- α_m è la extra-attenuazione media del mezzo trasmissivo

L'espressione (4.1) espressa in forma logaritmica diventa:

$$P_R\big|_{dBm} = P_T\big|_{dBm} + G_T\big|_{dB} + G_R\big|_{dB} - \alpha_0\big|_{dB} - \alpha_i\big|_{dB} - \alpha_m\big|_{dB} \tag{4.2}$$

Consideriamo il satellite equipaggiato con una antenna a doppia elica con guadagno 2dBi e la generica ground station di radioamatore equipaggiata con una antenna UHF 4 crossed Yagi (12dBi). Per l'antenna in ricezione assumiamo inoltre una attenuazione per disadattamento di polarizzazione di 3dB.

Considerando la Figura 4.13, dal teorema di Carnot si può calcolare la distanza R del satellite in funzione dell'elevazione e della altitudine con la formula:

$$R = -R_E \sin\theta_{EL} + \sqrt{R_E^2\left(\sin^2\theta_{EL} - 1\right) + R_{SV}^2} \tag{4.3}$$

con:

- R_E raggio della Terra
- θ_{EL} elevazione all'orizzonte
- R_{SV} distanza satellite – centro della Terra

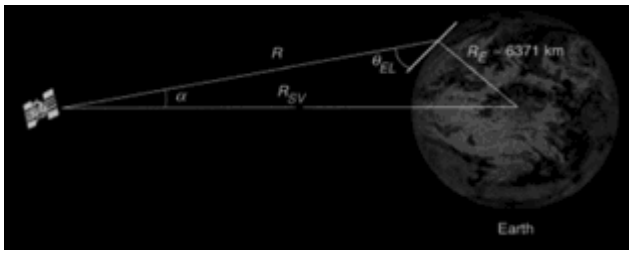


Figura 4.13. Distanza del satellite in funzione dell'elevazione e della altitudine

Ricordiamo che le orbite LEO sono comprese tra i 160 e i 2000 Km di altitudine. Per i nostri calcoli assumiamo una altitudine di 600Km e l'elevazione per il caso peggiore, cioè nulla (nella pratica la visibilità si interrompe prima). In queste condizioni risulta:

$$R = -R_E + \sqrt{-R_E^2 + R_{SV}^2} = 2830.83\,Km \tag{4.4}$$

e

$$\alpha_0 = \left(\frac{4\pi R}{\lambda}\right)^2 = 152.3\,dB \tag{4.5}$$

Per la perdita di implementazione (non idealità costruttive) e per la extra-attenuazione media del mezzo trasmissivo (propagazione in atmosfera) in caso di un canale satellitare, consideriamo i valori realistici α_i = 4 dB e α_m = 2 dB.

I calcoli dei link budget per la tratta in salita e in discesa sono rappresentati in Tabella 4.1 e Tabella 4.2.

Grandezza	**Operazione**	**Valore**
Potenza trasmessa P_T	+	47 dBm (50 W)
Guadagno antenna trasmittente G_T	+	12 dBi
Guadagno antenna ricevente G_R (incluso disadattamento di polarizzazione)	+	-1 dBi
Attenuazione di tratta α_0	-	152.3 dB
Perdite di implementazione α_i	-	4 dB
Attenuazione media del mezzo α_m	-	2 dB
Potenza ricevuta P_R		**-100.3 dBm**

Tabella 4.1. Link budget in uplink

Grandezza	Operazione	Valore
Potenza trasmessa P_T	+	34.8 dBm (3 W)
Guadagno antenna trasmittente G_T	+	2 dBi
Guadagno antenna ricevente G_R (incluso disadattamento di polarizzazione)	+	9 dBi
Attenuazione di tratta α_0	-	152.3 dB
Perdite di implementazione α_i	-	4 dB
Attenuazione media del mezzo α_m	-	2 dB
Potenza ricevuta P_R		**-112.5 dBm**

Tabella 4.2. Link budget in downlink

Si può dimostrare che il rapporto segnale-rumore al ricevitore SNR al ricevitore vale:

$$SNR\big|_{dB} = P_R\big|_{dB} - N\big|_{dB} - NF\big|_{dB} \tag{4.6}$$

dove:

- P_R è la potenza ricevuta, calcolata nel link-budget sopra;
- N è il livello del rumore termico al ricevitore ideale;
- NF è la cifra di rumore del ricevitore.

$$N = kTB \tag{4.7}$$

dove:

- k è la costante di Boltzmann;
- T è la temperatura totale al ricevitore;
- B è la banda del segnale.

In uplink consideriamo il ricevitore del satellite alla temperatura di 360K. Quindi:

$$N = 1.38 \cdot 10^{-23} J/K \cdot 360K \cdot 12.500Hz = 6.21 \cdot 10^{-14} mW \ \left(-132.1dBm\right) \tag{4.8}$$

Per il valore di NF consideriamo quello del CC1020, cioè 17 dB.

Grandezza	Operazione	Valore
Potenza ricevuta P_R		-100.3 dBm
N	-	-132.1 dBm
NF (CC1020)	-	17 dB
SNR		**14.8 dB**

Tabella 4.3. SNR al ricevitore del satellite

In downlink consideriamo il ricevitore della ground station alla temperatura di 320K, quindi:

$$N = 1.38 \cdot 10^{-23} J/K \cdot 310K \cdot 12.500Hz = 5.35 \cdot 10^{-14} mW \ \left(-132.7dBm\right) \tag{4.9}$$

Per il valore di NF consideriamo quello di un normale transceiver radioamatoriale, cioè 8 dB.

Grandezza	**Operazione**	**Valore**
Potenza ricevuta P_R		-112.5 dBm
N	-	-132.7 dBm
NF (transceiver)	-	8 dB
SNR		**12.2 dB**

Tabella 4.4. SNR al ricevitore della ground station

Noto che per la FSK :

$$P_b(e) = \frac{1}{2} erfc \sqrt{\frac{1}{2} \frac{E_{bRX}}{N_0}} \tag{4.10}$$

di cui si riporta il plot in Figura 4.14 (curva nera dato che la FSK ha una costellazione ortogonale) [12] si ricava:

$$P_b(e)_{UPLINK} \cong 2 \cdot 10^{-8} \tag{4.11}$$

e

$$P_b(e)_{DOWNLINK} \cong 3 \cdot 10^{-5} \tag{4.12}$$

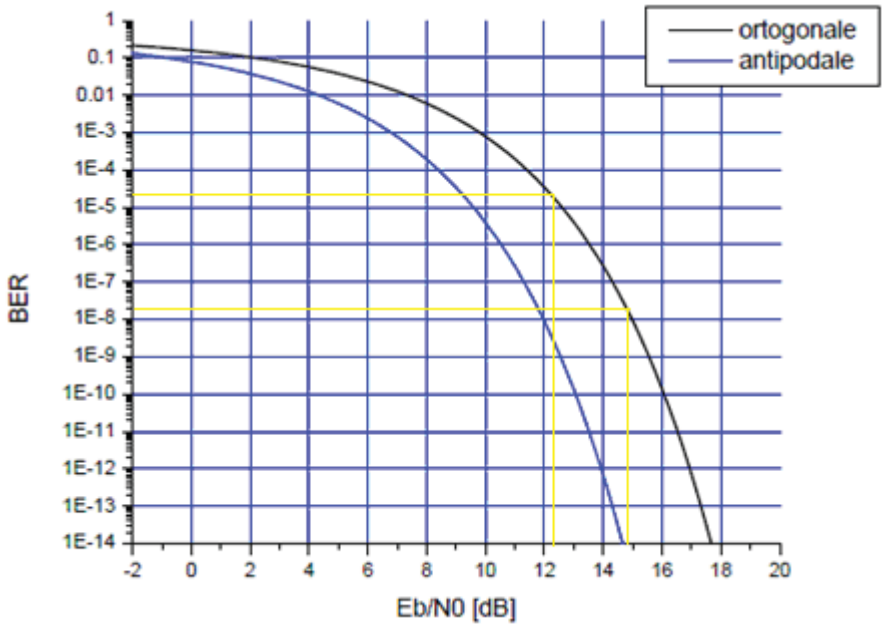


Figura 4.14. Bit error rate vs. signal to noise ratio

Capitolo 5

Il protocollo di comunicazione AX.25

In questo capitolo si descrive il protocollo AX.25 nella modalità connection-less, utilizzata nel sottosistema di comunicazione di AraMiS.

5.1 Introduzione

Nel 1984 la Internation Amateur Radio Union (IARU) incontrò la American Radio Relay League (ARRL) con l'intento di definire uno standard e una regolamentazione comune per le comunicazioni digitali tra radioamatori. Venne così formalizzato il protocollo AX.25, derivato dal protocollo X.25. Attualmente si è giunti alla versione 2.2, pubblicata nel 1996 dalla ARRL e TAPR Corporation [13].

Il protocollo viene usato in ambiente radioamatoriale per collegare stazioni in modalità point-to-point o in rete su canali half o full-duplex. Le sue funzionalità occupano il livello datalink e trasporto (quando usato in modalità connected) del modello ISO/OSI, indipendentemente dall'esistenza di protocolli di livello superiore. A differenza di altri protocolli appartenenti al livello data-link, dove si possono identificare un dispositivo master (DCE) e più dispositivi slave (DTE), l'AX.25 è un protocollo bilanciato, dove cioè le diverse stazioni radioamatoriali che condividono un canale radio sono dispositivi della stessa classe (TNC).

Il protocollo AX.25 permette:

- il trasferimento di blocchi di dati tra TNC, incapsulati in frame;
- comunicazioni in modalità connectionless per lo scambio di singoli frame (solo rilevazione degli errori introdotti dal canale di comunicazione)
- comunicazioni in modalità connected (con funzioni di recupero d'errore);

- di supportare semplici funzionalità di rete, per mezzo di dispositivi detti *digipeater*.

5.2 Frame in uso nel protocollo AX.25

Nell'AX.25 ci sono tre tipi basici di frame:

a) Unnumbered frame (U frame)
b) Supervisory frame (S frame)
c) Information frame (I frame)

Ogni frame è l'insieme di parti più piccole unite in sequenze logiche univoche, dette *field* (campi). Ogni field è composto di multipli interi di byte e ha specifiche funzioni. I possibili frame sono illustrati qui di seguito:

Flag	Address	Control	FCS	Flag
8 Bits	112 - 560 Bits	8 Bits	16 Bits	8 Bits
1111110	Callsigns & SSIDs of Destination, Source and Optionally, Digipeaters	Frame Type	Calculated Value	1111110

Figura 5.1. Struttura di un frame U e S

Flag	Address	Control	PID	Information	FCS	Flag
8 Bits	112 -560 Bits	8 Bits	8 Bits	N X 8 Bits	16 Bits	8 Bits
1111110	Callsigns & SSIDs of Destination, Source and Optionally, Digipeaters	Frame Type	Layer 3 Prot. Type	User Data	Calculated Value	1111110

Figura 5.2. Struttura di un frame I

I campi vengono trasmessi da sinistra verso destra, e i bit dentro di essi dal LSB fino al MSB, con eccezione di quelli del campo FCS.

Osserviamo che poiché le comunicazioni tra le stazioni di Terra e il nostro satellite avverranno in modalità connectionless, non siamo interessati ai tipi di frame U ed S.

Campo Flag

È una particolare sequenza lunga 8 bit che occorre all'inizio e alla fine di un frame e che consente al ricevitore di delimitare il frame. Due frame consecutivi possono condividere lo stesso flag che in tal caso denoterà la fine di uno e l'inizio del successivo. Questa sequenza consiste di un bit a "0" seguito da sei bit a "1" ed infine di nuovo un bit a "0":

“01111110” ovvero il valore 0x7E su 8 bit. L’utilizzo del flag come separatore impone il non ripetersi della relativa sequenza all’interno del frame. A tal fine, come si vedrà di seguito, si utilizza un meccanismo detto *bit stuffing*.

Campo Addess

Il campo Address contiene in successione l’indirizzo destinatario, l’indirizzo sorgente e opzionalmente l’indirizzo di ogni digipeaters che si vuole attraversare, fino ad un massimo di 8.

Dobbiamo premettere che nel packet radioamatoriale l’indirizzo di una stazione corrisponde al suo nominativo (callsign), cioè una stringa lunga non più di 6 caratteri. Quando all’interno della stessa stazione sono contemporaneamente attive più apparecchiature packet, queste possono essere distinte aggiungendo un carattere (un byte) detto SSID (Secondary Station Identification). Tale carattere permette di indirizzare fino a 16 apparecchiature diverse interne ad un'unica stazione.

Ogni indirizzo rappresentato dentro il campo Address deve essere fatto esattamente di 7 caratteri in Extended ASCII (8 bit per carattere), 6 per il callsign e 1 per il SSID. Nominativi che per qualche motivo sono lunghi meno di 6 caratteri quando inseriti dentro questo campo sono fatti seguire da un opportuno numero di caratteri ASCII di “Spazio” (padding). Ogni indirizzo all’interno del campo Address occupa quindi 7 byte, da cui la lunghezza indicata nelle figure che è uguale ad un minimo di 112 bit quando sono indicati solo gli indirizzi destinatario e sorgente, fino ad un massimo di 560 bit quando oltre questi due indirizzi sono indicati quelli di 8 digipeaters. Osserviamo che il destinatario, non conoscendo il path e quindi il numero di digipeater attraversati, non conosce a priori dove finisce il campo Address e inizia il campo Control. A tale fine si è pensato di operare uno shift verso sinistra della codifica di ogni carattere (ciò è possibile dato che avendo usato la codifica Extended ASCII il MSB era obbligatoriamente “0”). Dopo lo shift, il LSB di ogni indirizzo risulterà un bit “0”. L’ultimo indirizzo sarà quindi univocamente identificabile forzando a “1” il suo LSB.

Campo Control

Il Control Field contiene i numeri di sequenza dei frame ricevuti o inviati utilizzati nella modalità connessa. Può essere lungo 1 o 2 byte.

Campo PID (solo I o UI frame)

Il campo PID (Protocol Identifier field) identifica il tipo di protocollo di livello 3 in uso, ammesso che ve ne sia uno, secondo la codifica in Tabella 5.1. Nel nostro caso non dobbiamo specificare nessun protocollo tra quelli in tabella, quindi per quanto concerne noi, il campo PID contiene il valore 0xF0.

HEX	MSB LSB	Translation
**	yy01yyyy	AX.25 layer 3 implemented.
**	yy10yyyy	AX.25 layer 3 implemented.
0x01	00000001	ISO 8208/CCITT X.25 PLP
0x06	00000110	Compressed TCP/IP packet. Van Jacobson (RFC 1144)
0x07	00000111	Uncompressed TCP/IP packet. Van Jacobson (RFC 1144)
0x08	00001000	Segmentation fragment
0xC3	11000011	TEXNET datagram protocol
0xC4	11000100	Link Quality Protocol
0xCA	11001010	Appletalk
0xCB	11001011	Appletalk ARP
0xCC	11001100	ARPA Internet Protocol
0xCD	11001101	ARPA Address resolution
0xCE	11001110	FlexNet
0xCF	11001111	NET/ROM
0xF0	11110000	No layer 3 protocol implemented.
0xFF	11111111	Escape character. Next octet contains more Level 3 protocol information.
Escape character. Next octet contains more Level 3 protocol information.	00001000	

Tabella 5.1. Definizione del campo PID

Campo Information (solo I o UI frame)

È il campo che contiene il payload e può essere lungo fino a 256 byte. È importante osservare che benché i dati inseriti all'interno di questo campo vengono passati in modo "trasparente" dall'end sorgente all'end destinatario lungo il link, comunque la sua lunghezza deve essere un multiplo pari di byte.

Campo FCS

Il campo FCS (Frame-Check Sequence) è uno spazio di 2 byte in cui l'end sorgente inserisce un valore CRC calcolato sul frame in trasmissione (campi Flag e CRC esclusi). Il destinatario lo ricalcola sulla base di quanto ricevuto e lo confronta con quello contenuto nel campo FCS. Ha compiti di verifica dell'integrità del pacchetto dopo la trasmissione attraverso il mezzo e viene calcolato secondo l'algoritmo suggerito nella raccomandazione ISO 3309 (HDLC). Si ricorda che è necessario trasmettere prima il byte meno significativo del FCS (in quanto contiene i coefficienti del termine più grande). La Tabella 5.2 riassume le proprietà del CRC – 16.

Tipo di errore	Probabilità di rilevazione
1 bit error	100%
2 bit error	100%
Odd error	100%
Burst error su 16 bit	100%
Burst error su 17 bit	99,9969%
Altri burst error	99,9984%

Tabella 5.2. Probabilità di rilevazione dell'errore del CRC - 16

Bit stuffing

Avendo usato la sequenza 01111110 per i flag di inizio e fine frame, si deve garantire che questa non occorra accidentalmente in nessuna altra parte del frame. Il Bit stuffing contemplato nel protocollo è un meccanismo che in trasmissione identifica pattern costituiti da cinque o più bit a "1" consecutivi e la interrompe introducendo un bit a "0". In ricezione ogni volta che vengono ricevuti cinque bit a "1" consecutivi, il bit a "0" immediatamente seguente viene scartato.

Capitolo 6

Firmware e test dell'On-Board Radio Frequency module

In questo capitolo vengono descritte le funzioni svolte dal firmware con cui è stato programmato il modulo OBRF. Segue la descrizione dei test eseguiti e i risultati ottenuti.

6.1 Descrizione del programma

Il software per il microprocessore TI MSP430F149 che realizza il protocollo è scritto in C nell'ambiente di sviluppo *IAR Embedded Workbench*. È possibile programmare il microprocessore collegando l'interfaccia USB fornita dalla casa (Figura 6.1). In modalità debbug è inoltre possibile osservare l'esecuzione del programma passo-passo da parte nel microcontrollore.

Figura 6.1. TI MSP430 USB Debugging Interface MSP-FET430UIF

Allo stato attuale il programma consente al sottosistema di comunicazione di comunicare con l'emulatore di terminale in esecuzione su PC in modo da:

- ricevere frame AX.25 e stampare a video l'indirizzo mittente, l'indirizzo destinatario e il payload;
- acquisire stringhe di caratteri dal terminale e trasmetterle dentro frame AX.25.

Osserviamo che il PC sostituisce in questa fase di sviluppo il ruolo dell'OBC.

Il software è composto da sei file di codice sorgente, brevemente riassunte nella Tabella 6.1.

File	Descrizione
main.c	Tramite opportune chiamate alle funzioni definisce il loop del processo.
CC1020.c	Definisce tutte quelle funzioni che si occupano di configurare il transceiver per disporsi nei diversi stati possibili (TX, RX, PD, WakeUpToRX, WakeUpToRX).
ax25.c	Contiene le funzioni principali che vengono chiamate dal main AX25_SendPacket e AX25_ReceivePacket che realizzano la trasmissione e la ricezione di un pacchetto secondo le regole e i meccanismi del protocollo AX.25.
uart.c	Contiene i driver che realizzano il collegamento seriale con la porta RS-232 del PC.
SPI.c	Contiene i driver che realizzano sulla porta USART0 la *configure interface* col CC1020.
timer.c	Include le funzionalità del timer usate in varie parti del programma.

Tabella 6.1. Breve descrizione dei moduli software

Di seguito sono descritte le funzionalità svolte dal main e da alcune routine di particolare rilevanza. Il codice completo è fornito in Appendice A.

6.1.1 main.c

Il main definisce il comportamento del sottosistema di comunicazione come rappresentato dal diagramma di flusso in Figura 6.2.

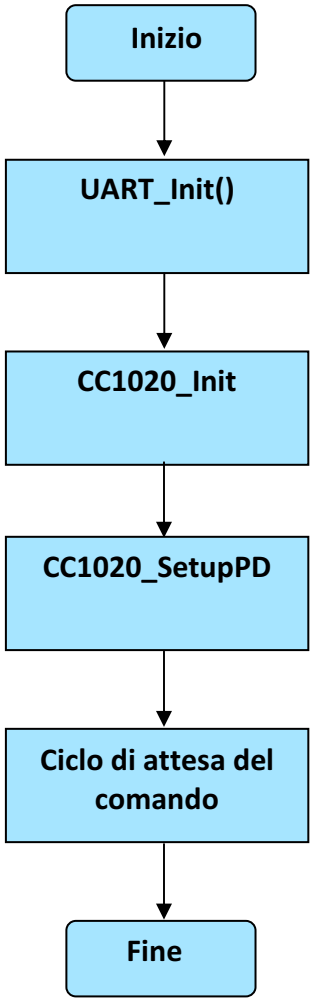


Figura 6.2. Diagramma di flusso definito nel main

Le prime due fasi rappresentano l'abilitazione della UART del microprocessore (che gli consente di poter comunicare con l'emulatore di terminale) e della configuration interface (che gli consente di poter configurare il CC1020). Il CC1020 viene poi messo in *Power Down mode* (PD). L'ultimo blocco rappresenta il ciclo di attesa del comando da parte del terminale, il cui diagramma di flusso è mostrato in Figura 6.3.

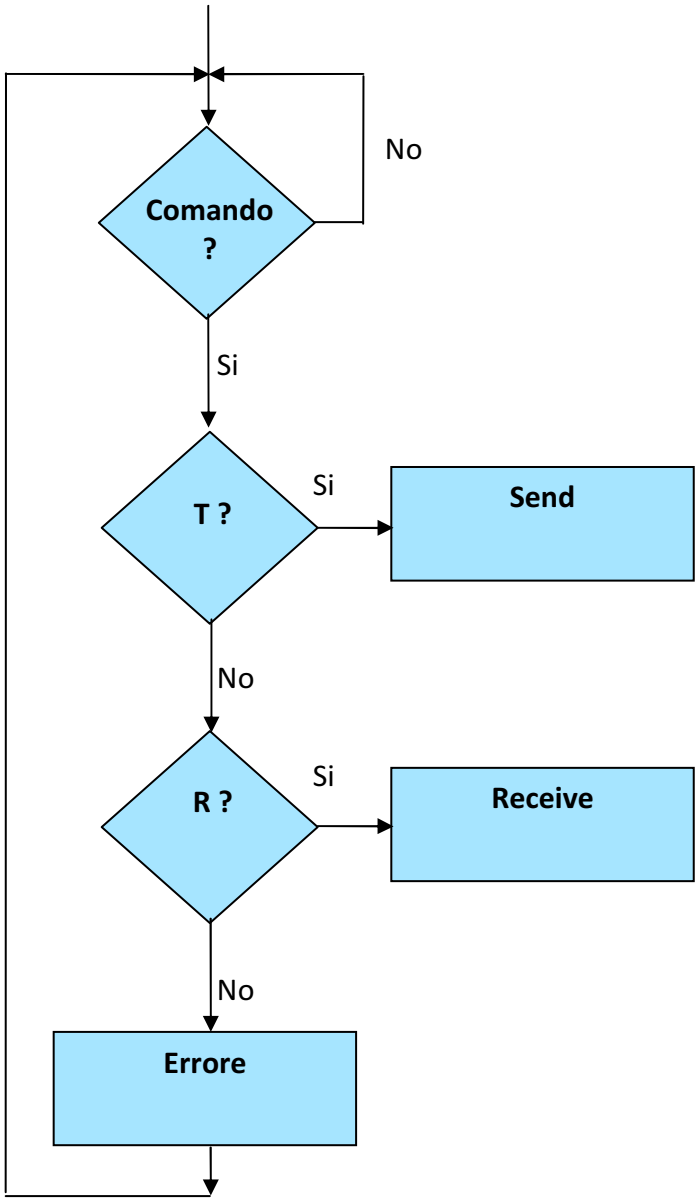


Figura 6.3. Ciclo di attesa dei comandi

Durante questo ciclo, mediante la routine `UART_ReceivePacket`, il sistema si pone in attesa di ricevere un comando dal terminale che rispetti la sintassi in Tabella 6.2.

Istruzione	Operazione svolta
`T+`*`n`*`+payload`	Pone l'OBRF nello stato di TX per un frame che contiene un payload di lunghezza `n` e contenuto come indicato nella parte `payload` del comando (max 255B).
`r`	Pone l'OBRF nello stato di RX fino alla ricezione di un frame o fino allo scadere di un timer. Se un frame viene ricevuto, il suo payload viene portato verso l'interfaccia.

Tabella 6.2. Le due istruzioni che costituiscono l'interfaccia dell'OBRF verso l'OBC

Se il comando non viene riconosciuto, viene restituito al terminale un segnale d'errore. In caso contrario, si entra nei blocchi Send o Receive, i cui diagrammi di flusso sono mostrati nella Figura 6.4 e nella Figura 6.5. Nel caso Send, il CC1020 viene configurato nel TXmode quindi viene chiamata la routine che si occupa di incapsulare la stringa nel frame AX.25 e di realizzare la sua trasmissione. La Data Interface col CC1020 viene quindi disabilita e il CC1020 messo in stato PD. Nel caso Receive, il CC1020 viene configurato nel RXmode, quindi viene chiamata la routine che si occupa di realizzare le funzioni relative alla ricezione dei frame AX.25. La fase di ricezione può terminare in tre casi: timeout, CRC non valido, frame ricevuto correttamente. Se il frame è stato ricevuto correttamente la routine `UART_SendPacket` si occuperà di trasmettere al terminale il messaggio contenuto nel suo interno, invece se il frame è stato ricevuto non correttamente verrà trasmesso un messaggio "NO CRC".

Sia che sia stato eseguito il blocco Send che il blocco Receive, al termine l'esecuzione ritorna all'inizio del loop, cioè con il CC1020 posto nello stato PD per risparmiare energia e il microprocessore in attesa di comandi provenienti dal terminale.

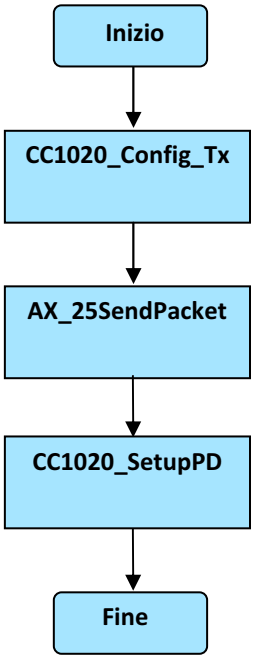


Figura 6.4. Diagramma di flusso del blocco Send

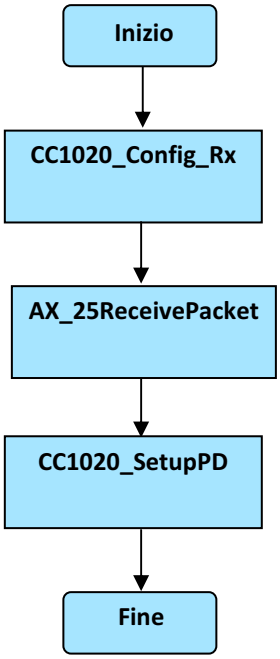


Figura 6.5. Diagramma di flusso del blocco Receive

Le routine `AX25_SendPacket` e `AX25_ReceivePacket` sono infine rappresentate rispettivamente nella Figura 6.6 e nella Figura 6.7. La trasmissione di un frame viene preceduto da un numero N configurabile di ripetizioni del FLAG. Segue la trasmissione delle varie parti che costituiscono la struttura del frame. La trasmissione si conclude con una duplice trasmissione del FLAG di fine. La routine di ricezione attiva un ciclo di riconoscimento di FLAG applicata alla sequenza di bit in uscita al CC1020. Quando vengono trovate M ripetizioni consecutive della sequenza di FLAG (0x7E), assume che i bit che seguono costituiscono la parte interna del frame, quindi vengono inseriti dentro un vettore. Questa fase si conclude con il calcolo del CRC e il confronto con quello ricevuto. Se il controllo è verificato, vengono ritornati i dati, altrimenti il valore NULL. Osserviamo che l'intera routine è controllata da un timer. Questo inizia all'inizio della routine e viene riavviato se si inizia a ricevere un frame. Quando il timer termina il suo conteggio, il controllo ritorna al processo chiamante, cioè al main. In pratica il timer stabilisce il tempo in cui il sistema può stare in "ascolto" quando gli si da un comando di ricezione.

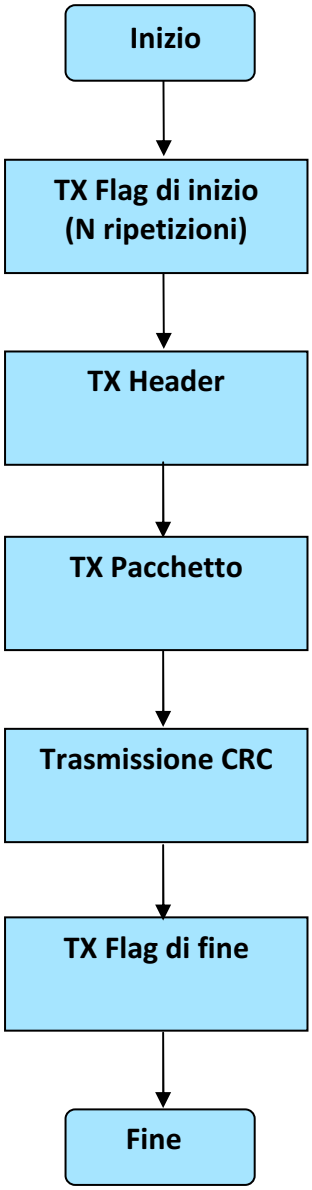


Figura 6.6. Diagramma di flusso del processo di trasmissione

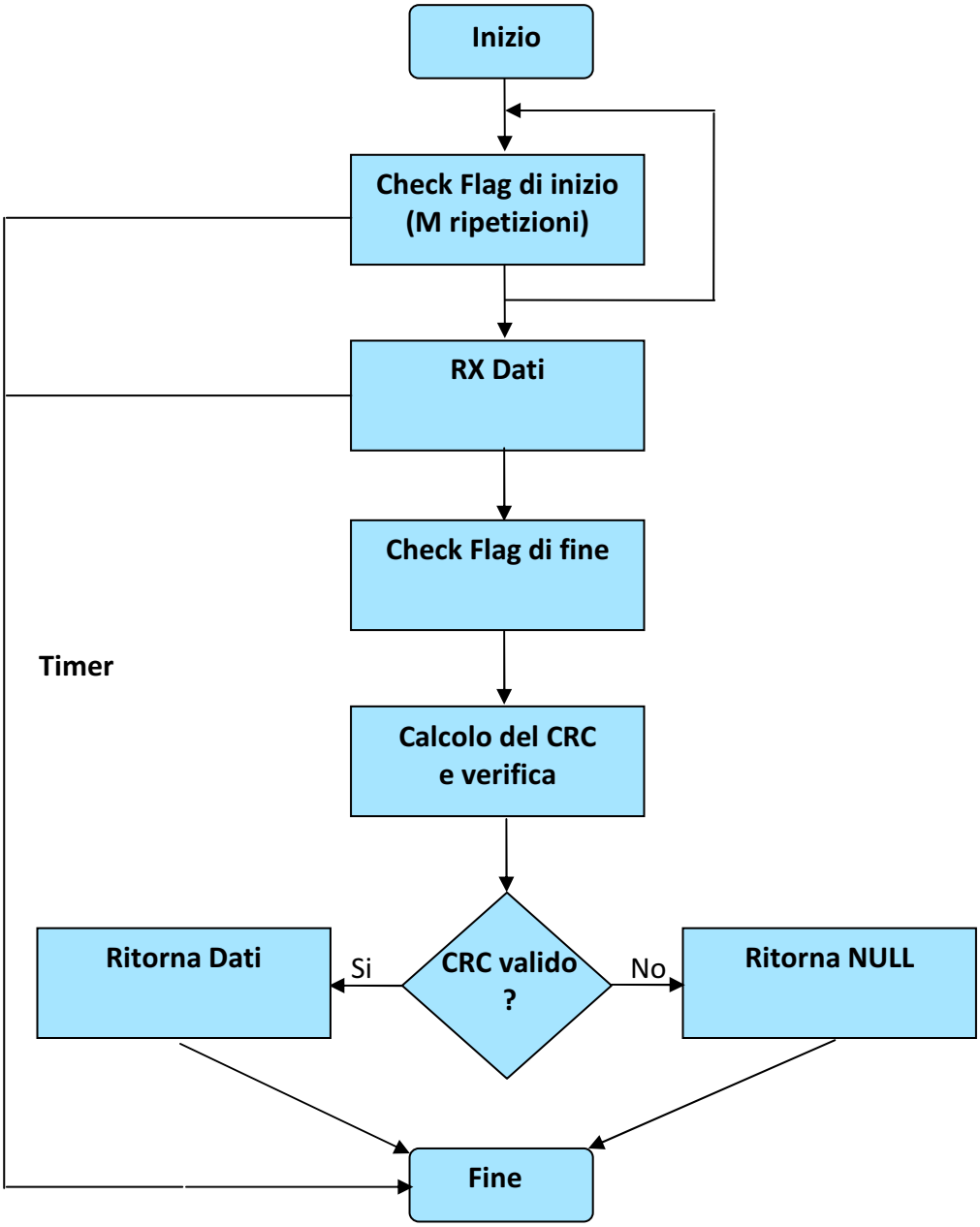


Figura 6.7. Diagramma di flusso del processo di ricezione

6.1.2 ax25.c

Include le funzioni principali che vengono chiamate dal main per la trasmissione e la ricezione di un pacchetto secondo il formato del frame AX25.

- AX25_SendByte (unsigned char byte, int flag)

  Permette di trasmettere un byte ed effettua il controllo del massimo numero di bit consecutive a 1 (bit stuffing). I parametri byte e flag rappresentano rispettivamente il byte da inviare e il flag che se messo a zero resetta il conteggio dei bit consecutivi uguali a 1.
- AX25_SendPacket (unsigned char *packet, unsigned int packet_len)

  Consente la trasmissione di un pacchetto di dati all'interno di un frame AX.25. I parametri packet e packet_len sono rispettivamente il puntatore al pacchetto da trasmettere e la sua lunghezza.
- AX25_ReceivePacket (unsigned char *data)

  Consente la ricezione di un frame AX.25 e la sua decodifica. Il parametro data è un puntatore al pacchetto decapsulato.
- pppfcs (char *cp, int len) e AX25_ComputeFCS (char *packet, int packet_len)

  Funzione che realizza l'algoritmo per il calcolo del FCS su un pacchetto ricevuto o in trasmissione.

6.1.3 cc1020.c

Definisce tutte quelle funzioni che consentono al microprocessore di attivare e configurare il CC1020 nei diversi stati possibili.

- CC1020_Calibrate (char pa_power)

  Provvede all'aggancio del PLL.
- CC1020_Config_Carrier, CC1020_Config_TX e CC1020_Config_RX

  Contengono i valori esadecimali necessari alla configurazione dei registri del CC1020 affinché il dispositivo funzioni secondo il modo di funzionamento portante, trasmissione e ricezione ed i parametri desiderati (frequenza, tipo di modulazione, potenza, eccetera).

- CC1020_ReadReg (char registro)

  Consente di leggere il valore contenuto in uno specifico registro.
- cc1020_Reset (void)

  Configura I registri del CC1020 con i valori iniziali predefiniti.
- CC1020_SetUpToRX (char RXANALOG, char PA_POWER)

  CC1020_SetupToTX (char txanalog, char pa_power)

  Configura il registro ANALOG per "svegliare" il CC1020 rispettivamente nel modo ricezione e trasmissione accendendo l'oscillatore al quarzo e il generatore di bias. Prima che di accendere il generatore di bias, il quarzo necessita di 2-5 ms per stabilizzarsi.
- CC1020_Init (void)

  Configura gli ingressi di configurazione del CC1020 (PSEL, PDI, PDO, PCLK) per poter procedere alla sua configurazione.
- CC1020_SetReg(char registro, char dato)

  È la funzione di base per configurare un certo registro.
- CC1020_SetupPD(void)

  Permette di porre il CC1020 nel Power Down mode, una modalità che spegne il sintetizzatore di frequenza, il generatore di bias e il quarzo, ma non cancella il contenuto dei registri, consentendo quindi una successiva configurazione più veloce.
- CC1020_WakeUpToRX (char RXANALOG) e

  CC1020_WakeUPToTX (char txanalog)

  Permettono una riattivazione del CC1020 rispettivamente nel modo ricezione e trasmissione a partire dallo stato PD. Non richiede quindi una completa riconfigurazione, ma solo la accensione del sintetizzatore di frequenza, il generatore di bias e il quarzo.

6.1.4 SPI.c

Realizza su una porta USART del MCU un collegamento SPI che costituisce la Data Interface col CC1020.

- SPI_Init (void)

  Abilita la porta USART0 del MSP430 per lavorare nel modo SPI.
- SPI_Disable (void)

  Disabilita la porta USART1 del MSP430 se precedentemente configurata per lavorare nel modo SPI.
- SPI_SendByte (unsigned char data) e SPI_ReceiveByte (void)

  Funzioni che permettono alla SPI rispettivamente di trasmettere e ricevere un byte dal buffer di trasmissione e ricezione.

6.1.5 uart.c

Implementa l'interfaccia del MCU verso la porta RS-232 del pc attraverso una porta USART, quindi gestisce lo scambio di dati in pacchetti tra il PC e il MCU.

- UART_Init (void)

  Abilita la porta USART0 del MSP430 per lavorare nel modo SPI. Definisce clock (2MHz) e baud-rate (9600kbps).
- UART_SendByte (unsigned char data) e UART_ReceiveByte (void)

  Consente rispettivamente di trasmettere e ricevere un byte dalla USART.
- printUART (unsigned char *message)

  Consente la stampa a video dei vari messaggi tramite il collegamento seriale RS-232 e un emulatore di terminale quale RealtTerm o Yat3.
- CRC (unsigned char *pDato, unsigned short dim)

  Effettua il CRC per i pacchetti ricevuti o trasmessi dalla USART.
- UART_ReceivePacket (unsigned char *pDato) e

  UART_SendPacket (unsigned char *pDato)

  Funzioni che permettono rispettivamente di ricevere e trasmettere un pacchetto dalla USART.

6.1.6 timer.c

Racchiude funzioni che si occupano delle temporizzazioni del sistema.

- TIMER_SetupTimer_ms (short volatile * semaforo)
  Contatore in ms.
- TIMER_Wait_us (short volatile semaforo)
  Contatore in µs.

6.2 Collaudi e test

Sono stati effettuati due collaudi di funzionamento con due configurazioni diverse. Nel primo si è fatto comunicare il prototipo con il sistema radioamatoriale formato dal transceiver Icom e il TNC PK-96 (Figura 6.8). Nel secondo, invece, si è fatto comunicare un prototipo con l'altro (Figura 6.9). I prototipi sono stati collaudati sia in ricezione che in trasmissione, invertendo il ruolo dei sistemi per ciascuna delle configurazioni. Sui PC sono stati utilizzati emulatori di terminale quali Yat3 o RealTerm a 9600 bps.

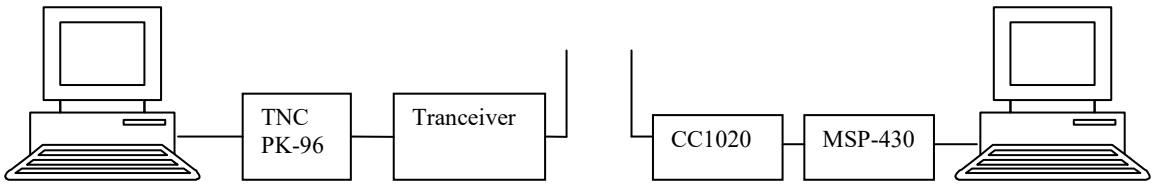


Figura 6.8. Comunicazione tra il prototipo e la stazione radioamatoriale

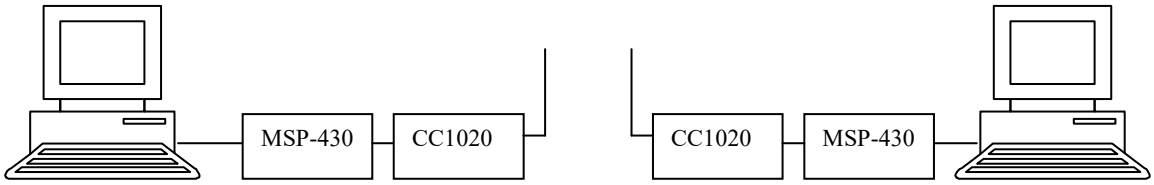


Figura 6.9. Comunicazione tra due prototipi

Con le configurazioni sopra sono stati eseguiti vari test di funzionamento e in particolare trasmissioni di singoli frame e trasmissione di sequenze di centinaia di frame senza intervallo tra l'uno e l'altro. In questo modo si è potuto collaudare oltre la

trasmissione/ricezione di singoli frame anche la possibilità di ricevere frame trasmessi con continuità. Dai test effettuati facendo comunicare i due prototipi, si sono evidenziati gli effetti derivanti dalla scelta dei valori di N (numero di ripetizioni del FLAG che anticipano l'inizio del frame) e di M (numero minimo di ripetizioni del FLAG che devono essere ricevuti per assumere che quello che segue al loro termine è il contenuto di un frame). Si è apprezzato come un numero N ridotto in trasmissione porti ad un aumento della probabilità di mancata ricezione dei frame al ricevitore, mentre un numero M elevato al ricevitore porti ad un aumento della probabilità di falsa ricezione (evidenziati da messaggi ACK NO). Per le condizioni in cui si sono svolti i test in laboratorio, si sono ottenuti buoni compromessi con M=10 e N=60, ottenendo la statistica seguente: 85% frame OK, 10% ACK NO, 5% frame non rilevati.

Questi risultati sono da considerarsi in termini qualitativi perché strettamente dipendenti dalle specifiche condizioni in cui stati effettuati i test in laboratorio (distanza, potenza utilizzata, antenne, eccetera).

Capitolo 7

Il protocollo elaborato

Le funzionalità realizzate dall'OBRF viste nel Cap. 6 sono inquadrabili nel livello fisico e di collegamento in riferimento al modello ISO/OSI in quanto i servizi che offre sono essenzialmente di ricevere il frame AX.25 dal canale radio, decapsulare il contenuto del campo Info e renderli disponibili per la lettura da parte dell'OBC o, nel senso inverso, di ricevere dall'OBC il contenuto destinato al campo Info, incapsularli nel frame AX.25 e trasmettere questo su canale radio. In questo capitolo viene invece descritto il protocollo proposto per i livelli superiori che consente una comunicazione tra Terra e satellite efficiente ed affidabile tenendo conto del contesto applicativo GENSO.

7.1 Introduzione

Il protocollo di livello superiore proposto in questo capitolo è implementato nell'OBC e poiché gestisce funzionalità legate al controllo dei duplicati in maniera differenziata secondo il tipo di comando ricevuto, può essere inquadrato nel livello trasporto e applicazione secondo un approccio *cross-layer*. Osserviamo che l'entità peer all'altro end dell'OBRF è una generica ground station (GSS equipaggiata di transceiver e TNC in kiss mode), mentre l'entità peer dell'OBC è la Mission Control Client (POLITO), come mostrato in Figura 7.1.

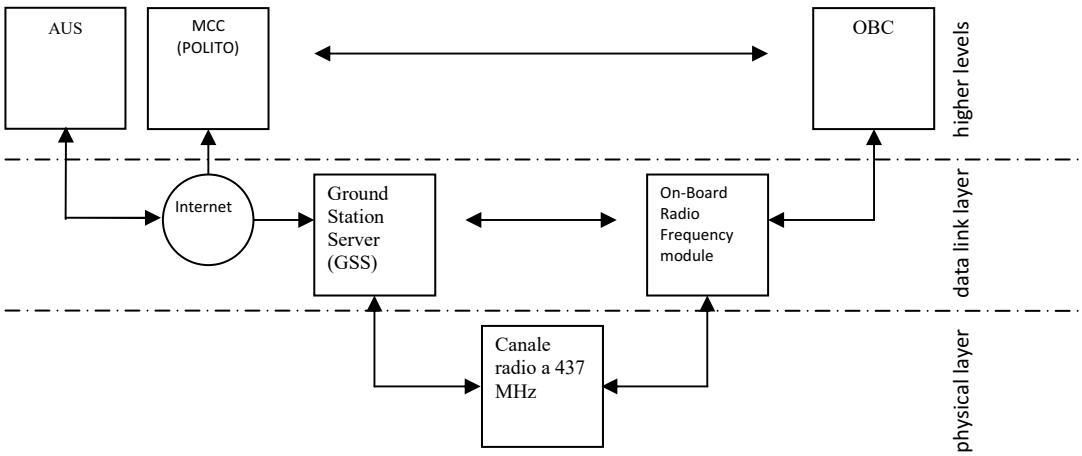


Figura 7.1. Strati ed entità della architettura Aramis-GENSO in riferimento al modello ISO/OSI

Per descrivere il protocollo proposto si fa riferimento alla Figura 7.2 dove vengono riportati i blocchi del satellite interessati nello sviluppo del protocollo. Del segmento di Terra consideriamo il Mission Controller, che nel nostro caso è il Politecnico di Torino, e una generica ground station radioamatoriale remota partecipante al progetto GENSO (Capitolo 2). Come detto, i due attori sono collegati via Internet.

Del satellite, oltre la memoria e le periferiche, consideriamo i moduli:

- On-board Radio Frequency, trattato nel Cap. 4, responsabile della ricezione e decodifica dei frame in upling e della codifica e trasmissione dei frame in downlink.
- Telemetry/Command Processor. I suoi principali compiti sono di leggere i comandi ricevuti in uplink, interpretarli ed eseguirli (o farli eseguire a una specifica periferica), di gestire la memoria (allocare spazio, inserire o recuperare i dati), e di avviare la trasmissione in downlink dei *mission data* (dati generati dal payload del satellite) e degli *housekeeping data* (informazioni relative al monitoraggio dello stato del satellite stesso inteso come supporto) [14].
- Housekeeping Processor, è responsabile di raccogliere misure dai vari sensori distribuiti nel satellite in modo da generare gli *housekeeping data*, quindi di inserirli in memoria. Appartengono a questi dati, ad esempio, quelli relativi alla

posizione orbitale, alla temperatura e allo stato di carica delle batterie, allo stato di disponibilità delle periferiche, eccetera.

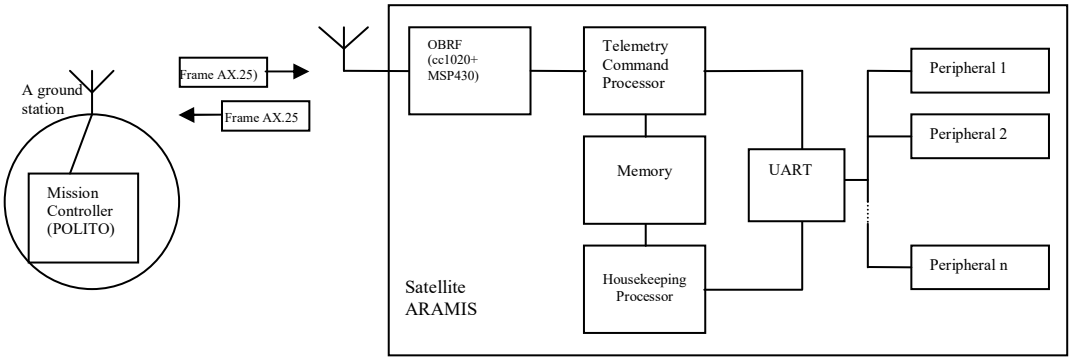


Figura 7.2. Schema semplificato del sistema di comunicazione a 437 MHz

Da un punto di vista generale si può pensare che ad ogni comando che il satellite riceve, corrisponderà al termine della sua esecuzione la generazione di dati di telemetria o payload che necessitano di essere recuperati a Terra. Il protocollo provvede a definire il formato dei messaggi e le modalità con cui tramite essi si possono far arrivare i comandi al satellite e i relativi dati a Terra.

Il disegno del protocollo è partito dalla distinzione di due tipi di commando:

- comandi a tempo di esecuzione lungo
- comandi a tempo di esecuzione breve.

I *comandi a tempo di esecuzione lungo*, d'ora in poi LONG, sono tali o perché danno luogo a esecuzioni che durano effettivamente un tempo lungo (ad esempio nel caso di un comando di Orientation) oppure perché la loro esecuzione è schedulata in un istante futuro (ad esempio un comando che richiede di acquisire un'immagine in una posizione orbitale distante).

I comandi a tempo di esecuzione breve, d'ora in poi SHORT, sono tali perché richiedono semplicemente il recupero e la trasmissione di dati già presenti in memoria. Questi dati si trovano già in memoria o perché generati periodicamente in automatico (funzioni di housekeeping) o perché sono il risultato dell'esecuzione di un comando LONG precedentemente ricevuto. Questa distinzione ha inciso in maniera determinante nel disegno della logica di funzionamento del protocollo dato che uno dei principali

requisiti che si vuole sia soddisfatto consiste nella possibilità di controllare quando far partire la trasmissione dei dati in downlink in modo tale che questi possano raggiungere una specifica ground station aderente al sistema GENSO di cui il Mission Controller ne abbia prenotato l'utilizzo. Per i comandi SHORT il requisito risulta automaticamente soddisfatto semplicemente autorizzando la trasmissione dei dati verso Terra automaticamente dopo la ricezione della richiesta. Infatti, poiché i dati sono immediatamente pronti, la ground station che ha trasmesso il comando sarà ancora in visibilità per raccoglierli. A ciò può fare eccezione l'ultimo comando trasmesso prima della fine del periodo di visibilità, evento, quest'ultimo, che si può interporre tra comando e risposta. In tal caso il comando viene perso e sarà necessaria una sua eventuale ritrasmissione.

Per soddisfare il requisito in riferimento ai comandi LONG, invece, sono stati considerati altri due approcci:

1. i dati vengono trattenuti in memoria e trasmessi automaticamente in un istante stabilito dal satellite stesso, ad esempio quando esso raggiunge una certa posizione orbitale;
2. i dati vengono trattenuti in memoria e trasmessi solo in seguito alla ricezione da Terra di un ulteriore comando di richiesta (GetData).

Dei due approcci, quello scelto è il secondo perché considerato più semplice e robusto per il nostro sistema.

7.1 Definizione del formato dei pacchetti

La Figura 7.3 mostra il processo di incapsulamento (o decapsulamento) del numero di sequenza all'interno del campo Ctlr e del pacchetto all'interno del campo Info. La linea tratteggiata rappresenta l'interfaccia tra l'OBRF che gestisce le funzioni di livello fisico/datalink e il TCP che gestisce le funzioni del protocollo a livello trasporto e applicazione. In questo paragrafo vengono mostrati i formati dei pacchetti, cioè dei comandi in uplink e dei messaggi in downlink gestiti dal TCP.

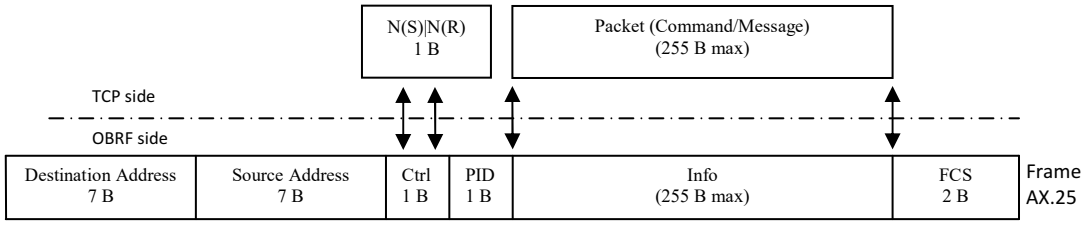


Figura 7.3. Fase di incapsulamento (decapsulamento)

Per il funzionamento del presente protocollo sono stati definiti i seguenti tipi di pacchetto.

In uplink:

1) backdoorCommand
2) ordinaryCommand

In downlink:

1) pureNACK
2) ordinaryDownlink

Ciascuno di questi pacchetti ha un proprio formato che predefinisce la suddivisione in sottocampi e all'interno di essi il significato di ogni bit/byte che occupa una certa posizione. Questo permette di aumentare l'efficienza del protocollo evitando il trasporto di informazione non necessaria, quali etichette delle grandezze e unità di misura. Il destinatario deve poter distinguere il tipo di pacchetto ricevuto per poterlo decodificare secondo il relativo formato.

7.1.1 Pacchetti in uplink

In uplink sono previste due categorie di formati di pacchetto.

1) backdoorCommand

I backdoorCommand sono dei comandi che consentono di eseguire un certo numero di funzioni di base direttamente al microprocessore dell'OBRF. La loro utilità si manifesta quando si necessita di operare procedure di ripristino o di gestione di alcune

contromisure in caso di malfunzionamenti nel resto del sistema. Il formato dei backdoorCommand è rappresentato in Figura 7.4:

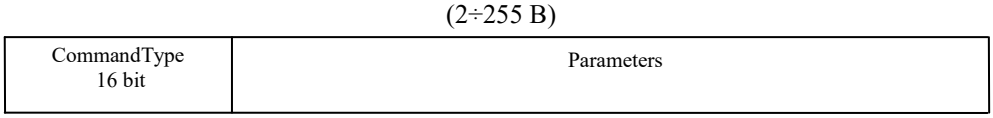


Figura 7.4. Formato dei backdoorCommand

`CommandType`: il primo bit è uguale a 0b0 in modo da indicare all'OBRF che si tratta di un backdoorCommand. In quanto tale, l'OBRF non lo inoltra in uscita ma lo esegue in locale. I restanti bit costituiscono il codice che identifica univocamente uno specifico comando all'interno del set dei comandi supportati dall'OBRF. Mediante una tabella di corrispondenza, l'MCU dell'OBRF può identificare le operazioni che devono essere eseguite quando si riceve un tale comando e le periferiche di competenza.

`Parameters`: è il campo in cui vengono specificati i valori degli eventuali parametri che possono accompagnare il comando.

2) ordinaryCommand

Agli ordinaryCommand appartengono la generalità dei comandi, cioè i LONG e i SHORT eseguiti dall'OBC. La loro struttura in generale è formata da due campi (Figura 7.5):

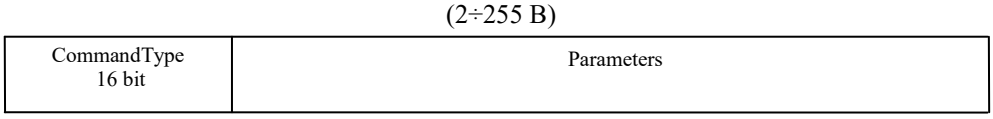


Figura 7.5. Formato di un ordinaryCommand

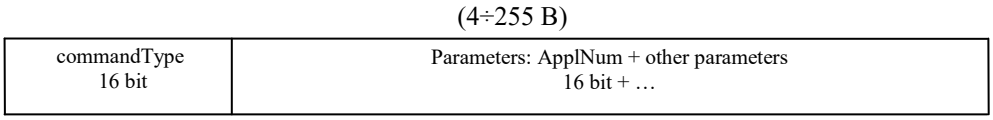


Figura 7.6. Particolarizzazione di un ordinaryCommand quando contiene un LONG

`CommandType`: il primo bit è uguale 0b1 in modo da indicare all'OBRF che si tratta di un ordinaryCommand e in quanto tale deve essere inoltrato in uscita affinché

possa essere acquisito dal Telemetry/Command Processor. I restanti bit costituiscono il codice che identifica univocamente il comando all'interno del set dei comandi supportati dal satellite. Mediante una tabella di corrispondenza il Telemetry/Command Processor potrà identificare le operazioni che devono essere eseguite quando si riceve un tale codice e le periferiche di competenza. Nella Tabella 7.1 è riportato un esempio di codifica per alcuni comandi (tenendo conto del primo bit a 1).

Tipo di ordinaryCommand	COMMAND_code
`GetData`	`0x8000`
`GetFrag`	`0x8001`
`BasicTelemetry`	`0x8002`
`BatteryLifeStatus`	`0x8003`
`Orientation`	`0x8004`
`Etc…`	…

Tabella 7.1 Possibili codifiche per il campo `CommandType`

`Parameters`: è il campo in cui vengono specificati i valori degli eventuali parametri che possono accompagnare un comando. Per i comandi LONG il primo valore che viene inserito è sempre l'Application Number, un numero che verrà utilizzato per assegnare un nome ai dati generati in seguito all'esecuzione di tale LONG, come si vedrà necessario per potersi a questi riferire in una successiva richiesta (Figura 7.6), mentre i valori che gli seguono sono specifici per il comando. Un comando di `Orientation`, ad esempio, potrà avere come parametri l'Application Number seguiti da un argomento che indica un valore in gradi. Per i comandi GetData e GetFrag, che come vedremo in seguito sono i comandi mediante cui si fa richiesta di un dato o di un frammento di dato, il primo valore che viene inserito è l'Application Number del precedente comando LONG di cui si stanno richiedendo i dati e nel caso di GetFrag seguiranno i numeri di frammento che si desidera ricevere con la prossima trasmissione (Figura 7.7 e Figura 7.8).

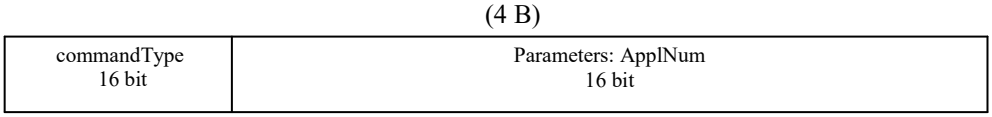


Figura 7.7. Particolarizzazione di un ordinaryCommand quando contiene un GetData

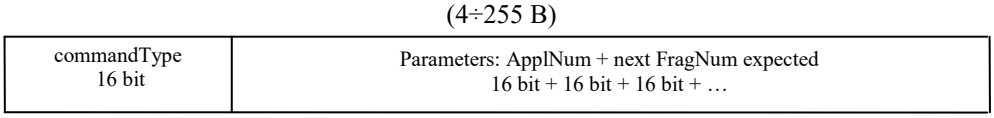


Figura 7.8 Particolarizzazione di un ordinaryCommand quando contiene un GetFrag

Data la criticità dei comandi, un cui errore potrebbe compromettere l'esito stesso della missione, oltre il controllo del CRC del frame AX.25, è previsto come ulteriore meccanismo di integrità dei dati, una triplice ripetizione dell'ordinaryCommand all'interno del campo Info. Forme più complesse di codifica sono valutabili.

7.1.2 Pacchetti in downlink

Anche in downlink, sono previste due categorie di formati di pacchetto.

1) pureNACK

È l'unico caso di messaggio che viene generato in maniera del tutto autonoma dall'OBRF e si ha quando viene ricevuto un comando che non supera il controllo del CRC, ma risulta correttamente indirizzato al satellite. Il formato dei pureNACK è costituito dal solo campo messageType (Figura 7.9). Il suo primo bit è posto a 0b0 per distinguere questo NACK generato dall'OBRF, quindi a livello datalink, da qualsiasi altro tipo di messaggio di livello superiore generato dall'OBC (ordinaryDownlink).

Figura 7.9. Formato per i pureNACK e tutti i messaggi che non contengono dati

La Tabella 7.2 illustra il comportamento dell'OBRF in base alla validità dell'indirizzo di destinazione e del CRC. Questo comportamento è stato definito in modo da evitare che frame provenienti da altri utilizzatori che arrivano incidentalmente al satellite possano eccitare l'OBRF, ma allo stesso tempo permettere al Mission Controller di sapere quando la trasmissione di un comando fallisce per via del controllo sul CRC non superato e quindi di anticipare la ritrasmissione senza dover attendere lo scadere del time-out.

Indirizzo destinazione	CRC	Comportamento
NO	NO	Ignora
NO	SI	Ignora
SI	NO	Invia a Terra il pureNACK
SI	SI	Inoltra il comando decapsulato al TelemetryCommandProcessor

Tabella 7.2. Comportamento dell'OBRF in base alla validità dell'indirizzo di destinazione e al CRC

2) ordinaryDownlink

A questa categoria appartiene la generalità dei messaggi generati a livello applicazione come risposta ai comandi e il segnale di beacon. Essi sono passati dall'OBC all'OBRF in maniera trasparente. La sua struttura in generale è formata da due campi (Figura 7.10):

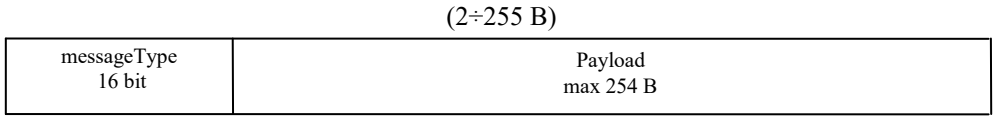


Figura 7.10. Formato di un ordinaryDownlink

`messageType`: il primo bit è uguale a 0b1 per distinguere questo messaggio da un pureNACK generato a livello datalink, mentre restanti bit costituiscono il codice che identifica univocamente il messaggio all'interno di un predefinito set. Nella Tabella 7.3 è riportato un esempio di codifica per i principali tipi di messaggio (tenendo conto del primo bit a 1).

Tipo di ordinaryDownlink	messageType_code
BEACON	0x8000
ACK_DATA	0x8001
ACK_FRAG	0x8002
CMD_RECEIVED	0x8003
CMD_DUPLICATED	0x8004
DATA_NRDY	0x8005
BAD_CMD	0x8006
CMD_NOT_EXE	0x8007
PER_ERR	0x8008
MEMORY_FULL	0x8009
UNKNOWN_COMMAND_NUM	0x8010

Tabella 7.3. Possibili codifiche per il campo MessageType

`Payload`: è il campo che ospita gli eventuali dati che possono accompagnare il messaggio, come verrà di seguito illustrato presentando i tipi principali tipi di messaggio.

BEACON

È un pacchetto che viene trasmesso ad intervalli regolari (ad esempio ogni 30 secondi) al fine di comunicare lo stato di funzionamento del satellite e facilitare le operazioni di tracking. Nel campo Payload trasporta una forma base di telemetria (Figura 7.10).

ACK_DATA

È il pacchetto che si ha come risposta ai comandi SHORT, e in particolare ad un comando GetData. È un messaggio di acknowledgement che contestualmente trasporta nel campo Payload il dato prodotto dalla esecuzione del comando (Figura 7.11).

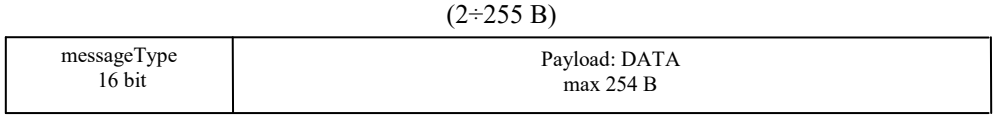


Figura 7.11. Particolarizzazione di un ordinaryCommand quando contiene un ACK_DATA

ACK_FRAG

È il pacchetto che si ha come risposta ad un comando GetFrag. È simile al precedente, ma i primi due byte del campo Parameters vengono utilizzati per indicare il numero di frammento mentre i restanti trasportano il frammento stesso del dato complessivo richiesto (Figura 7.12).

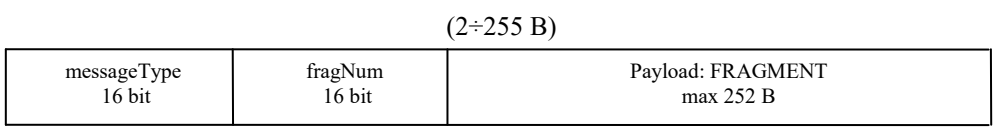


Figura 7.12. Particolarizzazione di un ordinaryCommand quando contiene un ACK_FRAG

Tutti i tipi di messaggio che seguono condividono lo stesso formato (Figura 7.13) in quanto necessitano del solo campo messageType.

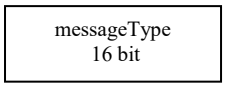


Figura 7.13. Formato per tutti i messaggi che non trasportano dati

CMD_RECEIVED

È il messaggio con cui il TCP fornisce al Mission Controller l'acknowledgment di avvenuta acquisizione di un comando lungo.

CMD_DUPLICATED

È il messaggio con cui il TCP fornisce al Mission Controller l'acknowledgment di avvenuta acquisizione di una copia di un comando lungo.

DATA_NRDY

È il messaggio con cui il TCP, in seguito alla ricezione di un GetData o GetFrag, comunica al Mission Controller che i dati richiesti non sono ancora pronti in memoria. Questa situazione si verifica normalmente quando l'esecuzione di un comando lungo non è ancora terminata.

BAD_CMD

È il messaggio con cui il TCP comunica al Mission Controller di aver ricevuto un comando non riconosciuto all'interno del set dei comandi supportati.

CMD_NOT_EXE

È il messaggio con cui il TCP comunica al Mission Controller di aver ricevuto un comando non eseguibile al momento.

PER_ERR

È il messaggio con cui il TCP comunica al Mission Controller di aver ricevuto un comando che fa riferimento ad una periferica che non risponde.

MEMORY_FULL

È il messaggio con cui il TCP comunica al Mission Controller di aver ricevuto un comando il cui dato non può essere memorizzato in memoria poiché questa è piena.

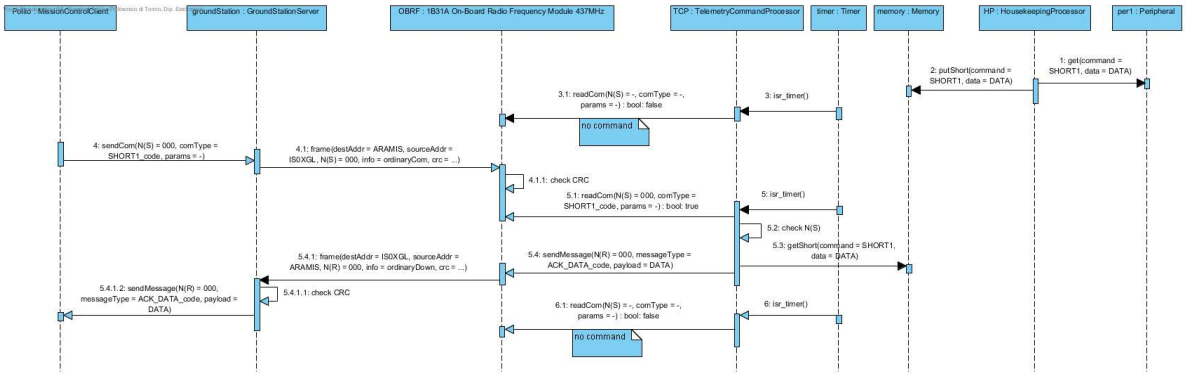
UNKNOWN_CMD_NUM

È il messaggio con cui il TCP comunica al Mission Controller di aver ricevuto un GetData o un GetFrag il cui ApplicationNumber non fa riferimento a nessun comando LONG precedentemente ricevuto.

7.2 Funzionamento del protocollo

Illustriamo il funzionamento del protocollo per mezzo di digrammi di sequenza che rappresentano la successione temporale degli eventi prima nel caso della gestione di un comando SHORT, poi nel caso della gestione di un comando LONG, entrambi nell'ipotesi di trasmissione in assenza di errori. Successivamente sono riconsiderate le stesse sequenze nell'ipotesi di presenza di errori. Infine è mostrato un diagramma che rappresenta la situazione in cui più comandi LONG e SHORT sono combinati assieme, situazione rappresentativa del comune esercizio del protocollo. I paragrafi seguenti mostrano volutamente l'aspetto grafico originale derivato dalla funzione di generazione di report del software *Visual Paradigm* usato dal gruppo di lavoro di AraMiS per descrivere in UML l'intero progetto.

Basic Protocol - Short command

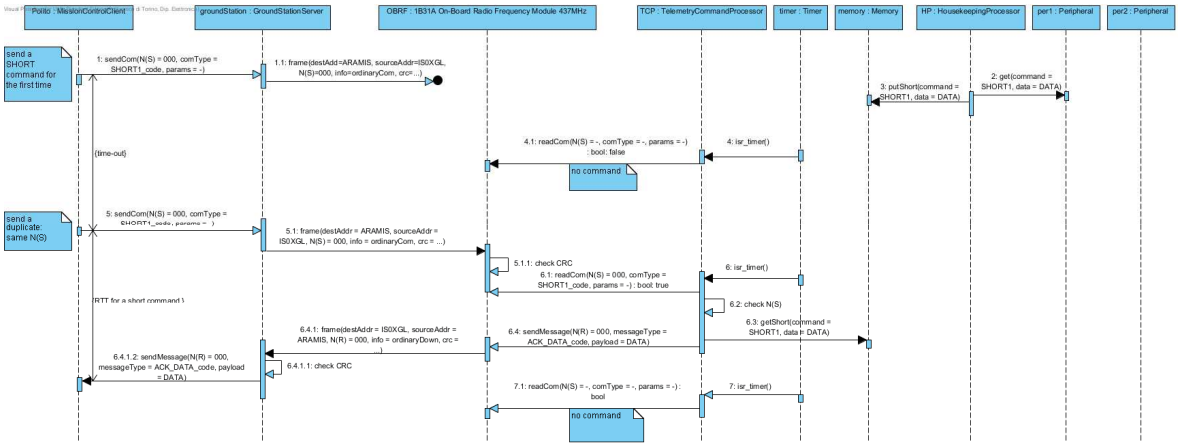


Documentation

La sequenza mostra la successione degli eventi relativi ad un comando breve in assenza di errori. La funzione (1. get) rappresenta l'Housekeeping Processor mentre effettua una delle interrogazioni periodiche dell'uscita di vari sensori distribuiti nel satellite al fine di raccogliere gli *housekeeping data* e la 2. putShort mentre li inserisce in memoria. In un generico istante si suppone che il Mission Controller decide di richiedere, ad esempio, i dati di Housekeeping Telemetry. Mediante il messaggio 4. sendCom, tramite Internet invia alla ground station il comando. Quando la ground station lo riceve, compone il frame AX.25 inserendo nei vari campi i dovuti argomenti e in particolare: ARAMIS nell'indirizzo destinatario, il proprio CALLSIGN nell'indirizzo mittente, il numero di sequenza nel campo N(S) e il comando nel campo Info. Il formato dei comandi sarà descritto in seguito. La ground station quindi trasmette il frame sul canale radio in direzione del satellite (4.1 frame). Il frame viene ricevuto dal modulo OBRF che ne controlla il CRC (4.1.1 check CRC) e l'indirizzo destinatario. Supponendo che il CRC e l'indirizzo destinatario siano corretti, il numero di sequenza e il comando vengono decapsulati e resi disponibili in uscita affinché possano essere letti dal Command Processor implementato sull'OBC, mediante la funzione 5.1 readCom. Bisogna qui precisare che la connessione tra l'OBRF e il TCP è basata su un polling loop essendo l'OBRF un oggetto hardware e il TCP un oggetto software. Il TPC, quindi, esegue letture periodiche (readCom) dell'uscita dell'OBRF, inizializzate da un timer. Una volta che il TCP ha acquisito il contenuto in uscita dall'OBRF, controlla il numero di sequenza (5.2 check N(S)) e poi interpreta il comando secondo una propria

tabella. Trovando che si tratta di un comando SHORT, l'esecuzione, a prescindere che sia ricevuto per la prima volta o no, consiste nel semplice recupero dei dati richiesti dalla memoria (5.3 getShort), e nella generazione di un pacchetto che ha il duplice scopo di acknowledgement della ricezione del frame proveniente da Terra e di trasporto dei dati richiesti. Questo messaggio viene passato all'OBRF (5.4 sendData) il quale poi lo trasmette in downlink con una procedura simmetrica a quella usata dalla ground station per l'uplink.

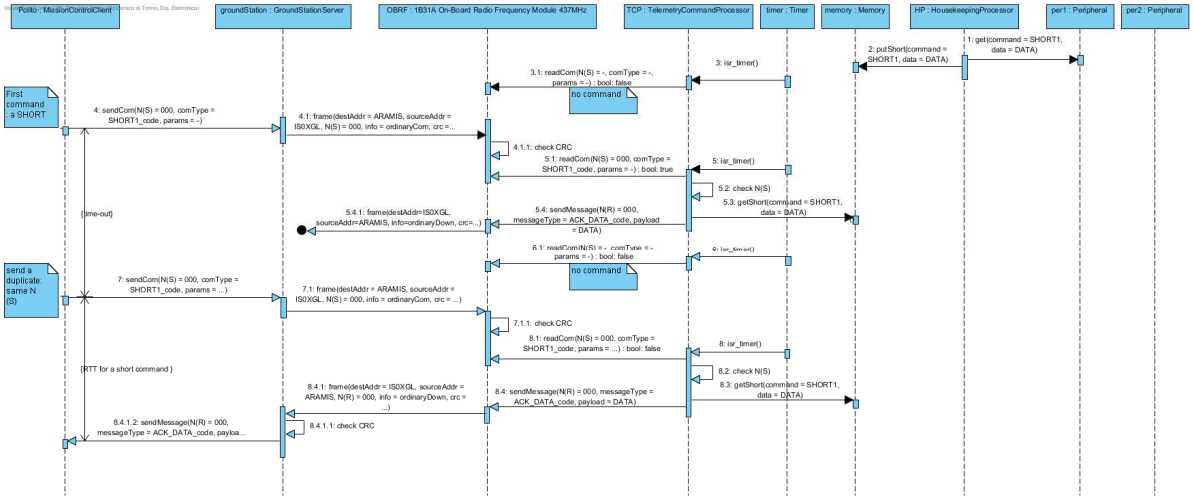
Basic Protocol - Short command with loss in uplink



Documentation

In figura è rappresentato il caso in cui si perde il frame in uplink contenente il comando breve (1.1 frame). In questa situazione il Mission Controller non riceverà mai l'ack con i dati (ACK_DATA), per cui trascorso un tempo pari a quello di time-out invia di nuovo lo stesso comando con lo stesso numero di sequenza (5. sendCom). Nell'esempio si suppone che questo viene ricevuto e tutto continua a svolgersi come descritto nel caso ideale.

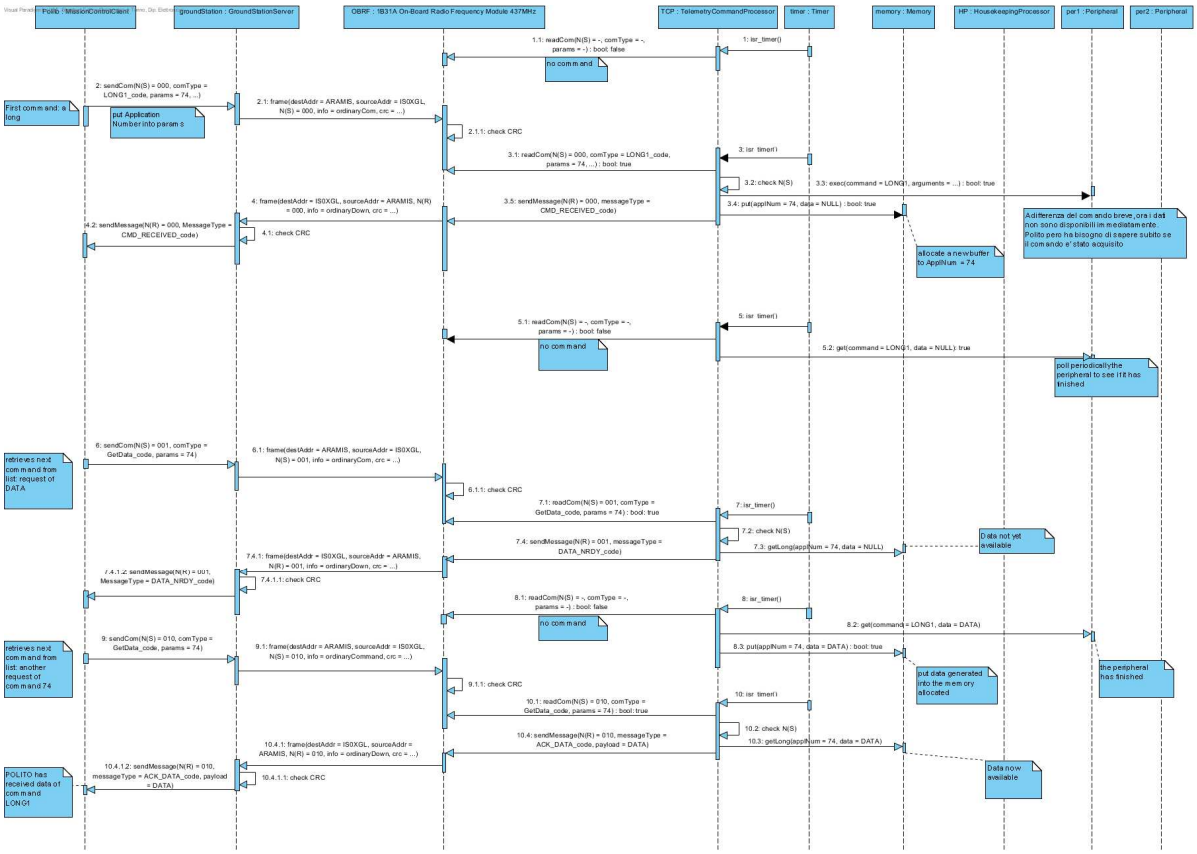
Basic Protocol - Short command with loss in downlink



Documentation

In figura è rappresentato il caso in cui si perde l'ACK_DATA in downlink (5.4.1 frame). Rispetto al caso in cui lo si perde in uplink, ora il satellite ha ricevuto ed eseguito il comando. Dal punto di vista del Mission Controller però la situazione è la stessa del caso precedente: trascorso un tempo pari a quello di time-out senza che abbia ricevuto l'ACK_DATA, la Mission Controller ritrasmette lo stesso identico frame contenente il comando (7. sendCom). Quando questo raggiunge il satellite, il TCP dal controllo del numero di sequenza N(S) è in grado di stabilire che si tratta di una copia del comando precedente. Poiché si tratta di un comando SHORT, cioè un comando che richiede solo dati assieme ad un ack, di fatto viene rieseguito identicamente (da 8.3 getShort fino al termine).

Basic Protocol - Long command with subsequent request command



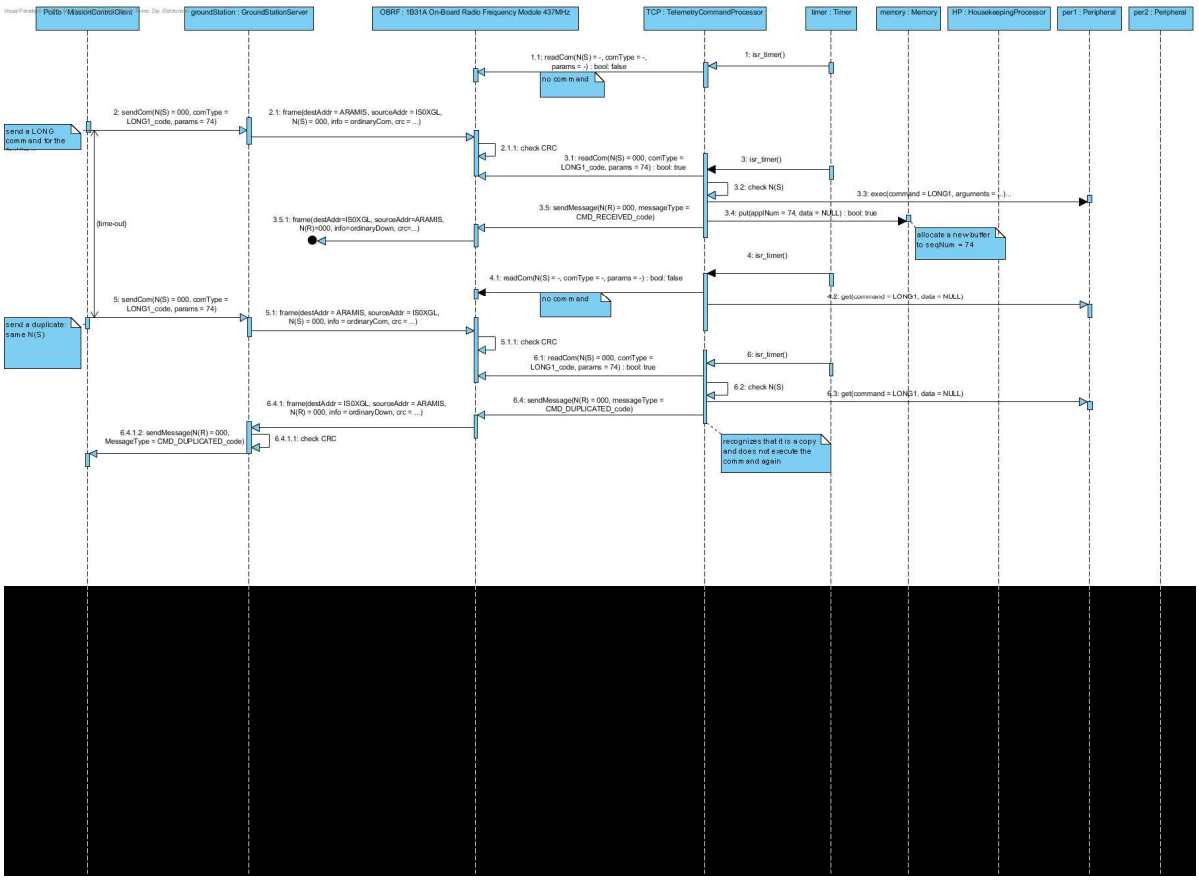
Documentation

La sequenza mostra la successione degli eventi relativi ad un comando LONG in assenza di errori. In un generico istante si suppone che il Mission Controller decide di richiedere, ad esempio, l'acquisizione di una immagine da eseguire un'ora dopo. Mediante il messaggio 2. sendCom, come nel caso precedente, invia alla ground station il comando accompagnato ora da un parametro, l'Application Number, che è un numero sequenziale che identifica ogni comando trasmesso e che servirà per poter riferirsi, in un successivo comando di richiesta, ai dati generati dalla sua esecuzione. Il comando viene trasportato fino al TCP nello stesso modo visto per il comando SHORT. Il TCP controlla il numero di sequenza 3.2 check N(S) per capire se il comando è ricevuto per la prima volta o meno e poi interpreta il comando secondo una propria tabella. Supponendo che sia la prima volta che viene ricevuto, il TCP comunica con la periferica di competenza per la sua esecuzione (3.3 exec) e immediatamente allocca dello spazio in memoria per ospitare i dati che verranno

generati al termine della sua esecuzione (3.4 put). Per i comandi LONG il Mission Controller necessita comunque di sapere subito se sono stati ricevuti. A tal fine, il TCP, alla ricezione di un comando LONG, genera immediatamente un acknowledge puro, cioè contenente il messaggio "command received" da trasmettere in downlink (da 3.5 sendMessage fino a 4.2 sendMessage). Da questo momento in poi, la periferica del satellite interessata dal precedente comando entra in una lista di periferiche che il TCP interrogherà periodicamente allo scopo di valutarne il termine dell'esecuzione e di acquisire il dato. La funzione 5.2 get mostra una di queste interrogazioni per la quale l'esecuzione risulta non ancora terminata. Dopo un tempo considerato ragionevole (in relazione al comando LONG precedentemente inviato), il Mission Controller può decidere di tentare il recupero dei dati. Genera quindi il comando di richiesta GetData accompagnato come parametro dall'Application Number del comando di cui si vogliono recuperare i dati e lo invia alla ground station (6. sendCom). Il comando raggiunge il TCP dove viene interpretato e la sua esecuzione consiste nella interrogazione della memoria (7.3 get). Alla allocazione in memoria appartenente all'Application Number specificato, però, non risulta ancora disponibile alcun dato, cosicché la funzione ritorna un NULL. Il TCP allora genera un pacchetto contenente il messaggio "data not ready" che viene mandato in downlink (da 7.4 sendMessage a 7.4.1.2 sendMessage).

La funzione 8.2 get mostra una ulteriore iterazione della interrogazione della periferica da parte del TCP per la quale, finalmente, l'esecuzione risulta terminata e il dato restituito e posto in memoria (8.3 put). In un istante successivo, il Mission Controller decide di tentare nuovamente il recupero dei dati, per cui invia nuovamente il comando GetData (9. GetData). Questa volta l'interrogazione alla memoria che ne segue (10.3 get) ritorna il dato e il TCP può generare un pacchetto ACK_DATA che passa all'OBRF (10.4 sendData) affinché venga trasmesso in downlink. Concludiamo osservando che il comando getData, di fatto, è un comando SHORT.

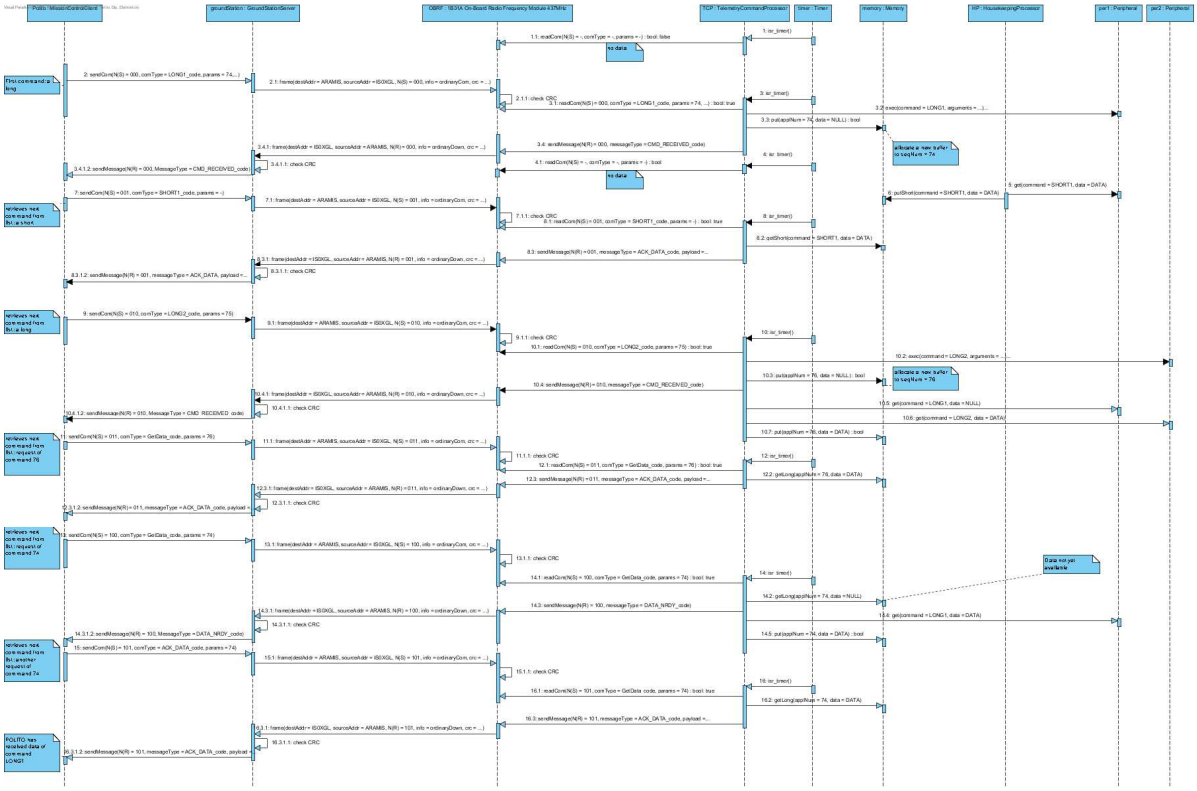
Basic Protocol - Long command with ACK loss in downlink



Documentation

Per quanto riguarda la perdita in uplink di un comando LONG si rimanda al caso di comando SHORT. Questo diagramma illustra invece il caso di perdita dell'ack puro che segue in downlink alla ricezione di un comando LONG. Si supponga quindi che il satellite riceva un comando lungo e attivi l'esecuzione, l'allocazione in memoria e la trasmissione in downlink del messaggio CMD_RECEIVED secondo il funzionamento visto illustrando il caso ideale. Per ipotesi il 3.5.1 frame risulta però perso. Trascorso un tempo pari a quello di time-out il Mission Controller invia di nuovo lo stesso comando e con lo stesso numero di sequenza (5.sendCom). Quando questo raggiunge il satellite, il TCP, dal controllo del numero di sequenza 6.2 check N(S), è in grado di stabilire che si tratta di una copia del comando precedente e non lo riesegue. Genera, invece, un nuovo acknowledge contenente il messaggio CMD_DUPLICATED. Questo viene trasmesso in downlink (6.4 sendMessage) e si suppone venga ricevuto dal Mission Controller, il quale terminerà così l'invio di quel frame avendo accertato che il comando è stato acquisito.

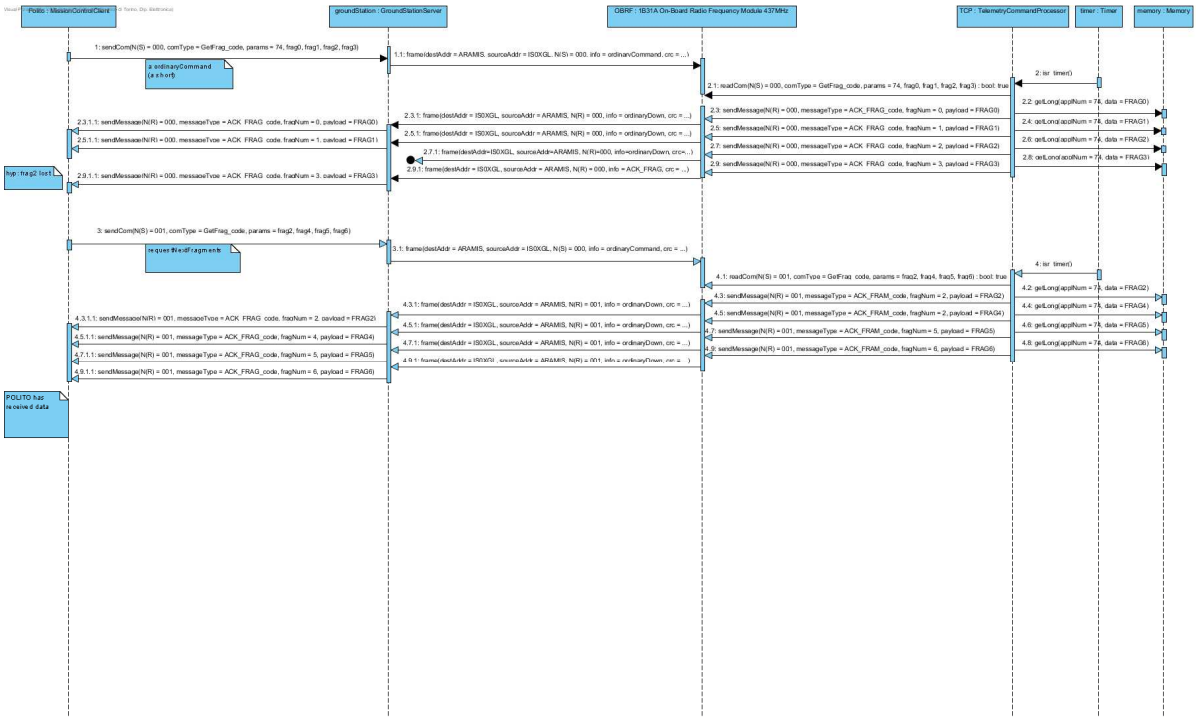
Basic Protocol - Combined operations



Documentation

La figura mostra un segmento temporale in cui comandi LONG si alternano a comandi SHORT, come dovrebbe essere durante il reale funzionamento del protocollo. È significativo osservare come tra un comando LONG e la relativa richiesta di dati possono essere trasmessi nel frattempo altri comandi, sia SHORT che LONG. Inoltre può accadere che il dato generato da un LONG più recente sia richiesto prima di quello generato da un LONG meno recente.

Basic Protocol - Download with Fragmentation



Documentation

Il comando GetData è idoneo a richiedere la trasmissione di un dato generato da un comando lungo la cui dimensione non necessita più di un frame. In caso contrario il dato richiesto necessita di essere frammentato in più frame. Per questi dati è stato definito il comando GetFrag. Il GetFrag, a differenza del GetData, fa seguire all' Application Number nel campo Params, anche l'elenco dei frammenti (Fragment Number) che si desidera ricevere consecutivamente con la prossima trasmissione. Nell'esempio in figura, con il messaggio 1 sendCom il Mission Controller richiede i primi quattro frammenti del dato generato dall'esecuzione del comando 74. In 2.2 get il TCP accede alla memoria per prelevare il primo frammento, genera un pacchetto ACK_FRAM e lo passa all'OBRF (2.3 sendFrag) affinché venga trasmesso in downlink. Questi passaggi vengono ripetuti autonomamente per gli altri tre frammenti. Nell'esempio in figura si ipotizza che il frame contenente il frammento 2 venga perso. Nella successiva richiesta 3.sendCom il comando GetFrag specificherà quindi di nuovo il frammento 3, più altri tre frammenti nuovi. Le trasmissioni dei comandi GetFrag sono iterabili fino alla completa ricezione del dato. Tipicamente la dimensione di un dato richiesto, e di conseguenza il numero di

frammenti necessari, è nota a Terra, quindi la comunicazione può terminare con la richiesta dell'ultimo frammento necessario a completare il download.

7.3 Funzioni definite

Il seguente elenco riassume le funzioni che sono state definite per il funzionamento del protocollo e del sistema.

1) sendCom (N(S), comType, params): è il messaggio con cui il MCC invia tramite Internet un comando alla GSS da trasmettere verso il satellite. I suoi argomenti sono:

- N(S): numero di sequenza del frame inviato (3 bit)
- comType: codice identificativo del comando (2 byte)
- Parameters: consiste in argomenti specificatori del comando inviato.

2) frame (destAddr, sourceAddr, N(S), Info): rappresenta la trasmissione su canale radio del frame in uplink e in downlink. I suoi argomenti sono i campi del frame AX.25:

- destAddr e sourceAddr: sono il callSing della stazione radio destinazione e mittente
- N(S): il numero si sequenza del frame trasmesso
- Info: il campo di dimensione massima pari a 255 byte che contiene il pacchetto.

3) sendMessage (N(R), messageType, payload): quando riferita al segmento di Terra, rappresenta il messaggio con cui la GSS invia tramite Internet il contenuto del frame ricevuto dal satellite alla MCC. I suoi argomenti sono:

- N(R): numero di sequenza dell'ultimo frame ricevuto dal satellite (3 bit)
- MessageType: codice identificativo del messaggio
- Payload: contiene il dato nel caso di ACK_DATA o un frammento nel caso di ACK_FRAG.

4) readCom (N(S), comType, params): rappresenta l'operazione di poll da parte del TCP sull'uscita dell'OBRF. Se nessun comando è presente ritorna false, in caso contrario ritorna true e gli argomenti:

- numero di sequenza N(S)
- comType
- params: eventuali argomenti specificatori del comando.

Le seguenti funzioni sono implicate durante un comando SHORT di richiesta telemetria.

5) get (command, data): rappresenta l'HP che esegue interrogazioni periodiche della generica Periferica al fine di raccogliere gli housekeeping data. I suoi argomenti sono:

- command: il codice che richiede una telemetria specifica
- data: i dati raccolti.

6) put (command, data): rappresenta l'HP che inserisce in memoria i dati di telemetria raccolti.

7) getShort: rappresenta il TCP che dopo aver ricevuto un comando SHORT di richiesta di dati di telemetria, fa una lettura della memoria.

Le seguenti funzioni sono implicate durante un comando LONG:

8) exec (command, arguments): rappresenta il TCP che dopo aver ricevuto un comando LONG, comunica con la periferica di competenza per istruirla. I suoi argomenti sono

- command: le istruzioni per la periferica
- arguments: eventuali dati che specificano il comando.

9) put (applNum, data): rappresenta il TCP che, dopo una exec, allocca dello spazio in memoria per ospitare i dati che verranno generati al termine della esecuzione di un comando LONG. Torna true se c'è spazio in memoria, false se la memoria è piena. Successivamente rappresenta il TCP che, dopo una get che rende il dato, inserisce il dato in memoria. I suoi argomenti sono:

- applNum: identifica il comando e il dato generato quando inserito in memoria
- data: è il dato generato in seguito all'esecuzione del comando.

10) get (command, data): rappresenta il TCP che interroga periodicamente le periferiche da cui attende un dato al termine dell'esecuzione di un LONG. Torna NULL se il dato non è presente, altrimenti torna il dato DATA. I suoi argomenti sono:

- command: specifica alla periferica che tipo di dato si attende
- data: contiene il dato ritornato

11) getLong (applLong, data): rappresenta il TCP che dopo aver ricevuto un comando SHORT di richiesta di dati di un precedente comando LONG, fa una lettura della

memoria. Torna NULL se il dato non è presente, altrimenti torna il dato DATA. I suoi argomenti sono:
- applNum: identifica il dato di cui si richiede il recupero dalla memoria
- data: è il dato recuperato.

7.4 Cenni sulla gestione della memoria

La memoria viene utilizzata per contenere gli housekeeping data e i mission data. Il criterio con cui la memoria viene liberata è differenziato a seconda che si tratti dei primi o dei secondi.

Per gli housekeeping data, la gestione consiste semplicemente nel rimpiazzare i dati precedentemente memorizzati con i rispettivi nuovi (refresh). Questo ci garantisce di avere una quantità di memoria da loro occupata costante e prestabilita, oltre che soddisfare l'esigenza di ottenere in downlink sempre la loro versione più aggiornata.

I mission data sono generati al termine dell'esecuzione di comandi LONG. Ricordiamo che ad ogni loro acquisizione da parte del TCP viene allocata della memoria e che questa viene riempita con i dati generati al termine della loro esecuzione. Una volta che la memoria viene saturata, l'arrivo di un nuovo comando LONG porterà alla cancellazione del dato avente Application Number inferiore dando la precedenza prima tra quelli letti almeno una volta e poi se necessario tra quelli mai letti (che di conseguenza verranno persi). Questo meccanismo richiede che il programma che gestisce la memoria si occupi di settare un flag sui dati letti. Quando l'arrivo di un nuovo comando porta alla cancellazione di un dato mai letto, tale condizione verrà segnalata a Terra con un apposito messaggio MEMORY_FULL.

Capitolo 8

Analisi delle prestazioni

In questo capitolo viene calcolato il goodput, cioè la quantità di dati utili trasferiti nell'unità di tempo, in relazione ad un comando SHORT per il recupero di un dato di dimensione massima su un singolo frame.

8.1 RDT per i comandi SHORT

Definiamo il *Request-Delivery Time* (RDT) come il tempo che intercorre tra l'istante in cui dal Mission Controller viene trasmesso il primo bit del comando e l'instante in cui il Mission Controller riceve l'ultimo bit della risposta.

Il RDT per i comandi SHORT è il tempo che intercorre tra l'istante in cui viene trasmesso il primo bit del comando in uplink e l'istante in cui viene ricevuto l'ultimo bit dell'ACK_DATA in downlink.

La Figura 8.1 rappresenta la sequenza già vista relativa all'invio di un comando SHORT e alla ricezione della sua risposta, l'ACK_DATA, dove però sono state evidenziate le durate delle diverse operazioni. In particolare si ha:

$T_{Internet_up}$ e $T_{Internet_down}$: sono rispettivamente il tempo impiegato dal comando inviato dal Mission Controller a raggiungere via Internet la ground station remota e il tempo impiegato dal dato inviato dalla ground station a raggiungere via Internet il Mission Controller. Il loro valore è fortemente variabile a seconda dallo stato della rete tra il Mission Controller e la ground station. Possiamo assumere che la loro somma sia dello stesso ordine di grandezza del round trip time ricavato eseguendo un test di ping verso un server remoto, cioè 50÷200 ms.

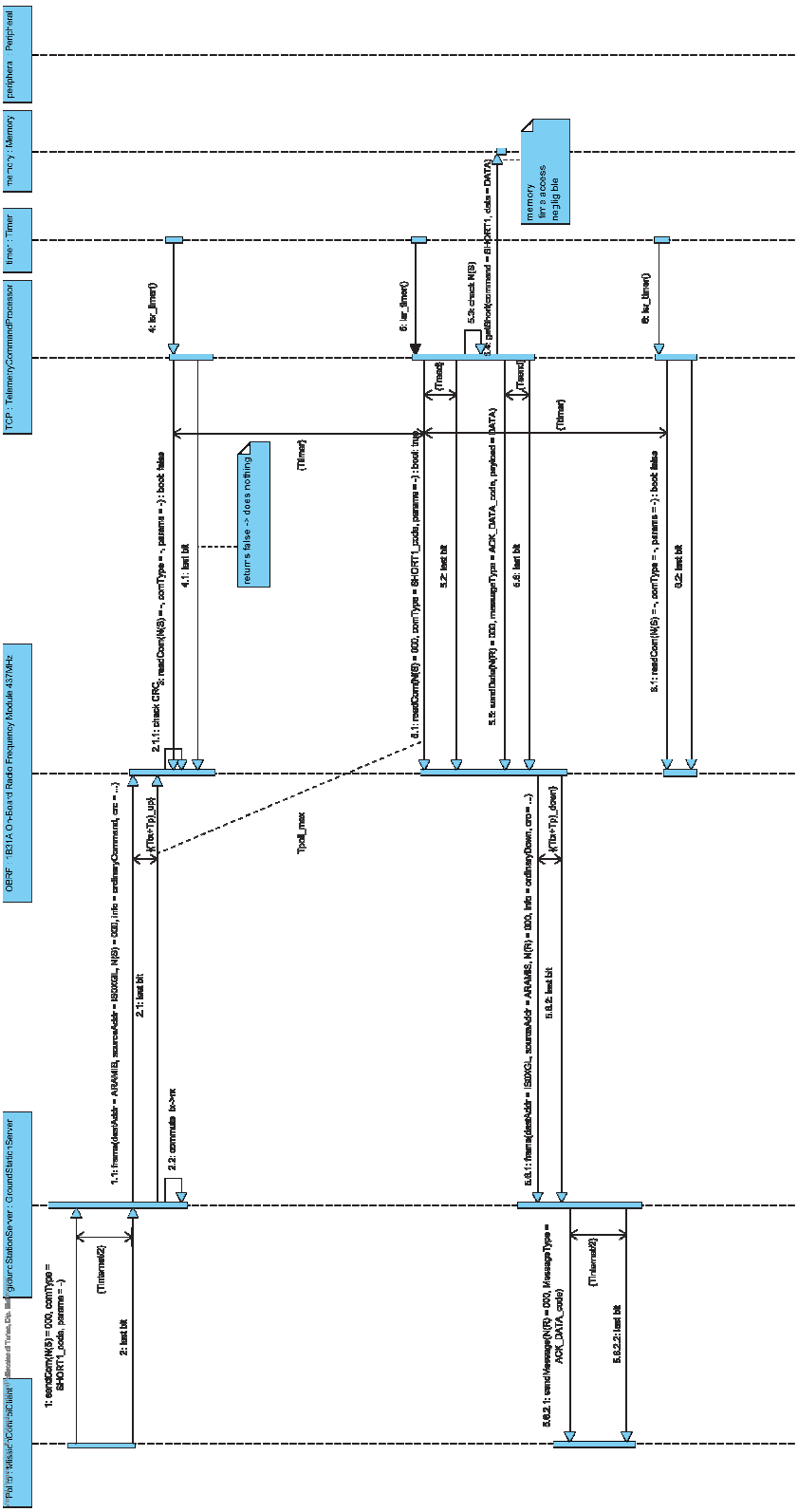


Figura 8.1. Esplicitazione dei tempi nel processo di invio di un comando SHORT

$\left(T_{tx}+T_p\right)_{up}$ e $\left(T_{tx}+T_p\right)_{down}$: sono rispettivamente il tempo impiegato dal frame trasmesso dalla ground station a raggiungere via radio il modulo OBRF del satellite e il tempo impiegato dal frame trasmesso dall'OBRF per raggiungere via radio la ground station.
Più in particolare si ha:

$$T_{tx}=\frac{L_{sync}+L_{header}+L_{payload}}{R_b} \tag{8.1}$$

e

$$T_p=\frac{d}{c} \tag{8.2}$$

T_{poll} tempo che può intercorrere tra l'istante in cui il comando è pronto all'uscita dell'OBRF e l'istante in cui inizia il poll da parte del TCP. Esso va considerato nel caso peggiore, cioè quello in cui il ciclo di lettura inizia un tempo infinitesimale prima che il comando sia disponibile. In questo caso la lettura effettiva slitta al successivo periodo del timer, per cui $T_{poll_MAX}=T_{timer}-\varepsilon$.

T_{read} e T_{send} : sono rispettivamente il tempo impiegato per la lettura da parte del TCP del numero di sequenza N(S) e del comando contenuto nel campo Info e il tempo impiegato per la scrittura da parte del TCP del numero di sequenza N(R) e del dato da incapsulare nel campo Info da trasmettere come risposta.

$$T_{read}=\frac{D_{com}}{R_{b_\mathrm{int}}} \tag{8.3}$$

$$T_{send}=\frac{D_{msg}}{R_{b_\mathrm{int}}} \tag{8.4}$$

Il bit-rate del collegamento tra OBRF e TCP dipende dal tipo di interfaccia interna usata nella realizzazione finale. Se questa è, ad esempio, una interfaccia seriale RS-232 allora $R_{b_\mathrm{int}}\le 115.2\,k\mathrm{bps}$.

Si sono considerati trascurabili i tempi di esecuzione delle routine firmware all'OBRF e dell'esecuzione software al TCP che coincide sostanzialmente con il tempo di accesso alla memoria.

Il bit-rate R_{b_int} e il periodo del timer T_{timer}, a differenza delle altre grandezze che sono imposte dal sistema, devono essere dimensionate in modo da migliorare il RDT. Osserviamo che deve essere soddisfatta la condizione:

$$T_{timer} > \left(T_{read} + T_{send}\right)_{MAX} \tag{8.5}$$

Il tempo di timer, cioè, non può essere inferiore alla somma del tempo di lettura del comando più il tempo di scrittura del messaggio di risposta, considerati nel caso questi siano di dimensione massima, cioè 255B.

È ragionevole considerare di poter utilizzare una interfaccia tra OBRF e TCP avente $R_{b_\text{int}} = 100\text{kbps}$. In tal caso $T_{timer} > 0.041s$.

Il Request-Delivery Time è quindi dato dalla somma di tutti questi tempi:

$$RDT = T_{Internet_up} + \left(T_{tx} + T_p\right)_{up} + T_{timer} + T_{read} + T_{send} + \left(T_{tx} + T_p\right)_{down} + T_{Internet_down} \tag{8.6}$$

Il diagramma in Figura 8.2 mostra il RDT per un comando SHORT e un messaggio ACK_DATA, entrambi a dimensione massima, in funzione di $R_{b_\text{int}} = 70 \div 150\,\text{kbps}$ e $T_{timer} = 0.06 \div 0.11s$.

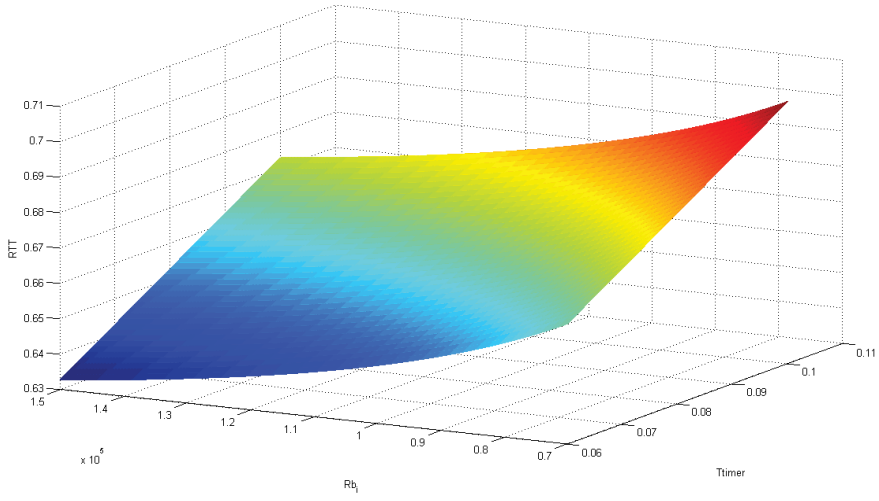


Figura 8.2. RDT in funzione di R_{b_int} e T_{timer} nel caso di frame a dimensione massima

Per avere una stima del RDT del presente protocollo assumiamo dunque $R_{b_\text{int}} = 100\text{kbps}$ e $T_{timer} = 0.06\,s$ per tenere un piccolo margine avendo trascurato il tempo di elaborazione del TCP. Inoltre fissiamo i valori tipici:

$T_{Internet_up} = T_{Internet_down} = 0.05\,s$

$L_{sync} + L_{header} + L_{payload} = 20B + 18B + 255B = 293B$

$d = 800\,Km$

$D_{com} = D_{msg} = 255B$

si ottiene $RDT = 0.656\,s$

Considerando l'espressione del goodput:

$$R_B^{'} = \frac{L_{payload}}{RDT}$$

Si ottiene la stima $R_{B\,up}^{'} = R_{B\,down}^{'} = 3.kbps$ a fronte di un data-rate sul canale radio di 9600 bps.

Capitolo 9

Conclusioni

La tesi svolta ha voluto dimostrare come nell'ambito di un satellite partecipante al progetto GENSO sia possibile utilizzare il livello data-link del protocollo AX.25, richiesto per soddisfare la compatibilità con le stazioni radioamatoriali, e un protocollo di livello superiore, definito ad-hoc per rispondere alle specifiche esigenze del contesto applicativo del satellite AraMiS. Questo protocollo di livello superiore si sostituisce a quello di livello Trasporto del protocollo AX.25 connected, il quale era stato progettato per un ambiente in cui i dispositivi comunicanti appartengono alla stessa classe, cioè eseguono operazioni bilanciate (TNC). Nel nostro contesto le operazioni non sono bilanciate, cosicché il meccanismo di controllo di errore, simile allo *Stop-and-wait* (conferma positiva o ritrasmissione), è utilizzato solo per i dati in downlink, e di conseguenza il controllo dei duplicati è effettuato solo per i comandi in uplink. Inoltre, poiché il comportamento che segue ad un comando duplicato è diverso a seconda che questo sia a tempo di esecuzione breve o lungo, esso viene gestito a livello applicazione dal TCP operando sul numero di sequenza dei frame N(S).

I possibili sviluppi di questo lavoro di tesi sono:

- un confronto approfondito tra il protocollo AX.25 connected e il protocollo qui proposto, in particolare in termini di vantaggi e limiti per ciascuno di essi;
- lo studio della possibilità di spostare parte del controllo delle funzioni del protocollo dal Mission Controller alla ground station, dando la facoltà a quest'ultima di gestire in automatico funzionalità più complesse. Ciò al fine di ridurre la frequenza degli scambi dei messaggi via Internet, che, come si è visto, costituiscono uno dei termini peggiorativi più importanti nel definire le prestazioni.

Si conclude osservando come questo lavoro di tesi sia stato condotto a stretto contatto con gli altri membri del team AraMiS, mettendo in evidenza come decisioni riguardanti un sottosistema tipicamente influenzano o vengono influenzate da quelle riguardanti altri sottosistemi. A riguardo dello sviluppo collaborativo, questo lavoro di tesi mi ha dato la possibilità di essere introdotto all'ambiente *Visual Paradigm for UML* e di recepirne le potenzialità riguardanti la produttività e l'affidabilità.

Appendice A

Codici sorgente

A.1 main.c

```
#include <msp430.h>
#include "main.h"
#include "uart.h"
#include "ax25.h"
#include "cc1020.h"
#include "SPI.h"
#include "timer.h"

#define VECT_SIZE 260
signed char option;
unsigned char i;
unsigned char dim;
unsigned char vect[ VECT_SIZE ];
unsigned char test[ VECT_SIZE ];
unsigned char crc;
char tmp,tmp1;

void main ()
{
 short volatile semaforo;
 int i;
 WDTCTL = WDTPW + WDTHOLD;              // Stop watchdog timer
 BCSCTL1 &= ~XT2OFF;                 // XT2on

 do
 {
  IFG1 &= ~OFIFG;                    // Clear OSCFault flag bit
  for (i = 0xFF; i > 0; i--);         // Time for flag to set
 }
 while ((IFG1 & OFIFG));             // OSCFault flag still set?
 BCSCTL2 |= SELM_2 + SELS;           // MCLK = SMCLK = XT2 (safe)
 UART_Init();
 CC1020_Init();
 CC1020_SetupPD();
 _BIS_SR(GIE);
 printUART("RESET\n\r");

 while (1)
 {
  printUART("menu\n\r");
```

```
crc = UART_ReceivePacket(vect);
if (crc != 0)
{
 switch (vect[0])
 {
 case PORTANTE_CMD:
  printUART("PORTANTE\n\r");
  SPI_Init();
  CC1020_Config_Carrier();
  CC1020_WakeUpToTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3);
  CC1020_SetupTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3, POWER_10_DBM);
  semaforo = 5000;
  TIMER_SetupTimer_ms(&semaforo);
  while (semaforo)
   SPI_SendByte(0xFF);
  printUART("carrier\n\r");
  SPI_Disable();
  CC1020_SetupPD();
  break;
 case TX_CMD:
    printUART("TX\n\r");
    SPI_Init();//USART1 in SPI mode
    CC1020_Config_Tx();
    CC1020_WakeUpToTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3);
    CC1020_SetupTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3, POWER_10_DBM);
    AX25_SendPacket(vect+2,vect[1]);
    SPI_Disable();
    CC1020_SetupPD();
  break;
 case RX_CMD:
  /*
  //config port 1
  P1SEL = 0x00;
  P1DIR = 0xFF;
  P1OUT = 0x00;
  */
  printUART("RX\n\r");
  SPI_Init();
  CC1020_Config_Rx();
  CC1020_WakeUpToRX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3);
  CC1020_SetUpToRX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3, 0x00);
  SPI_Disable();  //disable USART module and its SPI mode; use port5 as usual I/O port.
  unsigned int rec_bytes  = AX25_ReceivePacket(&(test[2]));
  if ( rec_bytes != 0 )
  {
   test[0]='R';
   test[1]=rec_bytes;
   uart_SendPacket(test);
   printUART("\n\r");
   CC1020_SetupPD();
  }
```

```
      break;
     default:
      printUART("NO\n\r");
     }
    }
    else
    {
     printUART("NO CRC\n\r");
    }
  }
}
```

A.2 main.h

```
#ifndef MAIN_H
#define MAIN_H

#define FOSC 4000000
#define TX_CMD              'T'
#define RX_CMD              'r'
#define PORTANTE_CMD  'p'
#define STATUS_CMD     's'
#define ECO_CMD        'e'
#define CONFIG_CMD          'c'
#define ON                  1
#define OFF                 0
#define ACK           'A'
#define NACK_WRONG_COMMAND 'n'
#define NACK_WRONG_CRC   'N'
#endif
```

A.3 ax.25.c

```
#include "ax25.h"
#include "uart.h"
#include "SPI.h"
#include "CC1020.h"
#include "timer.h"

int  totalbytes;
char ONEScount = 0;
char lastBit = 0;
char nextBit;
char isFlag;
unsigned int total_byte;
unsigned char preamble = 0x00;
#define INTSCRAMBLER
//look-up table used in Byte-wise CRC caculation
const unsigned short fcstab[256] = {
  0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
  0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
  0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0x643e,
  0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
  0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
  0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0xd9f5,
  0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
  0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xea66, 0xd8fd, 0xc974,
  0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
  0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0xbaf3,
  0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x0528, 0x37b3, 0x263a,
  0xdecd, 0xcf44, 0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0xaa72,
  0x6306, 0x728f, 0x4014, 0x519d, 0x2522, 0x34ab, 0x0630, 0x17b9,
  0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3, 0x8a78, 0x9bf1,
  0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0x0738,
  0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0x8b70,
  0x8408, 0x9581, 0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0xf0b7,
  0x0840, 0x19c9, 0x2b52, 0x3adb, 0x4e64, 0x5fed, 0x6d76, 0x7cff,
  0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324, 0xf1bf, 0xe036,
  0x18c1, 0x0948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0x6c7e,
  0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0xd1b5,
  0x2942, 0x38cb, 0x0a50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0x5dfd,
  0xb58b, 0xa402, 0x9699, 0x8710, 0xf3af, 0xe226, 0xd0bd, 0xc134,
  0x39c3, 0x284a, 0x1ad1, 0x0b58, 0x7fe7, 0x6e6e, 0x5cf5, 0x4d7c,
  0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0xb2b3,
  0x4a44, 0x5bcd, 0x6956, 0x78df, 0x0c60, 0x1de9, 0x2f72, 0x3efb,
  0xd68d, 0xc704, 0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0xa232,
  0x5ac5, 0x4b4c, 0x79d7, 0x685e, 0x1ce1, 0x0d68, 0x3ff3, 0x2e7a,
  0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3, 0x8238, 0x93b1,
  0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0x0e70, 0x1ff9,
  0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0x8330,
  0x7bc7, 0x6a4e, 0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0x0f78
};
```

```
#define PROLOGO_LEN 16
const char prologo[PROLOGO_LEN] = {
 'A' << 1, 'L' << 1, 'L' << 1, ' ' << 1, ' ' << 1, ' ' << 1, 0xE0,
 'A' << 1, 'R' << 1, 'A' << 1, 'M' << 1, 'I' << 1, 'S' << 1, 0x61,
 0x03, 0xF0
};
unsigned char out;
#ifdef INTSCRAMBLER
unsigned short scrambled1, scrambled2;
#else
unsigned char scrambled1, scrambled2, scrambled3;
#endif
union
{
 struct
 {
  unsigned char crc2; //byte piu' significativo
  unsigned char crc1;       //byte meno significativo
 }b;
 unsigned short u;
}fcs;


void pppfcs(unsigned char *cp, int len)
{
 while (len--)
  fcs.u = (fcs.u >> 8) ^ fcstab[(fcs.u ^ *cp++) & 0xFF];
}

void AX25_ComputeFCS(unsigned char * packet, int packet_len)
{
 INITFCS();
 pppfcs(packet, packet_len);
 CPLFCS();

}

void AX25_SendByte(unsigned char byte, int flag)
{
 char i;
 unsigned int tmp;
 unsigned char tx;
 static unsigned char ones_cnt;
 /* for every bit in byte */
 for (i=0; i<8; i++)
 {
  /* extract LSB and shift */
  tmp = byte & 0x01;
  byte >>= 1;
  if (tmp)
   ones_cnt ++;
```

```
   else {
    ones_cnt = 0;
    out = ~out;
   }
#ifdef INTSCRAMBLER
   tx = (out ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
   scrambled2 = (scrambled1 >> 15);
   scrambled1 = (scrambled1 << 1) | tx;
#else
   tx = (out ^ (scrambled2 >> 3) ^ (scrambled3)) & 0x1;
   scrambled3=(((signed char)scrambled2) < 0) ? 1 : 0;
   scrambled2<<=1;
   scrambled2+=((signed char)scrambled1)<0? 1:0;
   scrambled1 = (scrambled1 << 1) | tx;
#endif
   SPI_SendBit(tx);
   /* if sent bit is 1 increment ones counter */
   /* if we have reached max number
   * of ones allowed by AX.25 proto */
   if (ones_cnt >= MAX_AX25_ONES)
   {
    // ... and we reset ones counter
    ones_cnt = 0;
    // ... we send a stuffing 0 bit...
    out = ~out;
#ifdef INTSCRAMBLER
    tx = (out ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
    scrambled2 = (scrambled1 >> 15);
    scrambled1 = (scrambled1 << 1) | tx;
#else
    tx = (out ^ (scrambled2 >> 3) ^ (scrambled3)) & 0x1;
    scrambled3=((signed char)scrambled2)<0? 1:0;
    scrambled2<<=1;
    scrambled2+=((signed char)scrambled1)<0? 1:0;
    scrambled1 = (scrambled1 << 1) | tx;
#endif
    // send it
    SPI_SendBit(tx);
   }
   if(flag)
    ones_cnt=0;
  }
}

void AX25_SendPacket(unsigned char * packet, unsigned int packet_len)
{
  int i;
  scrambled1 = 0;
  scrambled2 = 0;
#ifndef INTSCRAMBLER
  scrambled3 = 0;
```

```
#endif
  out = 0;
  INITFCS();
  pppfcs( (unsigned char *)prologo, PROLOGO_LEN);
  pppfcs( packet, packet_len);
  CPLFCS();
  SPI_ResetBit();
  /* prologo transmission */
  for (i=0; i<TX_DELAY; i++)
    AX25_SendByte(FLAG, 1);
  /* send prologo */
  for (i=0; i<PROLOGO_LEN; i++)
    AX25_SendByte (prologo[i], 0);
  /* send packet */
  for (i=0; i<packet_len; i++)
    AX25_SendByte (packet[i], 0);
  /* now we send 15:8 bits MSB first */
  AX25_SendByte(fcs.b.crc2, 0);
  /* and now 7:0 MSB first */
  AX25_SendByte(fcs.b.crc1, 0);
  for (i=0; i<2; i++)
    AX25_SendByte(FLAG, 1); /* ones count =-3 removes bit stuffing from Flag TX */
  AX25_SendByte(0x00, 1);       //wait for the end of transmission (one more byte, to ensure reception)
}

unsigned int AX25_ReceivePacket(unsigned char *data)
{
  int flag_counter = 0;
  char byte = 0x00;
  char DIN;
  short volatile semaforo;
  semaforo = 25000;
  TIMER_SetupTimer_ms(&semaforo);
  do
  {
    scrambled1 = 0;
    scrambled2 = 0;
    byte = 0;
    do
    {
      ONEScount = 0;
      lastBit = 0;
      totalbytes = 0;
      // Check for flag ------------------------------------------------------------------
      while ( (byte != FLAG) && semaforo )              // Check for flag
      {
        byte = byte >> 1;               // shift byte 1 bit to the right inserting a 0 as MSB
        while( (P5IN & DCLK) == 0 );    // wait for CC1020 clock to rise and then
        //P1OUT = 0x01;
        DIN = ( (P5IN & DIO) != 0 );        // sample nextBit from CC1020 DIO
        nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
```

```
    scrambled2 = (scrambled1 >> 15);
    scrambled1 = (scrambled1 << 1) | DIN;
    if ( nextBit == lastBit )          // if no change, then bit is a 1 using NRZI
    {
      byte = byte | 0x80;              // insert a 1 as the MSB by ORing 'byte' with 0x80
    }
    else                          // otherwise, bit is a 0 using NRZI
    {
      byte &= 0x7f;                // insert a 0 as the MSB by ANDing 'byte' with 0x7F
    }
    lastBit = nextBit;                // change the value of lastBit for next NRZI comparison
    //P1OUT = 0x00;     // one bit descrambing needs 10 us
    while( (P5IN & DCLK) != 0 );     // wait for next clock cycle
   }
   // End flag check ------------------------------------------------------------------
   byte = 0x00;               // reset byte so it is no longer 0x7e
   totalbytes = 0;
   flag_counter = 1;
   byte = RxByte();
   while ( byte == FLAG )
   {
    byte = RxByte();
    flag_counter++;
   }
  }while ((flag_counter < MIN_FLAG_PREAMBLE) && semaforo);
  INITFCS();     //initial FCS value

  while (( byte != FLAG ) && (totalbytes < MAX_PACKET_SIZE) && semaforo )
  {
   data[totalbytes] = byte;
   totalbytes++;
   //P1OUT = 0x01;
   fcs.u = (fcs.u >> 8) ^ fcstab[(fcs.u ^ data[totalbytes-1]) & 0xFF];
   if ( (fcs.b.crc2 == (data[totalbytes-2]^0xFF)) && (fcs.b.crc1 == (data[totalbytes-1]^0xFF)) )
    return totalbytes-2;
   //P1OUT = 0x00;      // one byte CRC caculation needs 5us when FOSC = 8 MHz
   byte = RxByte();
  }

 } while ( (totalbytes < MIN_PACKET_LENGTH) && semaforo);

//  AX25_ComputeFCS(data, totalbytes);          //receiver caculates CRC
//  if ( (fcs.b.crc2 == data[totalbytes-2]) && (fcs.b.crc1 == data[totalbytes-1]) )
//  //    if ( (fcs.b.crc2 == (data[totalbytes-2]^0xFF)) && (fcs.b.crc1 == (data[totalbytes-1]^0xFF)) )
//  return totalbytes-2;
//    else
//  return 0;


} // End void RxPacket(void)
```

```
char RxByte(void)
{
  char byte = 0x00;
  char DIN;
  for ( int len = 0; len < 8; len++ )            // for all 8 bits
  {
    byte = byte >> 1;                // shift the byte over to the right inserting a 0 for MSB
    while( (P5IN & DCLK) == 0 );     // wait for CC1020 clock to rise and then
    DIN = ( (P5IN & DIO) != 0 );        // sample nextBit from CC1020 DIO
    nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
    scrambled2 = (scrambled1 >> 15);
    scrambled1 = (scrambled1 << 1) | DIN;
    if ( nextBit != lastBit )
    {
      byte &= 0x7f;                  // zero out MSB
      ONEScount = 0;                 // reset ONEScount
    }
    else if ( nextBit == lastBit )     // then no change so one in NRZI
    {
      byte = byte | 0x80;        // MSB when shifting right
      ONEScount = ONEScount + 1;
    }
    lastBit = nextBit;
    while( (P5IN & DCLK) != 0 );     // wait for next clock cycle

    //-------------------------------------------------------------------------------------
    // if there have been 5 ones and the next bit is a zero (change in bit stream)
    // then remove the bitstuffed zero.
    if ( ONEScount >= 5 )
    {
      //still need to right shift byte---lv
      byte = byte >> 1;
      // Check the next bit to see if it is a bitstuffed zero; if it is not then it
      // is probably the flag
      len = len + 1;
      while( (P5IN & DCLK) == 0 );     // wait for CC1020 clock to rise and then
      //for (int j = 0; j < 10; j++) asm("nop;");
      DIN = ( (P5IN & DIO) != 0 );        // sample nextBit from CC1020 DIO
      nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
      scrambled2 = (scrambled1 >> 15);
      scrambled1 = (scrambled1 << 1) | DIN;
      if (nextBit != lastBit)     // then zero needs to be skipped
      {
        lastBit = nextBit;     // fix lastBit for NRZI
        ONEScount = 0;         // reset ONEScount
        len--;
        byte = byte << 1;     //shift back, zero skipped---lv
        while( (P5IN & DCLK) != 0 );     // wait for next clock cycle
      }
      else                    // otherwise, flag byte has been encountered
      {
```

```
      while( (P5IN & DCLK) != 0 );     // wait for next clock cycle
      ONEScount = 0;         // Reset ones since we're assuming a flag has been received
      len = len + 1;
      while( (P5IN & DCLK) == 0 );     // wait for CC1020 clock to rise and then
      DIN = ( (P5IN & DIO) != 0 );        // sample nextBit from CC1020 DIO
      nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
      scrambled2 = (scrambled1 >> 15);
      scrambled1 = (scrambled1 << 1) | DIN;
      lastBit = nextBit;
      while( (P5IN & DCLK) != 0 );     // wait for next clock cycle
      return 0x7e;          // If we have 6 ones then it is either the flag
      // or an error.  Either way we trick TxPacket into
      // thinking it is an end flag by setting it as 0x7e &
     }
    }
  }
  return byte;
}
```

A.4. ax25.h

```
#ifndef APRS_H
#define APRS_H

#define PPPINITFCS     0xffff  /* Initial FCS value */
#define INITFCS() fcs.u=PPPINITFCS
#define CPLFCS() fcs.u ^=0xffff

#define TX_DELAY                    80
#define  FLAG                       0x7E
#define MAX_AX25_ONES                    5
#define MAX_PACKET_SIZE       255
#define MIN_FLAG_PREAMBLE     7
#define MIN_PACKET_LENGTH     17

void AX25_SendPacket(unsigned char * packet, unsigned int packet_len);
unsigned int AX25_ReceivePacket(unsigned char *data);
char RxByte(void);
#endif
```

A.5. cc1020.c

```
#include <msp430.h>
#include "main.h"
#include "uart.h"
#include "cc1020.h"

//frequency sepration, choose one
//#define FREQ_SEP_4_95
#define FREQ_SEP_5_85

//channel spacing,choose one
//#define CHANNEL_SPCACE_12_5
//#define CHANNEL_SPCACE_25
#define CHANNEL_SPCACE_50
//#define CHANNEL_SPCACE_100

void CC1020_Init(void)
{
  CONFIG_PORT_OUT = (1<<PSEL);
  CONFIG_PORT_SEL &= ~((1<<PSEL) | (1<<PDI) | (1<<PCLK) | (1<<PDO));
  CONFIG_PORT_DIR |= (1<<PSEL) | (1<<PDI) | (1<<PCLK);
  CONFIG_PORT_DIR &= ~(1<<PDO);
}

char CC1020_Reset(void)
{
  // after RESET CC1020 is in Power-Down mode: it is necessary to wake it up for tx/rx
  CC1020_SetReg(CC1020_MAIN,PD_MODE_1 | FS_PD | XOSC_PD | BIAS_PD);
  CC1020_SetReg(CC1020_MAIN,PD_MODE_1 | FS_PD | XOSC_PD | BIAS_PD | RESET_N);
  if ( CC1020_ReadReg(CC1020_FREQ_1A) == 0xB1)
    return TRUE;
  else
    return FALSE;
}

char CC1020_Config_Carrier(void)
{
  char status;
  if (CC1020_Reset() == FALSE)
    printUART ("RESET CC1020 failed.\r\n");

  CC1020_SetReg(CC1020_MAIN,0x81);
  CC1020_SetReg(CC1020_INTERFACE,0x4F);       //IMPORTANT:  SEP_DI_DO = 1
  CC1020_SetReg(CC1020_RESET,0xFF);
  CC1020_SetReg(CC1020_SEQUENCING,0x8F);
  // fref=14.745e6/(REF_DIV_A+1)=7.3725e6
  // FREQ_A=0x3A7AF1=1916280.5
  // fc=436674943
  CC1020_SetReg(CC1020_FREQ_2A,0x3A);
  CC1020_SetReg(CC1020_FREQ_1A,0x7A);
```

```
  CC1020_SetReg(CC1020_FREQ_0A,0xF1);
  // REF_DIV_A = 1
  CC1020_SetReg(CC1020_CLOCK_A,0x39);
  // FREQ_B=0x3A859D=1917646.5
  // fc=436982281
  CC1020_SetReg(CC1020_FREQ_2B,0x3A);
  CC1020_SetReg(CC1020_FREQ_1B,0x85);
  CC1020_SetReg(CC1020_FREQ_0B,0x9D);
  // REF_DIV_B = 1
  CC1020_SetReg(CC1020_CLOCK_B,0x39);
  CC1020_SetReg(CC1020_VCO,0x44);
  CC1020_SetReg(CC1020_MODEM,0x50);
  CC1020_SetReg(CC1020_DEVIATION,0x00);
  CC1020_SetReg(CC1020_AFC_CONTROL,0x3B);
  CC1020_SetReg(CC1020_FILTER,0x27);
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x13);
  CC1020_SetReg(CC1020_VGA4,0x17);
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);
  CC1020_SetReg(CC1020_CALIBRATE,0x34);
  CC1020_SetReg(CC1020_PA_POWER,0xF0);
  CC1020_SetReg(CC1020_MATCH,0x00);
  CC1020_SetReg(CC1020_PHASE_COMP,0x00);
  CC1020_SetReg(CC1020_GAIN_COMP,0x00);
  CC1020_SetReg(CC1020_POWERDOWN,0x00);
  CC1020_SetReg(CC1020_TEST1,0x4D);
  CC1020_SetReg(CC1020_TEST2,0x10);
  CC1020_SetReg(CC1020_TEST3,0x06);
  CC1020_SetReg(CC1020_TEST4,0x00);
  CC1020_SetReg(CC1020_TEST5,0x40);
  CC1020_SetReg(CC1020_TEST6,0x00);
  CC1020_SetReg(CC1020_TEST7,0x00);

  status = CC1020_ReadReg(CC1020_STATUS2); //status2
  return(status);
}

char CC1020_Config_Tx(void)
{
  char status;
  if (CC1020_Reset() == FALSE)
    printUART ("RESET CC1020 failed.\r\n");

  CC1020_SetReg(CC1020_MAIN,0x81);
  CC1020_SetReg(CC1020_INTERFACE,0x0F);        //IMPORTANT:  SEP_DI_DO = 1
```

```
  CC1020_SetReg(CC1020_RESET,0xFF);
  CC1020_SetReg(CC1020_SEQUENCING,0x8F);
  CC1020_SetReg(CC1020_FREQ_2A,0x3A);
  CC1020_SetReg(CC1020_FREQ_1A,0x7A);
  CC1020_SetReg(CC1020_FREQ_0A,0xF1);
  CC1020_SetReg(CC1020_CLOCK_A,0x39);
  CC1020_SetReg(CC1020_FREQ_2B,0x3A);
  CC1020_SetReg(CC1020_FREQ_1B,0x85);
  CC1020_SetReg(CC1020_FREQ_0B,0x9D);
  CC1020_SetReg(CC1020_CLOCK_B,0x39);
  CC1020_SetReg(CC1020_VCO,0x44);
  CC1020_SetReg(CC1020_MODEM,0x50);

#ifdef FREQ_SEP_4_95
  CC1020_SetReg(CC1020_DEVIATION,0x1B);       //
  CC1020_SetReg(CC1020_AFC_CONTROL,0xC6);     //
#endif
#ifdef FREQ_SEP_5_85
  CC1020_SetReg(CC1020_DEVIATION,0x1D);       //
  CC1020_SetReg(CC1020_AFC_CONTROL,0xC7);     //
#endif

#ifdef CHANNEL_SPCACE_12_5
  CC1020_SetReg(CC1020_FILTER,0x58);         //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2B);          //
  CC1020_SetReg(CC1020_VGA4,0x08);          //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0x9E);          //
#endif
#ifdef CHANNEL_SPCACE_25
  CC1020_SetReg(CC1020_FILTER,0x2F);         //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2F);          //
  CC1020_SetReg(CC1020_VGA4,0x2D);          //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);          //
#endif
#ifdef CHANNEL_SPCACE_50
  CC1020_SetReg(CC1020_FILTER,0x2B);         //
  CC1020_SetReg(CC1020_VGA1,0x61);
```

```
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2E);         //
  CC1020_SetReg(CC1020_VGA4,0x29);         //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);          //
#endif
#ifdef CHANNEL_SPCACE_100
  CC1020_SetReg(CC1020_FILTER,0x25);        //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2D);         //
  CC1020_SetReg(CC1020_VGA4,0x29);         //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);          //
#endif

  CC1020_SetReg(CC1020_CALIBRATE,0x34);
  CC1020_SetReg(CC1020_PA_POWER,0xF0);
  CC1020_SetReg(CC1020_MATCH,0x00);
  CC1020_SetReg(CC1020_PHASE_COMP,0x00);
  CC1020_SetReg(CC1020_GAIN_COMP,0x00);
  CC1020_SetReg(CC1020_POWERDOWN,0x00);
  CC1020_SetReg(CC1020_TEST1,0x4D);
  CC1020_SetReg(CC1020_TEST2,0x10);
  CC1020_SetReg(CC1020_TEST3,0x06);
  CC1020_SetReg(CC1020_TEST4,0x00);
  CC1020_SetReg(CC1020_TEST5,0x40);
  CC1020_SetReg(CC1020_TEST6,0x00);
  CC1020_SetReg(CC1020_TEST7,0x00);

  status = CC1020_ReadReg(CC1020_STATUS2); //status2
  return(status);
}

char CC1020_Config_Rx(void)
{
  char status;
  if (CC1020_Reset() == FALSE)
    printUART ("RESET CC1020 failed.\r\n");

  CC1020_SetReg(CC1020_MAIN,0x01);
  CC1020_SetReg(CC1020_INTERFACE,0x0F);//////////1F or 0F
  CC1020_SetReg(CC1020_RESET,0xFF);
```

```
  CC1020_SetReg(CC1020_SEQUENCING,0x8F);
  CC1020_SetReg(CC1020_FREQ_2A,0x3A);
  CC1020_SetReg(CC1020_FREQ_1A,0x7A);
  CC1020_SetReg(CC1020_FREQ_0A,0xF1);
  CC1020_SetReg(CC1020_CLOCK_A,0x39);
  CC1020_SetReg(CC1020_FREQ_2B,0x3A);
  CC1020_SetReg(CC1020_FREQ_1B,0x85);
  CC1020_SetReg(CC1020_FREQ_0B,0x9D);
  CC1020_SetReg(CC1020_CLOCK_B,0x39);
  CC1020_SetReg(CC1020_VCO,0x44);
  CC1020_SetReg(CC1020_MODEM,0x50);

#ifdef FREQ_SEP_4_95
  CC1020_SetReg(CC1020_DEVIATION,0x1B);       //
  CC1020_SetReg(CC1020_AFC_CONTROL,0xc6);     ///////////////0 8 c
#endif
#ifdef FREQ_SEP_5_85
  CC1020_SetReg(CC1020_DEVIATION,0x1D);       //
  CC1020_SetReg(CC1020_AFC_CONTROL,0xc7);     //////////
#endif

#ifdef CHANNEL_SPCACE_12_5
  CC1020_SetReg(CC1020_FILTER,0x58);        //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2B);         //
  CC1020_SetReg(CC1020_VGA4,0x08);         //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0x9E);          //
#endif
#ifdef CHANNEL_SPCACE_25
  CC1020_SetReg(CC1020_FILTER,0x2F);        //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2F);         //
  CC1020_SetReg(CC1020_VGA4,0x2D);         //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);          //
#endif
#ifdef CHANNEL_SPCACE_50
  CC1020_SetReg(CC1020_FILTER,0x2B);        //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
```

```
  CC1020_SetReg(CC1020_VGA3,0x2E);         //
  CC1020_SetReg(CC1020_VGA4,0x29);         //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);          //
#endif
#ifdef CHANNEL_SPCACE_100
  CC1020_SetReg(CC1020_FILTER,0x25);        //
  CC1020_SetReg(CC1020_VGA1,0x61);
  CC1020_SetReg(CC1020_VGA2,0x55);
  CC1020_SetReg(CC1020_VGA3,0x2D);         //
  CC1020_SetReg(CC1020_VGA4,0x29);         //
  CC1020_SetReg(CC1020_LOCK,0x20);
  CC1020_SetReg(CC1020_FRONTEND,0x78);
  CC1020_SetReg(CC1020_ANALOG,0x47);
  CC1020_SetReg(CC1020_BUFF_SWING,0x14);
  CC1020_SetReg(CC1020_BUFF_CURRENT,0x22);
  CC1020_SetReg(CC1020_PLL_BW,0xAE);          //
#endif

  CC1020_SetReg(CC1020_CALIBRATE,0x34);
  CC1020_SetReg(CC1020_PA_POWER,0x0F);
  CC1020_SetReg(CC1020_MATCH,0x00);
  CC1020_SetReg(CC1020_PHASE_COMP,0x00);
  CC1020_SetReg(CC1020_GAIN_COMP,0x00);
  CC1020_SetReg(CC1020_POWERDOWN,0x00);
  CC1020_SetReg(CC1020_TEST1,0x4D);
  CC1020_SetReg(CC1020_TEST2,0x10);
  CC1020_SetReg(CC1020_TEST3,0x06);
  CC1020_SetReg(CC1020_TEST4,0x00);
  CC1020_SetReg(CC1020_TEST5,0x40);
  CC1020_SetReg(CC1020_TEST6,0x00);
  CC1020_SetReg(CC1020_TEST7,0x00);

  status = CC1020_ReadReg(CC1020_STATUS2); //status2
  return(status);

}

void CC1020_WakeUpToTX(char TXANALOG)
{
  volatile int i;
  // Turn on xtal oscillator core
  CC1020_SetReg(CC1020_MAIN, RESET_N | BIAS_PD | FS_PD | PD_MODE_1 | F_REG | RXTX );
  // Setup bias current adjustment
  CC1020_SetReg(CC1020_ANALOG,TXANALOG);
  // Insert wait routine here, must wait for xtal oscillator to stabilise,
  // typically takes 2-5ms.
```

```
  // whait for 8 ms
  for (i=0xBB8; i > 0; i--) asm("nop;");
  //P2OUT=0;
  // Turn on bias generator
  CC1020_SetReg(CC1020_MAIN, RESET_N | FS_PD | PD_MODE_1 | F_REG | RXTX);
  // Wait for 150 usec
  for (i=0x0038; i > 0; i--) asm("nop;");
  // Turn on frequency synthesiser
  CC1020_SetReg(CC1020_MAIN, RESET_N | PD_MODE_1 | F_REG | RXTX);
}

char CC1020_Calibrate(char pa_power)
{
  volatile unsigned int TimeOutCounter;
  volatile int nCalAttempt;
  volatile char tmp;
  // Turn off PA to avoid spurs during calibration in TX mode
  CC1020_SetReg(CC1020_PA_POWER,POWER_m25_DBM);
  // Calibrate, and re-calibrate if necessary:
  for (nCalAttempt = CAL_ATTEMPT_MAX; (nCalAttempt>0); nCalAttempt--)
  {
    // Start calibration
    CC1020_SetReg(CC1020_CALIBRATE,CAL_ITERATE_4 | CAL_WAIT_3 | CAL_START);
    // Monitor actual calibration start (ref. Errata Note 04 - CC1020)   CAL_COMPLETE
    // wait 1 ms (should be at least 100 us)
    for ( TimeOutCounter = 0x0040; TimeOutCounter > 0; TimeOutCounter--)  asm("nop;");
    TimeOutCounter = 300;
    while((CC1020_ReadReg(CC1020_STATUS)&CAL_COMPLETE) != CAL_COMPLETE )
    {
      TimeOutCounter--;
      if ( TimeOutCounter <= 0 )
        return LOCK_NOK;
    }

    TimeOutCounter = 300;
    char reg = CC1020_ReadReg(CC1020_STATUS);
    while(( reg & LOCK_CONTINUOUS) !=  LOCK_CONTINUOUS )
    {
      TimeOutCounter--;
      if ( TimeOutCounter <= 0 )
        return LOCK_NOK;
      reg = CC1020_ReadReg(CC1020_STATUS);
    }
    // Abort further recalibration attempts if successful LOCK
    if((CC1020_ReadReg(CC1020_STATUS)&LOCK_CONTINUOUS) == LOCK_CONTINUOUS) {
      break;
    }
  }
  // Restore PA setting
  CC1020_SetReg(CC1020_PA_POWER, pa_power);
  // Return state of LOCK_CONTINUOUS bit
```

```
  return ((CC1020_ReadReg(CC1020_STATUS)&LOCK_CONTINUOUS)==LOCK_CONTINUOUS);
}

char CC1020_SetupTX(char txanalog, char pa_power)
{
  volatile int TimeOutCounter;
  char lock_status;
  // Turn off PA to avoid frequency splatter
  CC1020_SetReg(CC1020_PA_POWER,POWER_m25_DBM);
  // Setup bias current adjustment
  CC1020_SetReg(CC1020_ANALOG,txanalog);
  // Switch into TX, switch to freq. reg B (0xC1)
  CC1020_SetReg(CC1020_MAIN, RESET_N | F_REG | RXTX);
  // Monitor LOCK
  for(TimeOutCounter=LOCK_TIMEOUT;
((CC1020_ReadReg(CC1020_STATUS)&PLL_LOCK)==0)&&(TimeOutCounter>0); TimeOutCounter--);
  // If PLL in lock
  if((CC1020_ReadReg(CC1020_STATUS)&PLL_LOCK)==PLL_LOCK){
    // Indicate PLL in LOCK
    lock_status = LOCK_OK;
    // Else (PLL out of LOCK)
  }else{
    // If recalibration ok
    if(CC1020_Calibrate(pa_power)){
      // Indicate PLL in LOCK
      lock_status = LOCK_RECAL_OK;
      // Else (recalibration failed)
    }else{
      // Indicate PLL out of LOCK
      lock_status = LOCK_NOK;
      printUART ("LOCK_NOK\r\n");
    }
  }
  // Restore PA setting
  CC1020_SetReg(CC1020_PA_POWER,pa_power);
  // Turn OFF DCLK squelch in TX
  CC1020_SetReg(CC1020_INTERFACE,CC1020_ReadReg(CC1020_INTERFACE)&~DCLK_CS);
  // Return LOCK status to application
  return (lock_status);
}


/*  This routine wakes the CC1020 up from PD mode to RX mode            */
/*************************************************************************/

void CC1020_WakeUpToRX(char RXANALOG)
{
  volatile int i;
  // Turn on xtal oscillator core
  CC1020_SetReg(CC1020_MAIN, RESET_N | BIAS_PD | FS_PD | PD_MODE_1);
  // Setup bias current adjustment
```

```
  CC1020_SetReg(CC1020_ANALOG,RXANALOG);
  // Insert wait routine here, must wait for xtal oscillator to stabilise,
  // typically takes 2-5ms.
  for (i=0xBB8; i > 0; i--) asm("nop;");
  // Turn on bias generator
  CC1020_SetReg(CC1020_MAIN, RESET_N | FS_PD | PD_MODE_1);
  // Wait for 150 usec
  for (i=0x0038; i > 0; i--) asm("nop;");
  // Turn on frequency synthesiser
  CC1020_SetReg(CC1020_MAIN, RESET_N | PD_MODE_1);
}

/*  This routine puts the CC1020 into RX mode (from TX). When switching to  */
/*  RX from PD, use WakeupC1020ToRX first                                    */
/***************************************************************************/
char CC1020_SetUpToRX(char RXANALOG, char PA_POWER)
{
  volatile int TimeOutCounter;
  char lock_status;
  // Switch into RX, switch to freq. reg A
  CC1020_SetReg(CC1020_MAIN, RESET_N | PD_MODE_1);
  // Setup bias current adjustment
  CC1020_SetReg(CC1020_ANALOG,RXANALOG);
  // Monitor LOCK
  for(TimeOutCounter=LOCK_TIMEOUT;
((CC1020_ReadReg(CC1020_STATUS)&0x10)==0)&&(TimeOutCounter>0); TimeOutCounter--);
  // If PLL in lock
  if((CC1020_ReadReg(CC1020_STATUS)&0x10)==0x10){
    // Indicate PLL in LOCK
    lock_status = LOCK_OK;
    printUART ("LOCK_OK\r\n");
    // Else (PLL out of LOCK)
  }else{
    // If recalibration ok
    if(CC1020_Calibrate(PA_POWER)){
      // Indicate PLL in LOCK
      lock_status = LOCK_RECAL_OK;
      //printUART ("LOCK_R_OK\r\n");
      // Else (recalibration failed)
    }else{
      // Indicate PLL out of LOCK
      lock_status = LOCK_NOK;
      printUART ("LOCK_NOK\r\n");
    }
  }
  // Switch RX part of CC1020 on
  CC1020_SetReg(CC1020_MAIN, RESET_N);
  // Return LOCK status to application
  return (lock_status);
}
```

```
void CC1020_SetupPD(void)
{
 // Put CC1020 into power-down 0x1F
 CC1020_SetReg(CC1020_MAIN, RESET_N | BIAS_PD | XOSC_PD | FS_PD | PD_MODE_1 );
 // Turn off PA to minimise current draw
 CC1020_SetReg(CC1020_PA_POWER,0x00);
}

void CC1020_SetReg(char registro, char dato)
{
 char count,wait;
 CONFIG_PORT_OUT = 0;
 WAIT_CYCLE();
 for (count = 6; count != 255; count --)
 {
  CONFIG_PORT_OUT = (((registro >> count) & 0x01) << PDI );
  WAIT_CYCLE();
  CONFIG_PORT_OUT |= (1<<PCLK);
  WAIT_CYCLE();
  CONFIG_PORT_OUT &= ~(1<<PCLK);
 }
 //write byte
 CONFIG_PORT_OUT |= 1<<PDI;
 WAIT_CYCLE();
 CONFIG_PORT_OUT |= (1<<PCLK);
 WAIT_CYCLE();
 CONFIG_PORT_OUT &= ~(1<<PCLK);
 WAIT_CYCLE();
 for (count = 7; count != 255; count --)
 {
  CONFIG_PORT_OUT = (((dato >> count) & 0x01) << PDI );
  WAIT_CYCLE();
  CONFIG_PORT_OUT |= (1<<PCLK);
  WAIT_CYCLE();
  CONFIG_PORT_OUT &= ~(1<<PCLK);
 }
 WAIT_CYCLE();
 CONFIG_PORT_OUT = (1<<PSEL);
 WAIT_CYCLE();
}


char CC1020_ReadReg(char registro)
{
 char count,wait;
 char temp = 0;

 //setto PSEL a 0
 //P1OUT &= ~(BIT3 | BIT2);
 CONFIG_PORT_OUT = 0;
 WAIT_CYCLE();
```

```
 for (count = 6; count != 255; count --)
 {
  CONFIG_PORT_OUT = (((registro >> count) & 0x01) << PDI );
  WAIT_CYCLE();
  CONFIG_PORT_OUT |= (1<<PCLK);
  WAIT_CYCLE();
  CONFIG_PORT_OUT &= ~(1<<PCLK);
 }
 //read byte
 CONFIG_PORT_OUT &= ~(1<<PDI);
 WAIT_CYCLE();
 CONFIG_PORT_OUT |= (1<<PCLK);
 WAIT_CYCLE();
 CONFIG_PORT_OUT &= ~(1<<PCLK);
 WAIT_CYCLE();
 for (count = 7; count != 255; count --)
 {
  CONFIG_PORT_OUT |= (1<<PCLK);
  WAIT_CYCLE();
  temp |= ((CONFIG_PORT_IN & (1<<PDO))>>PDO) << count;
  CONFIG_PORT_OUT &= ~(1<<PCLK);
  WAIT_CYCLE();
 }
 CONFIG_PORT_OUT = (1<<PSEL);
 WAIT_CYCLE();
 return temp;
}
```

A.6. cc1020.h

```
#ifndef CC1020_H
#define CC1020_H

#define PIN0            0
#define PIN1            1
#define PIN2            2
#define PIN3            3
#define PIN4            4
#define PIN5            5
#define PIN6            6
#define PIN7            7
#define PIN8            8
#define PIN9            9
#define PINA            10
#define PINB            11
#define PINC            12
#define PIND            13
#define PINE            14
#define PINF            15

#define CONFIG_PORT_IN P2IN
#define CONFIG_PORT_OUT P2OUT
#define CONFIG_PORT_DIR P2DIR
#define CONFIG_PORT_SEL P2SEL

#define PSEL     PIN0
#define PCLK     PIN1
#define PDI      PIN2
#define PDO      PIN3
#define LOCK_FLAG PIN4

/* Constants defined for CC1020 */
void CC1020_SetReg(char registro, char dato);
char CC1020_ReadReg(char registro);
void CC1020_Init(void);
char CC1020_Reset(void);
void CC1020_WakeUpToTX(char txanalog);
char CC1020_Calibrate(char pa_power);
char CC1020_SetupTX(char txanalog, char pa_power);
void CC1020_SetupPD(void);
char CC1020_Config_Carrier(void);
char CC1020_Config_Tx(void);
char CC1020_Config_Rx(void);
void CC1020_WakeUpToRX(char RXANALOG);
char CC1020_SetUpToRX(char RXANALOG, char PA_POWER);

#define WAIT_CYCLE()  for(wait=0;wait < 25; wait++) asm("nop;")

#define TRUE 1
```

```
#define FALSE 0

/* Register addresses */

#define CC1020_MAIN          0x00
#define CC1020_INTERFACE     0x01
#define CC1020_RESET         0x02
#define CC1020_SEQUENCING    0x03
#define CC1020_FREQ_2A       0x04
#define CC1020_FREQ_1A       0x05
#define CC1020_FREQ_0A       0x06
#define CC1020_CLOCK_A       0x07
#define CC1020_FREQ_2B       0x08
#define CC1020_FREQ_1B       0x09
#define CC1020_FREQ_0B       0x0A
#define CC1020_CLOCK_B       0x0B
#define CC1020_VCO           0x0C
#define CC1020_MODEM         0x0D
#define CC1020_DEVIATION     0x0E
#define CC1020_AFC_CONTROL   0x0F
#define CC1020_FILTER        0x10
#define CC1020_VGA1          0x11
#define CC1020_VGA2          0x12
#define CC1020_VGA3          0x13
#define CC1020_VGA4          0x14
#define CC1020_LOCK          0x15
#define CC1020_FRONTEND      0x16
#define CC1020_ANALOG        0x17
#define CC1020_BUFF_SWING    0x18
#define CC1020_BUFF_CURRENT  0x19
#define CC1020_PLL_BW        0x1A
#define CC1020_CALIBRATE     0x1B
#define CC1020_PA_POWER      0x1C
#define CC1020_MATCH         0x1D
#define CC1020_PHASE_COMP    0x1E
#define CC1020_GAIN_COMP     0x1F
#define CC1020_POWERDOWN     0x20
#define CC1020_TEST1         0x21
#define CC1020_TEST2         0x22
#define CC1020_TEST3         0x23
#define CC1020_TEST4         0x24
#define CC1020_TEST5         0x25
#define CC1020_TEST6         0x26
#define CC1020_TEST7         0x27
#define CC1020_STATUS        0x40
#define CC1020_RESET_DONE    0x41
#define CC1020_RSSI          0x42
#define CC1020_AFC           0x43
#define CC1020_GAUSS_FILTER  0x44
#define CC1020_STATUS1       0x45
#define CC1020_STATUS2       0x46
```

```
#define CC1020_STATUS3      0x47
#define CC1020_STATUS4      0x48
#define CC1020_STATUS5      0x49
#define CC1020_STATUS6      0x4A
#define CC1020_STATUS7      0x4B

#define RXDEVIATION    0x9D
#define RXTX           BIT7
#define F_REG          BIT6
#define PD_MODE_0      0
#define PD_MODE_1      BIT4
#define PD_MODE_2      BIT5
#define PD_MODE_3      BIT4 | BIT5
#define FS_PD          BIT3
#define XOSC_PD        BIT2
#define BIAS_PD        BIT1
#define RESET_N        BIT0
#define BANDSELECT     BIT7
#define LO_DC          BIT6
#define VGA_BLANKING   BIT5
#define PD_LONG        BIT4
#define PA_BOOST       BIT2
#define DIV_BUFF_CURRENT_0    0
#define DIV_BUFF_CURRENT_1    BIT0
#define DIV_BUFF_CURRENT_2    BIT1
#define DIV_BUFF_CURRENT_3    BIT0 | BIT1
#define PLL_LOCK       BIT4
#define CAL_ITERATE_0  0
#define CAL_ITERATE_1  BIT0
#define CAL_ITERATE_2  BIT1
#define CAL_ITERATE_3  BIT0 | BIT1
#define CAL_ITERATE_4  BIT2
#define CAL_ITERATE_5  BIT2 | BIT0
#define CAL_ITERATE_6  BIT2 | BIT1
#define CAL_ITERATE_7  BIT2 | BIT1 | BIT0
#define CAL_WAIT_0     0
#define CAL_WAIT_1     BIT4
#define CAL_WAIT_2     BIT5
#define CAL_WAIT_3     BIT4 | BIT5
#define CAL_START      BIT7
#define CAL_COMPLETE   BIT7
#define LOCK_CONTINUOUS   BIT4
#define DCLK_CS        BIT4

/* Functions for accessing the CC1020 */

extern void ConfigureCC1020(char Count, short Configuration[]);
extern void SetupCC1020ForSPI(void);
extern void WriteToCC1020Register(char addr, char data);
extern void WriteToCC1020RegisterWord(short addranddata);
extern char ReadFromCC1020Register(char addr);
```

```
extern void ResetCC1020(void);
extern char SetupCC1020RX(char RXANALOG, char PA_POWER);
extern void WakeUpCC1020ToRX(char RXANALOG);
extern int  ReadRSSIlevelCC1020(void);

#define CAL_ATTEMPT_MAX 4
// Time-out values
#define CAL_TIMEOUT    (unsigned int)0x7FFE
#define LOCK_TIMEOUT   (unsigned int)0x0001
#define RESET_TIMEOUT  (unsigned int)0x7FFE

// LOCK status definitions
#define  LOCK_NOK        0x00
#define  LOCK_OK         0x01
#define  LOCK_RECAL_OK    0x02

// output power settings
#define POWER_0_DBM 0x0F
#define POWER_3_DBM 0x60
#define POWER_5_DBM 0x80
#define POWER_10_DBM 0xF0
#define POWER_m5_DBM 0x09
#define POWER_m25_DBM 0x02

// PA power setting
#define PA_VALUE 0xA0

#endif
```

A.7. SPI.c

```
#include "main.h"
#include "uart.h"
#include "SPI.h"
#include <msp430x14x.h>

unsigned char SPI_tmp_value;
signed char SPI_tmp_index;

void SPI_Init(void)
{
  P5SEL |= DIO | DCLK | LOCK_BIT;       // P5.1,2,3 SPI option select
  U1CTL = CHAR + SYNC + SWRST;          // 8-bit, SPI, Slave
  //U1TCTL = STC;                       // Polarity, SMCLK, 3-wire
  U1TCTL = 0x82;                       //UCLK delayed one half circle--lv
  U1BR0 = 0x02;                       // SPICLK = SMCLK/2
  U1BR1 = 0x02;
  U1MCTL = 0x00;                       //no modulation---lv
  ME2 |= USPIE1;                       // Module enable
  U1CTL &= ~SWRST;                     // SPI enable SWRST = 0x01
}

void SPI_Disable(void)
{
  P5SEL &= ~(DIO | DCLK | LOCK_BIT);    // P5.1,2,3  I/O function is selected
  P5DIR &= ~(DIO | DCLK | LOCK_BIT);     //set SPI pin as input
  ME2 &= ~USPIE1;                      // Module disable
  U1CTL |= SWRST;                      // SPI disable, software reset enable, usart logic held in reset state.
}

// Function Transmits Character from RXTXData Buffer
void SPI_SendByte (unsigned char data)
{
  while (!(IFG2 & UTXIFG1));             // USART1 TX buffer ready?
  TXBUF1 = data;
}

// Function Receive Character from RXTXData Buffer
unsigned char SPI_ReceiveByte ( void )
{
  unsigned char data;
  while (!(IFG2 & URXIFG1));             // USART1 RX buffer ready?
  data = RXBUF1;
  return data;
}

void SPI_ResetBit (void)
{
  SPI_tmp_value = 0;
  SPI_tmp_index = 7;
```

```
}

// Function Transmits Character from RXTXData Buffer
void SPI_SendBit (unsigned char data)
{
 SPI_tmp_value += (data & 1) << SPI_tmp_index;
 SPI_tmp_index--;
 if (SPI_tmp_index < 0)
 {
  //UART_SendByte(SPI_tmp_value);
  while (!(IFG2 & UTXIFG1));              // USART1 TX buffer ready?
  TXBUF1 = SPI_tmp_value;
  SPI_tmp_index = 7;
  SPI_tmp_value = 0;
 }
}
```

A.8. SPI.h

```
#ifndef SPI_H
#define SPI_H
#define DIO      BIT2
#define DCLK     BIT3
#define LOCK_BIT  BIT1
void SPI_Init(void);
void SPI_Disable(void);
void SPI_SendByte (unsigned char data);
unsigned char SPI_ReceiveByte (void);
void SPI_ResetBit (void);
void SPI_SendBit (unsigned char data);
#endif
```

A.9. uart.c

```
#include <msp430.h>
#include "main.h"
#include "uart.h"
#include "SPI.h"

unsigned int RXTXData;
unsigned char BitCnt;
unsigned short uart_i;
unsigned char uart_crc;
unsigned char uart_crc1;
unsigned char uart_crc2;
unsigned char CRC( unsigned char *pDato, unsigned short dim)
{
  uart_crc = 0;
  for ( uart_i=0; uart_i<dim ; uart_i++ )
    uart_crc += pDato[uart_i];
  return uart_crc;
}

void printUART (unsigned char *message)
{
  int i;
  for (i=0; message[i]!=0; i++) UART_SendByte(message[i]);
}

void UART_Init(void)
{
  P3SEL |= 0x30;                        // P3.4,5 = USART0 TXD/RXD
  ME1 |= UTXE0 + URXE0;                   // Enabled USART0 TXD/RXD
  UCTL0 |= CHAR;                        // 8-bit character
  UTCTL0 |= SSEL1 + SSEL0 + URXSE;         // UCLK = SMCLK, start edge detect;                        // UCLK =
SMCLK
  UBR00 = (unsigned char)(FOSC/9600);      // 9600 bps
  UBR10 = (unsigned char)((FOSC/9600) >> 8);// 9600 bps
  UMCTL0 = 0x00;                        // no modulation
  UCTL0 &= ~SWRST;                        // Initialize USART state machine
}

// Function Transmits Character from RXTXData Buffer
void UART_SendByte (unsigned char data)
{
  while (!(IFG1 & UTXIFG0));              // USART0 TX buffer ready?
  TXBUF0 = data;
}

// Function Transmits Character from RXTXData Buffer
unsigned char UART_ReceiveByte ( void )
{
  unsigned char data;
```

```
 while (!(IFG1 & URXIFG0));              // USART0 RX buffer ready?
 data = RXBUF0;
 return data;
}

unsigned char UART_ReceivePacket(unsigned char *pDato)
{
 pDato[0] = UART_ReceiveByte();
 pDato[1] = UART_ReceiveByte();

 for ( uart_i = 2 ; uart_i < ((short)pDato[1]) + 2 ; uart_i++ )
  pDato[uart_i] = UART_ReceiveByte();
 //*dim = pDato[1];
 pDato[uart_i] = UART_ReceiveByte();
 uart_crc2 = CRC(pDato, ((short)(pDato[1])) + 2 );
 if (uart_crc2 == pDato[uart_i])
  return 1;
 else
  return 0;
}

unsigned char uart_SendPacket(unsigned char *pDato)
{
 for ( uart_i = 0 ; uart_i < pDato[1] + 2 ; uart_i++ )
  UART_SendByte(pDato[uart_i]);
 uart_crc = CRC(pDato, pDato[1] + 2);
 UART_SendByte(uart_crc);
 return uart_i;
}
```

A.10. uart.h

```
#define UART_H
#include <msp430x14x.h>
#define DATARATE 9600
#define RXD      32                 // RXD on P3.5
#define TXD      16                 // TXD on P3.4
#define Bitime_5  ((int)((float)FOSC/(1.91*DATARATE)))       // ~ 0.5 bit length + small adjustment
#define Bitime    ((int)(((float)(FOSC))/DATARATE))          // 8.6 us bit length ~ 115942 baud

void UART_SendByte (unsigned char data);
unsigned char UART_ReceiveByte (void);
void UART_Init(void);
void printUART (unsigned char *message);
unsigned char CRC( unsigned char *pDato, unsigned short dim);
unsigned char UART_ReceivePacket(unsigned char *pDato);
unsigned char uart_SendPacket(unsigned char *pDato);
#endif
```

A.11. timer.c

```
#include <msp430x14x.h>
#include "timer.h"
#include "uart.h"
#include "main.h"
short volatile *contatore;

void TIMER_SetupTimer_ms(short volatile *semaforo)
{
 contatore = semaforo;
 TACTL = 0;                    //stop the timer
 TAR = 0;                      //reset timer
 TACCR0 = (short)(FOSC/1000);            // 1 ms at 4 MhZ
 TACTL = TASSEL1 | MC_1 | TAIE;   //start!
 TACCTL0 = CCIE;               //enable interrupt
}

void TIMER_Wait_ms(short volatile semaforo)
{
 contatore = &semaforo;
 TACTL = 0;                    //stop the timer
 TAR = 0;                      //reset timer
 TACCR0 = (short)(FOSC/1000);            // 1 us at 4 MhZ
 TACTL = TASSEL1 | MC_1 | TAIE;   //start!
 TACCTL0 = CCIE;               //enable interrupt
 while(semaforo);
}

void TIMER_Wait_us(short volatile semaforo)
{
 contatore = &semaforo;
 TACTL = 0;                    //stop the timer
 TAR = 0;                      //reset timer
 TACCR0 = (short)(FOSC/1000000);           // 1 us at 4 MhZ
 TACTL = TASSEL1 | MC_1 | TAIE;   //start!
 TACCTL0 = CCIE;               //enable interrupt
 while(semaforo);
}
// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
 TACTL &= ~TAIFG;        //reset interrupt flag
 (*contatore)--;
 if ((*contatore) == 0)
 {
  TACCTL0 = 0;
  TACTL = 0;
 }
}
```

A.12. timer.h

```
#ifndef TIMER_H
#define TIMER_H
void TIMER_SetupTimer_ms(short volatile *semaforo);
void TIMER_Wait_ms(short volatile semaforo);
void TIMER_Wait_us(short volatile semaforo);
#endif
```

Appendice B

Tabella dei comandi e dei dati

Nome	Categoria	Dimensioni
GetSatellitePosition	Maneuver	Short
Deferred	Maneuver	Short
GetSatelliteAttidute	Maneuver	Short
GetSatelliteSpin	Maneuver	Short
Aim	Maneuver	Short
Track	Maneuver	Short
Deploy	Maneuver	Short
Deorbit	Maneuver	Short
Orbit	Maneuver	Short
Orientation	Maneuver	Short
Despin	Maneuver	Short
Script	Maneuver	Medium
ResetHW	Maintenance & ReconfigureSatellite	Short
ResetSW	Maintenance & ReconfigureSatellite	Short
UploadSW	Maintenance & ReconfigureSatellite	Large
Tiles_ExludeBattery	Maintenance & ReconfigureSatellite	Short
Tiles_Activate	Maintenance & ReconfigureSatellite	Short
Tiles_Deactivate	Maintenance & ReconfigureSatellite	Short
Tiles_ExcludeSensor	Maintenance & ReconfigureSatellite	Short
Update_HeartMagneticMap	Maintenance & ReconfigureSatellite	Medium
UpdateTLE	Maintenance & ReconfigureSatellite	Short
UpdateOn-boardClock	Maintenance & ReconfigureSatellite	Short
UpgradeTLC_SW	Maintenance & ReconfigureSatellite	Large
ReconfigureOBC	Maintenance & ReconfigureSatellite	Short
UpgradePMT_SW	Maintenance & ReconfigureSatellite	Large
RecoverAttidude	Maintenance &	Short

Short data <250 B
Medium data 251 B – 4 KB
Large Data > 4KB
Very large data 1 MB

	ReconfigureSatellite	
TurnOnOff_magnetic_coils	Maintenance & ReconfigureSatellite	Short
SetMechanicalProperties	Maintenance & ReconfigureSatellite	Short
SetSatelliteTime	Maintenance & ReconfigureSatellite	Short
DebugModes	Maintenance & ReconfigureSatellite	Short
SetClock	Maintenance & ReconfigureSatellite	Short
SensorCrossCheck	Monitoring	Short
DataBusTest	Monitoring	Short
MemoryTests	Monitoring	Very large
CollectingTelemetry	Monitoring	Large
BatteryLifeStatus	Monitoring	Short
StarSensorImage	Telemetry	Very large
Extended	Telemetry	Large
ConfigureHistory	Telemetry	Short
SafeState	Telemetry	Short
Telemetry History	Telemetry	Very large
BasicTelemetry	Telemetry	Medium
PayloadTelemetry	Payload	Medium
PayloadDataUplink	Payload	Medium
PayloadDataDownlink	Payload	Medium
DeliverCommand	Payload	Short
TurnOnOff	Payload	Short

Appendice C

Calcolo della durata del periodo di visibilità

Per ottenere un'orbita circolare intorno alla Terra di raggio R+h, percorsa alla velocità angolare ω, è richiesta un'accelerazione centripeta a_C pari a:

$$a_C = \varpi^2 (R+h) \tag{c.1}$$

Tale accelerazione è fornita dal campo gravitazionale che, alla distanza R+h, produce un'accelerazione pari a:

$$a_g(b) = g\frac{R^2}{(R+h)^2} \tag{c.2}$$

La condizione di equilibrio (orbita stazionaria) si ha per:

$$a_g = a_C \Leftrightarrow g\frac{R^2}{(R+h)^2} = \varpi^2 (R+h) \tag{c.3}$$

da cui si ricava l'espressione della velocità angolare:

$$\varpi = \sqrt{g\frac{R^2}{(R+h)^3}}\left[\frac{rad}{s}\right] \tag{c.4}$$

Dalla precedente si osserva come satelliti a orbite più alte sono caratterizzati da velocità angolari minori, cioè da tempi di rivoluzione maggiori.

Calcoliamo la precedente per i due valori estremi delle quote previste per AraMiS, $h_{\min} = 400Km$ e $h_{\max} = 1000Km$:

$$\varpi\big|_{h=400\,Km} = 1.1324 \cdot 10^{-3} rad/s$$

$$\varpi\big|_{h=1000\,Km} = 0.9970 \cdot 10^{-3} rad/s$$

Minimo angolo di elevazione all'orizzonte – Visibilità

Nella Figura c.1 per semplicità stiamo considerando che la Stazione di Terra sia posizionata sul piano orbitale. Dal punto di vista geometrico, la condizione minima per cui un satellite è visibile da una Stazione di Terra è β=0°. Tuttavia la propagazione delle onde radio quasi parallelamente al terreno è sorgente di anomalie quali multipath, scattering multiplo, diffrazioni (ray bending), che inficiano la qualità della ricezione dei segnali. Inoltre sono esaltati i fenomeni di bloccaggio dovuti sia alle caratteristiche morfologiche del territorio circostante la Stazione, sia agli artefatti umani specie se la stazione è ubicata in prossimità di centri urbani. Pertanto viene convenzionalmente fissato un valore minimo dell'angolo di elevazione sull'orizzonte. Questo valore può andare da 5° a 15° in dipendenza della situazione climatica e topografica locale. Noi assumiamo $\beta_{\min} = 7°$.

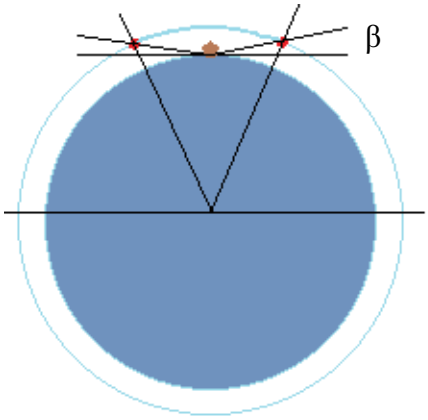


Figura c.1

Stima della durata del periodo di visibilità

In base a quanto detto, il satellite è visibile quando $7° \le \beta = 173°$, cioè quando percorre l'orbita tra i punti A e B in Figura c.1.

Il legame tra l'angolo θ e l'angolo β è dato dalla relazione:

$$\theta = 90° - \cos^{-1}\left[\frac{R}{R+h}\cos\beta\right] - \beta \qquad \text{(c.5)}$$

per cui ai valori di elevazione β per i quali assumiamo si abbia la visibilità, possiamo calcolare per le due orbite:

$76.05° \le \theta\big|_{h=400\,Km} \le 103.94°$

$66.07° \le \theta\big|_{h=1000\,Km} \le 113.92°$

L'angolo Δθ che corrisponde al tratto di orbita in cui il satellite è visibile, è pari a:

$\Delta\theta\big|_{h=400\,Km} = 103.94° - 76.05° = 27.89° = 486.77 \cdot 10^{-3}\,rad$

$\Delta\theta\big|_{h=1000\,Km} = 113.92° - 66.07° = 47.85° = 835.14 \cdot 10^{-3}\,rad$.

Il tempo impiegato per percorrere tali tratti di orbita, nei due casi, sarà pari a:

$$T_{vis}\big|_{h=400\,Km} = \frac{486.77 \cdot 10^{-3}\,rad}{1.1324 \cdot 10^{-3}\,rad/s} = 429.85s = 7.16\,\text{min}$$

$$T_{vis}\big|_{h=1000\,Km} = \frac{835.14 \cdot 10^{-3}\,rad}{0.9970 \cdot 10^{-3}\,rad/s} = 837.65s = 13.96\,\text{min}$$

Dalle precedenti si osserva come il satellite a orbite più elevate sia caratterizzato da un periodo di visibilità più lungo. Dobbiamo ricordare che tali valori sono ricavati sotto l'ipotesi che la stazione di Terrà si trovi sul piano orbitale. Benché le stazioni radioamatoriali in certi paesi abbiano una densità elevata, tale circostanza è improbabile ed è ragionevole ritenere che il tempo medio di visibilità equivalga al 70% del caso ottimo, per cui:

$T_{vis}\big|_{h=400\,Km} = 300.89s = 5.01\,\text{min}$

$T_{vis}\big|_{h=1000\,Km} = 586.35s = 9.77\,\text{min}$.

Bibliografia

[1] http://www.cubesat.org/

[2] http://polimage.polito.it/picpot

[3] S. Speretta, L. Reyneri, C. Sansoè, *Modular Architecture for satellites* (IAC 200)

[4] http://www.genso.org/

[5] http://www.esa.int/esaMI/Education/SEMKO03MDAF_0.html

[6] M. Davidoff, *The radio amateur's satellite handbook*

[7] Tanenbaum, *Reti di calcolatori* , 4 ed.

[8] http://www.tapr.org/

[9] Klofas, Anderson, *A survey of CubeSat Communication Systems* (California Polytechnic State University)

[10] TI Data Sheet, *CC1020 Low-Power RF Transceiver for Narrowband System*

[11] TI Data Sheet, MSP430x14x *Mixed Signal Microcontroller*, 2004

[12] Roberto Garello, "Lucidi del corso di Trasmissione Numerica", 2009

[13] AX.25 Link Access Protocol for Amateur Packet Radio Version 2.2, Revision: July 1998

[14] Wiley J. Larson, James R. Wertz, *Space Mission Analysis and Design*, 3 ed., Space Technology Library, 1999

[15] Couch L.W. II, *Fondamenti di telecomunicazioni*, Apogeo

[16] Stephen Horan, *Introduction to PCM Telemetering Systems*, 2 ed., Taylor&Francis Group