POLITECNICO DI TORINO

III Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Sistema di ricetrasmissione UHF per il satellite modulare AraMiS

Relatore:
Prof. Dante Del Corso

Candidato:
Ten. Co.Ing. Giuseppe Cosentino

Settembre 2008

Sommario

Il crescente interesse verso lo sviluppo dei nanosatelliti, che il mondo universitario ha sviluppato nel corso degli ultimi anni, ha portato nel 1999 a definire lo standard 'CubeSat', il cui obiettivo riguarda l'acquisizione di una metodologia base per l'approccio alla progettazione di tali sistemi, limitandone dimensioni e costi.

Il Politecnico di Torino nel 2006 terminò la realizzazione di un suo nanosatellite (PiCPoT), di dimensioni poco più grandi rispetto allo standard CubeSat, le cui finalità principali furono volte a testare il funzionamento dei COTS (componenti commerciali) e all'acquisizione d'immagini e parametri dall'ambiente spaziale; purtroppo, un problema idraulico del lanciatore durante il lancio, determinò il fallimento della missione. L'esperienza acquisita permise, comunque, di realizzare un altro progetto denominato 'ARAMIS' e di ovviare ad alcune carenze di PicPot, attraverso il concetto di modularità del satellite.

PicPot fu realizzato 'su misura' per la missione, ciò rendeva scarsamente riutilizzabile il progetto, aumentandone altresì i costi. In AraMiS, sono stati definiti dei moduli standard compatibili meccanicamente ed elettronicamente tra loro, che possono essere 'assemblati' nelle quantità e nelle modalità richieste dalla singola missione, determinando un abbattimento dei costi di progettazione e sviluppo, non indifferente.

Sono stati realizzati un modulo della gestione della potenza e uno di telecomunicazioni:

- Power Management tile: Moduli che si occupano di ricavare (attraverso l'energia solare), l'energia elettrica necessaria per ricaricare le batterie, di generare le alimentazioni per la circuiteria di bordo e di fornire l'assetto desiderato al satellite, attraverso l'utilizzo di altri moduli Power Management.

  Tali moduli saranno montati sulla parte esterna del satellite, poiché su di essi risiedono i pannelli solari utilizzati per ricavare l'energia; inoltre fungeranno da struttura portante per il satellite.

  Questi moduli interagiscono fra loro attraverso bus dati e di potenza, ciò permette di stabilire, secondo lo stato di carica delle batterie e della loro temperatura, quale ricaricare e con quale pannello (tra quelli illuminati), eseguire tale ricarica. La presenza di più moduli Power Management con attuatori d'assetto, permette al satellite di ruotare in ogni direzione.

- Telecommunication tile: Moduli che rendono possibile la comunicazione bidirezionale tra satellite e stazione di terra, tramite due distinti canali di comunicazione.

  Un canale di comunicazione, operante alla frequenza di 437 MHz, è stato pensato per permetterne l'utilizzo ai radioamatori, che possono ricevere il beacon del satellite; mentre il secondo canale lavora alla frequenza di 2.4GHz.

  La comunicazione è gestita da un microcontrollore a bordo di tale modulo, che si occupa della gestione del protocollo di comunicazione, ma anche dell'interpretazione dei comandi d'assetto provenienti dalla stazione di terra, in modo da fornire ai singoli moduli Power Management, il comando riguardante l'assetto da generare.

Le frequenze del sistema di comunicazione, sono state selezionate nella banda dedicata alle comunicazioni satellitari amatoriali (437MHz e 2.4 GHz). Il protocollo

del progetto prevede che i due canali di comunicazione selezionati siano half-duplex e mutuamente esclusivi, in altre parole, la trasmissione e la ricezione devono essere alternate (lo stesso vale per le due frequenze). In virtù di tali considerazioni, la scheda Tx/Rx può essere considerata suddivisa in due blocchi indipendenti, funzionanti alle due diverse frequenze.

L'obiettivo che ci si è posto in questa tesi, è stato il test funzionale e la programmazione della scheda Tx/Rx a 437MHz, che svolge la funzione di trasmissione e ricezione dei segnali tra ARAMIS e la stazione di terra, progettata da un precedente tesista.

Il ricetrasmettitore, scelto per la scheda a 437MHz, è un integrato della Chipcon CC1020; mentre il microcontrollore utilizzato è un MSP430F1121A (sostituito in seguito con un MSP430F149).

Al fine di poter stabilire una corretta comunicazione fra microcontrollore e transceiver, sono state apportate varie modifiche (sia hardware, sia software), alla scheda precedentemente realizzata; successivamente si è passato al collaudo della scheda 'modificata' e alla realizzazione di un nuovo PCB del sistema, attraverso il software della 'Mentor Graphics'.

Il lavoro è suddiviso nei seguenti capitoli:

- Capitolo 1: descrive le architetture PicPot, ARAMIS e alcuni progetti realizzati da altre università.

- Capitolo 2: Descrive il progetto del sottosistema di comunicazione satellite-terra, la sua struttura e i vincoli ambientali a cui è sottoposto.

- Capitolo 3: Descrive le prove e le verifiche funzionali svolte sulla scheda (prima e dopo le varie modifiche apportate).

- Capitolo 4: Descrive i protocolli di comunicazione adoperati.

- Capitolo 5: Descrive i software e la realizzazione delle funzioni.
- Capitolo 6: Descrive la realizzazione del PCB e il collaudo della scheda.

Indice

Sommario II

1 Introduzione **1**

1.1 Progetto PicPoT 2

1.2 Progetto AraMiS 4

2 Progetto del sottosistema di comunicazione a RF del satellite **6**

2.1 GENSO (progetto ESA) 6

2.2 Scelta della frequenza di lavoro 7

2.3 Link Budget 8

2.3.1 Dimensionamento della potenza in trasmissione del satellite a 437MHz(verifica) 11

2.3.2 Valutazione della qualità di trasmissione a 437MHz 13

2.4 Scelta dei componenti 14

2.4.1 Scelta del transceiver 14

2.4.2 Scelta del microcontrollore e interfacciamento col CC1020 . . . 17

2.5 Schema a blocchi 20

3 Verifiche funzionali sulla scheda **22**

3.1 Problemi Hardware 22

3.2 Identificazioni delle soluzioni 23

4 Protocolli di comunicazione **29**

4.1 Il protocollo AX-25 29

4.1.1 Introduzione 29

4.1.2 Definizione dei campi 30

4.1.3 Procedure di trasmissione 34

4.1.4 Implementazione software 35

5 Programmazione della scheda **40**

5.1 Interfaccia CC1020-Msp430F149 40

5.2 Struttura del Software 45

5.3 Descrizione delle funzioni create 50

5.3.1 CC1020.c 50

5.3.2 Uart.c 53

5.3.3 SPI.c 54

5.3.4 AX.25.c 54

5.3.5 Timer.c 55

6 Realizzazione del nuovo circuito stampato **56**

6.1 Progetto dello schema elettrico 57

6.2 Realizzazione PCB 57

6.3 Collaudo 59

7 Conclusioni **65**

A Schemi elettrici **67**

B PCB Layout **77**

C Source Files **83**

C.1 main.c 84

C.2 main.h 89
C.3 CC1020.c 90
C.4 CC1020.h 105
C.5 Uart.c 111
C.6 Uart.h 116
C.7 SPI.c 117
C.8 SPI.h 120
C.9 AX25.c 121
C.10 AX25.h 136
C.11 timer.c 137
C.12 timer.h 138

Bibliografia **139**

Elenco delle figure

1.1 Esterno di PiCPoT 3

2.1 Una delle bande allocate dalla IARU per le comunicazioni satellitari . 8

2.2 Distanza tra Aramis e stazione di terra 10

2.3 CC1020 15

2.4 *Schema a blocchi semplificato del CC1020* 16

2.5 Interfaccia del CC1020 con un microcontrollore 17

2.6 MSP430F149 18

2.7 Schema a Blocchi del microcontrollore MSP430F149 19

2.8 Schema a Blocchi della scheda Tx/Rx a 437MHz 21

3.1 Prime modifiche della scheda 23

3.2 Scheda Tx/Rx dopo la sostituzione del microcontrollore 25

3.3 Partitore R7-L14 27

4.1 *Pacchetto U o S* 30

4.2 *Pacchetto I* 30

4.3 *Definizioni di PID* 32

4.4 *Bit stuffing* 35

4.5 *CRC-hardware* 36

5.1 Segnali che gestiscono la comunicazione tra CC1020 e MSP430F149 in lettura e scrittura 42

5.2 Schema della comunicazione Ground Station/scheda in trasmissione e ricezione 43
5.3 Schema per lo sviluppo del software 44
5.4 Flusso delle funzioni chiamate nel main 46
5.5 Flow Chart del ciclo di attesa dei comandi 47
5.6 Diagramma di flusso del blocco Send 48
5.7 Diagramma di flusso del blocco receive 49
5.8 Diagramma di flusso del blocco portante 49
5.9 Settaggio dei registri tramite il programma SmartrfStudio 51
6.1 Comando che consente il riscontro di errori nel progetto dello schema elettrico 58
6.2 Comando Compile CDB 58
6.3 Segnale trasmesso dal CC1020, visto all'analizzatore di spettro 60
6.4 Comando Send 62
6.5 Yaesu e TNC 63
6.6 Segnale trasmesso dal CC1020, ricevuto dalla radio Yaesu, elaborato dal TNC e poi inviato alla Ground Station 64
A.1 Schema elettrico del CC1020 68
A.2 Schema elettrico del microcontrollore MSP430F149 69
A.3 Schema elettrico del regolatore di tensione LM317 70
A.4 Schema elettrico del Switch d'antenna 71
A.5 Blocco rappresentante lo schema elettrico del CC1020 72
A.6 Blocco rappresentante lo schema elettrico del microcontrollore MSP30F149 73
A.7 Blocco rappresentante lo schema elettrico del regolatore di tensione LM317 74
A.8 Blocco rappresentante lo schema elettrico del switch d'antenna 75

A.9 Interconnessione dei blocchi realizzati, rappresentante lo schema elettrico della scheda 76
B.1 Generazione del PCB 78
B.2 Top del PCB 79
B.3 Bottom del PCB 80
B.4 Top file Gerber 81
B.5 Bottom del file Gerber 82

Capitolo 1

Introduzione

Negli ultimi anni diverse università, tra le quali il Politecnico di Torino (e i suoi diversi dipartimenti), si sono impegnate nello sviluppo e nella realizzazione di progetti per il lancio in orbita di picosatelliti. Gli obiettivi principali di tali progetti, riguardano la possibilità di realizzare un satellite attraverso l'impiego di componenti a basso costo ed in grado di operare correttamente nelle situazioni estreme di orbita spaziale. La realizzazione della maggior parte tali progetti, è avvenuta attenendosi allo standard CubeSat[1] (satellite di forma cubica) sviluppato nel 2001 dal Professor Robert Twiggs, docente della Stanford University (USA), in collaborazione con lo Space Systems Development Laboratory (SSDL) della Stanford University e la California Polytechnic State University.

Tale standard definisce le dimensioni dei picosatelliti che devono avere 10 cm di lato e massa inferiore a 1 kg. La struttura è definita in funzione all'adattamento al lanciatore POD (Picosatellite Orbital Deployer).

Alcuni esempi di satelliti già in orbita, sono AAU-Cubesat[2], CanX-1[2], NCube[2], UWE-1[2] e UNISAT[2], realizzati rispettivamente delle università in Danimarca, Canada, Norvegia, Germania e Italia.

1.1 Progetto PicPoT

Nell'Autunno del 2003, nacque il progetto 'Piccolo Cubo del Politecnico di Torino' (PicPoT), con il fine di costruire un nanosatellite, nell'arco di un anno, a scopo educativo e di ricerca. Il progetto fu basato sui seguenti requisiti:

- Forma cubica con lato 13 cm;
- massa inferiore 5 kg;
- Potenza media non superiore a 1,5 W;
- Almeno 90 giorni di vita;
- Utilizzo di componenti COTS nello spazio;
- Orbita LEO (Low Earth Orbit, fra i 600 e gli 800 km di altitudine);
- Compatibilità con il lanciatore POD;

Le funzioni principali del satellite riguardavano l'acquisizione di misure di temperatura e illuminamento, scattare fotografie e trasmettere i dati raccolti alla stazione di terra.

Esternamente il satellite si presentava come in figura 1.1

Su cinque delle sue sei facce esterne, erano presenti celle solari utilizzate per convertire l'energia solare in energia elettrica; sulla sesta faccia erano presenti due antenne (437MHz e 2.4GHz), tre fotocamere, due kill switch (per 'spegnere' il satellite durante il lancio) e un connettore di test, per verificare il corretto funzionamento dell'elettronica di bordo.

Internamente era alimentato da sei gruppi di batterie ricaricabili disposte fra i pannelli solari e le schede elettroniche. Su PicPoT erano presenti tre processori:

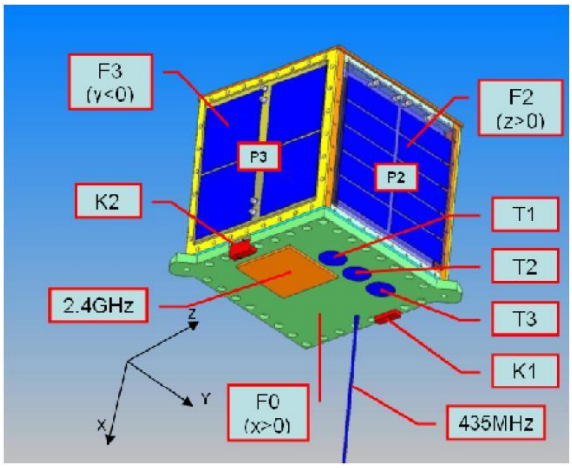

Figura 1.1. Esterno di PiCPoT

- ProcA: per la gestione di bordo, associato al canale di comunicazione a 437MHz, frequenza di clock a 11 MHz e tensione di alimentazione a 3.3 V.

- ProcB: per la gestione di bordo associato al canale di comunicazione a 2.4 GHz. E' stato utilizzato un MSP430 a basso consumo, con frequenza di clock a 4 MHz e tensione di alimentazione di 3.3V

- Payload: per l'acquisizione delle immagini delle tre telecamere, che successivamente sono inviate a terra tramite ProcA e ProcB.

I due processori ProcA e ProcB erano indipendenti fra loro e condividevano tutti i circuiti di condizionamento dei sensori, ma ciascuno disponeva di un suo multiplexer.

Tutti e tre i processori erano alimentati periodicamente ed eseguivano sempre lo stesso codice.

La comunicazione a terra avveniva tramite due canali, entrambi in banda amatoriale con protocollo APRS.

1.2 Progetto AraMiS

AraMiS rappresenta l'evoluzione del precedente progetto PicPoT; gli obiettivi rimangono inalterati rispetto al suo predecessore, ma la sua funzionalità ha subito un profondo mutamento, dovuto all'introduzione del concetto di 'modularità satellitare'.

PicPoT era stato progettato ad hoc per la missione, questo era un difetto che incideva particolarmente anche sui costi e sui tempi di progettazione.

Il nuovo concetto introdotto, prevede la realizzazione di moduli standard, che possono essere messi insieme fra loro secondo l'esigenza del caso.

AraMiS prevede due moduli standard:

- Power Management

  Provvede a gestire le alimentazioni dell'elettronica di bordo e al controllo d'assetto. In esso sono incorporate le funzionalità dei blocchi di Solar Panel, Power Supply, Batteries e Power Switch presenti anche su PicPoT, ma con l'aggiunta di un controllo attivo dell'assetto del satellite.

- Telecommunication

  Questo modulo comprende quanto occorre per consentire la comunicazione satellite-stazione di terra e viceversa, prelevando e immettendo i dati e i comandi sul bus di sistema; inoltre interpreta, ricavando a sua volta, i comandi d'assetto per i singoli moduli di Power Management.

I vantaggi, ottenuti attraverso la modularità, riguardano i costi di progettazione (ridotti, poiché diluiti su più missioni) e la possibilità di variare configurazione secondo il tipo di missione.

Aumentando il numero di moduli Power Management, o il numero dei canali di comunicazione (aggiungendo moduli Telecommunication), è possibile inoltre ottenere satelliti di dimensioni diverse e differenti prestazioni, in termini di energia disponibile e capacità di comunicazione con le stazioni di terra.

I due moduli Power Management e Telecommunication, insieme ai moduli Payload e Ground Station, rappresentano i macrocomponenti del satellite.

Capitolo 2

Progetto del sottosistema di comunicazione a RF del satellite

Per il satellite AraMiS, è previsto l'utilizzo di almeno due bande (437MHz e 2.4GHz), ma in questa tesi ci si occuperà della banda amatoriale a 437MHz.

In questo capitolo saranno discusse le scelte dei componenti per la realizzazione del sottosistema di comunicazione in oggetto, le modalità di utilizzo e soprattutto le finalità. In primis, però, sarà trattato in linea generale e breve, quella che rappresenta la filosofia del progetto dei nanosatelliti a scopo didattico, ovvero la filosofia del progetto GENSO.

2.1 GENSO (progetto ESA)

Il progetto GENSO (Global Educational Network for Satellite Operations), è organizzato e coordinato dal dipartimento ESA. Il suo obiettivo è perfezionare le conoscenze sui satelliti, attraverso lo sviluppo di varie applicazioni che permettono di stabilire una comunicazione fra satellite e stazione di terra, (Ground Station).

In diverse località, sono state realizzate varie stazioni di terra e sono tuttora in orbita diversi satelliti con fini didattici. Spesso una stazione di terra può comunicare con diversi satelliti senza particolari problemi e i dati delle missioni possono essere controllati tramite internet.

A partire da questa premessa, nasce la filosofia GENSO, che in virtù di tale situazione, ha realizzato una forma di cooperazione e, cercando di rendere i processi più automatici possibile al fine di ridurre al minimo l'intervento umano, permette di mettere insieme le limitate risorse economiche di cui si dispone, distribuirle e sfruttarle al meglio.

Oggi un gran numero di stazioni di terra (e quasi tutti i satelliti universitari), partecipano a questa cooperazione (difatti i partecipanti al progetto GENSO, sono per lo più radioamatori e università), utilizzando bande di frequenze radioamatoriali tipicamente UHF o VHF, rientranti in uno standard che definisce protocolli dei dati, baud rates e le configurazioni delle stazioni di terra.

Il progetto GENSO, mira a fornire un HAL (Hardware Abstraction Layer), che consiste in una standardizzazione del linguaggio per i controlli delle stazioni di terra e dei dati inviati e ricevuti. Tra gli obiettivi futuri, inoltre, vi è la volontà definire una libreria di 'drivers', che facciano da interfacce tra i componenti hardware specifici e gli HAL.

2.2 Scelta della frequenza di lavoro

Il punto di partenza per la progettazione del sistema di comunicazione di PicPoT e successivamente di AraMiS, è stata la scelta della frequenza di lavoro, proprio sulla base della filosofia GENSO. Partendo dal presupposto, che condividere i dati del satellite con i radioamatori di tutto il mondo, avrebbe comportato l'enorme vantaggio di poter identificare i parametri orbitali del satellite subito dopo il lancio, si

è voluto realizzare il sistema in maniera tale che potesse lavorare anche su frequenze da radioamatori. In virtù di ciò, è stato consultato il sito della IARU (International Amateur Radio Union), per verificare le frequenze allocate, per le comunicazioni satellitari amatoriali.

Tra le bande allocate, era presente la banda 435-438MHz, che è stata scelta per PicPot e successivamente per AraMiS.

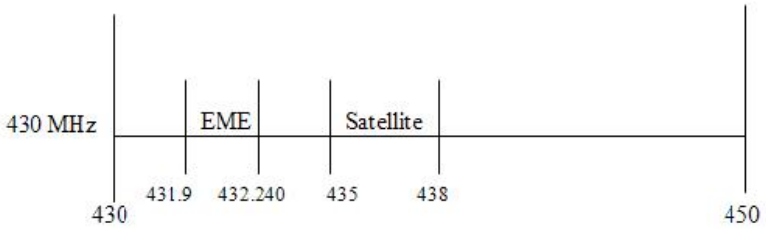

Figura 2.1. Una delle bande allocate dalla IARU per le comunicazioni satellitari

2.3 Link Budget

In quest'applicazione, i calcoli del Link Budget e la scelta dei componenti, sono fortemente intrecciati.

Inizialmente viene verificato il Link Budget, utilizzando alcuni parametri legati al componente che sarà utilizzato (Chipcon CC1020).

AraMiS utilizzerà la stessa frequenza e orbita di PiCPoT, pertanto il Link Budget, utilizzato per la valutazione dei parametri trasmissivi di tratta, restituirà gli stessi valori ottenuti precedentemente.

I valori ricavati sono stati calcolati dall'equazione della propagazione nello spazio libero:

$$Pr = PtGt\frac{1}{(\frac{4\pi D}{\lambda})^2}Gr = PtGt\frac{1}{\alpha o}Gr \tag{2.1}$$

In cui:

- Pr e Pt rappresentano potenza trasmessa e ricevuta;
- Gr e Gt rappresentano i guadagni rispettivamente dell'antenna in ricezione e in trasmissione;
- $\frac{1}{\alpha o}$ rappresenta l'attenuazione di propagazione;

Riscrivendo l'equazione in forma logaritmica, si ottiene:

$$Pr|_{dBm} = Pt|_{dBm} + Gt|_{dB} - \alpha o|_{dBm} + Gr|_{dB} \tag{2.2}$$

da cui si ricava:

$$\alpha o|_{dBm} = 32.45 + 20log(\frac{D}{km}) + 20log(\frac{f}{MHz}) \tag{2.3}$$

Sostituendo la 1.3 nella 1.2 e riscrivendola in funzione di Pt, si ottiene:

$$Pr|_{dBm} = Pt|_{dBm} + Gt|_{dB} + Gr|_{dB} - 32.45 - 20log(\frac{D}{km}) - 20log(\frac{f}{MHz}) \tag{2.4}$$

Supponendo che il satellite si trovi in un'orbita perfettamente circolare all'altezza di 600 km e che la terra sia perfettamente sferica, con un raggio di 6378 km, la

distanza 'D' del satellite all'orizzonte (Fig.2.2), rispetto alla stazione di terra, si calcola tramite teorema di Pitagora:

$$D = \sqrt{2Rh + h^2} = 2830.830km \tag{2.5}$$

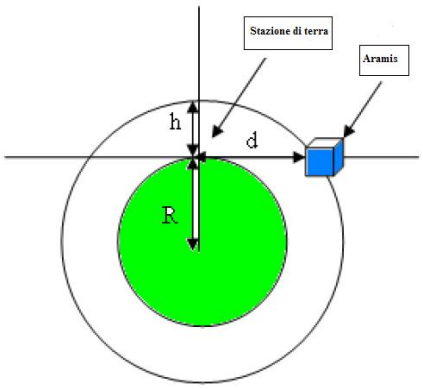

Figura 2.2. Distanza tra Aramis e stazione di terra

Una volta ottenuto il risultato di tale distanza, è possibile calcolare il valore dell'attenuazione di propagazione, alla frequenza di 437 MHz:

$$\alpha o|_{dBm} = 32.45 + 20log(\frac{2830.830km}{km}) + 20log(\frac{437.540MHz}{MHz}) = 154.31|_{dBm} \tag{2.6}$$

2.3.1 Dimensionamento della potenza in trasmissione del satellite a 437MHz(verifica)

In tabella 2.1 sono elencati i dati noti, per il dimensionamento della potenza, intorno alla frequenza di 435 MHz. I valori dei guadagni sono riferiti a un'antenna a dipolo sul satellite e a un'antenna ad elica con polarizzazione circolare, sulla stazione di terra.

Parametro	Valore	Unità
Frequenza di lavoro	f=437.540	MHz
Attenuazione	α=154.31	dBm
Guadagno in trasmissione	Gt=2	dB
Guadagno in ricezione	Gr=12	dB
IF Bandwidth	Bif=175	kHz
Separazione di frequenza	Δf=6	kHz
Cifra di rumore	NF=12	dB

Tabella 2.1. Parametri elettrici

La sensitivity al ricevitore, può essere calcolata come segue:

$$S|_{dBm} = N|_{dBm} + NF|_{dB} + SNR|_{dB} \tag{2.7}$$

N è la cifra di rumore associata ad un ricevitore ideale e vale:

$$N = kTB = 1.38 * 10^{-23} J/K * 290K * 12 * 10^3 Hz = -133.2dBm \tag{2.8}$$

Il rapporto segnale rumore SNR:

$$SNR = \frac{Eb}{No} * \frac{R}{Bif} \tag{2.9}$$

In cui il rapporto $\frac{Eb}{No}$, dipende dal tipo di modulazione utilizzato e dal BER. Scegliendo una modulazione di tipo FSK non coerente e un $BER = 10^{-6}$, si ottiene $\frac{Eb}{No} \cong 14dB$ e $SNR = 1.4dB$

Sostituendo tali dati, nell'equazione 2.7, si ottiene:

$$S|_{dBm} = N|_{dBm} + NF|_{dB} + SNR|_{dB} = -119.8dBm \tag{2.10}$$

Parametro	Valore
N	$-132.2dBm$
$Nf[CC1020]$	$+12dBm$
SNR con $BER = 10^{-6}$	$+14dBm$
S	$-119.8dBm$

Tabella 2.2. Calcolo della sensitivity

La potenza in trasmissione necessaria per soddisfare la sensitivity richiesta, si calcola con la formula ricavata precedentemente, in cui si sostituisce al termine Pr, il valore di $S|_{dBm}$:

$$Pt|_{dBm} = S|_{dBm} - Gt|_{dB} - Gr|_{dB} + 154.31_{dBm} = 20.51_{dBm} = 13.12mW \tag{2.11}$$

Se si considera la potenza $Pt = 33dBm$ come dato noto e si calcola tramite il link budget la rispettiva potenza in ricezione, confrontandola con la sensitivity richiesta, si ottiene:

$$Pr|_{dBm} = Pt|_{dBm} + Gt|_{dB} + Gr|_{dB} - 154.31_{dBm} = -107.31_{dBm} \tag{2.12}$$

Ovvero si ha un margine di rumore accettabile (circa 12 dB).

2.3.2 Valutazione della qualità di trasmissione a 437MHz

Il parametro che consente la valutazione della qualità di trasmissione, è il BER (Bit Error Rate).

$$BER = \frac{1}{2} e^{\frac{-1}{2}\frac{Eb}{No}} \tag{2.13}$$

In cui:

- Eb è l'energia associata ad ogni bit;
- $No = KTop$ è la densità spettrale di rumore;
- il rapporto $\frac{Eb}{No} = \eta = \gamma|_{dBHz} - 10logRb$;
- il parametro $\gamma = \frac{Pr}{No} = \frac{Pr}{KTop}|_{dBHz} = Pr|_{dBW} - 10log(k) - 10log(Top)$, rappresenta il rapporto Segnale/Rumore.

Se prendiamo in considerazione i valori precedentemente ottenuti, ipotizzando una temperatura operativa del ricevitore (Top)di 450K, abbiamo:

- $Top = 450K$;
- $Pr = -107.31dBm = -137.31dBW$;
- $k = 1.38 * 10^{-23} = 228.6dB$(costante di Boltzmann);

dunque, è possibile calcolare il rapporto segnale /Rumore $\gamma|_{dBHz}$:

$$\gamma|_{dBHz} = Pr|_{dBW} - 10log(k) - 10log(Top) = 64.76dBHz \tag{2.14}$$

Se ora consideriamo la velocità di trasmissione $Rb = 9.6kbps$, possiamo calcolare il parametro η:

$$\eta|_{dB} = \gamma|_{dBHz} - 10logRb = 25dB \tag{2.15}$$

e scegliendo una modulazione FSK non coerente, tenendo conto di un margine di 8dBm, otteniamo il BER:

$$BER = \frac{1}{2}e^{\frac{-1}{2}\frac{Eb}{No}} = 8.35 * 10^{-7} \tag{2.16}$$

2.4 Scelta dei componenti

2.4.1 Scelta del transceiver

Una volta effettuati i calcoli e le successive considerazioni, era stata rivolta l'attenzione verso la scelta dei componenti; tale fase è di ovvia importanza per la realizzazione di un qualunque sistema di comunicazione.

In primis è stata compiuta un'analisi dei criteri che tali componenti dovevano rispettare:

- Banda di frequenza compresa tra [420-450]MHz;

- Minima tensione di alimentazione;
- Minimo consumo di corrente;
- Massima potenza in uscita;
- Minimo bit rate;
- Possibilità di un eventuale interfacciamento con PA esterno;

Dopo una ricerca effettuata sul web ed un confronto fra i vari componenti trovati, la scelta si era orientata sul Chipcon CC1020, poiché oltre a rispettare i vari requisiti, permetteva il suo utilizzo in applicazioni narrowband, con larghezza di banda di [12.5-25]kHz.

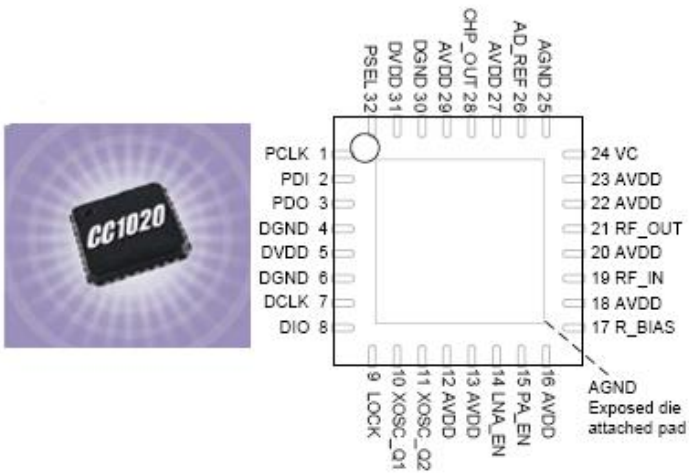

Figura 2.3. CC1020

In particolare le caratteristiche principali sono:

- Banda di frequenza compresa tra [402-470]MHz e [804-940]MHz;
- Alta sensitivity (-118dBm per una larghezza di banda a 12.5kHz);

- Richiede pochi componenti esterni;
- Basso consumo di corrente (19.9 mA in ricezione);
- Tensione di alimentazione da 2.3V a 3.6V;
- Data Rate sopra i 153.6 kbps;
- Modulazione FSK/GFSK e OOK;
- Adatto per sistemi di frequency hopping;

I parametri operativi principali del CC1020, possono essere programmati attraverso un bus seriale, rendendolo un ricetrasmettitore molto flessibile e facile da usare.

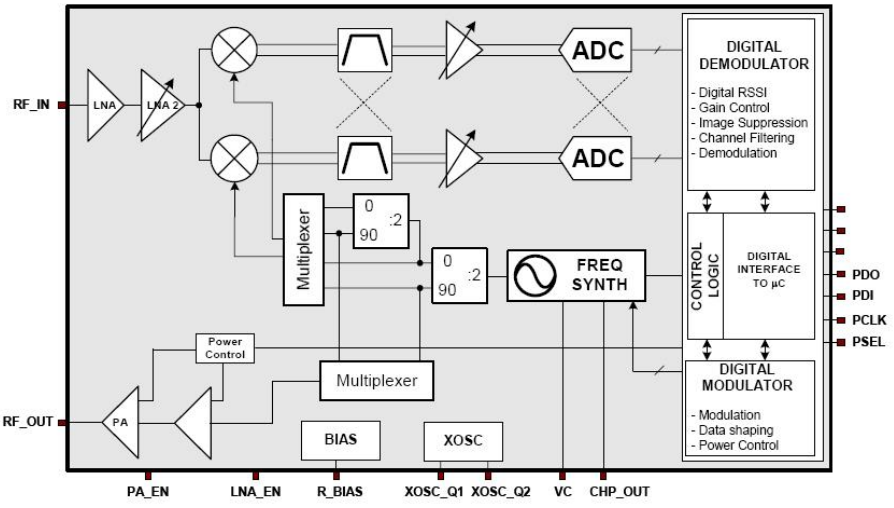

Figura 2.4. *Schema a blocchi semplificato del CC1020*

Dallo schema in figura 2.4, si nota chiaramente, che la radiofrequenza ricevuta viene amplificata con due LNA (il primo a guadagno fisso, il secondo a guadagno che varia secondo la potenza del segnale d'ingresso), per poi entrare in un demodulatore I/Q ad una frequenza intermedia (IF), per poi entrare nel convertitore A/D.

I dati demodulati e digitalizzati, vengono restituiti sul pin DIO, mentre il clock di sincronismo sui dati è dato sul pin DCLK.

In trasmissione il segnale passa dal pin DIO al modulatore digitale e dopo essere stato modulato, entra in un PA.

2.4.2 Scelta del microcontrollore e interfacciamento col CC1020

L'idea originariamente studiata, prevedeva l'utilizzo di un transceiver con microcontrollore integrato al fine di realizzare un circuito il più semplice possibile; ma al momento della scelta dei componenti, non sono stati trovati in commercio integrati del genere, che lavorassero alla frequenza desiderata. Avendo dovuto escludere questa possibilità, si sono scelti di due componenti.

L'interfacciamento del CC1020, prevede l'utilizzo di 7 pin, di cui 4 servono per la programmazione dello stesso (PDI, PDO, PCLK, PSEL) e gli altri 3 (DIO, DCLK, LOCK), si occupano rispettivamente, della trasmissione e ricezione dei dati, della loro sincronizzazione e del monitoraggio (opzionale), del segnale di lock del PLL.

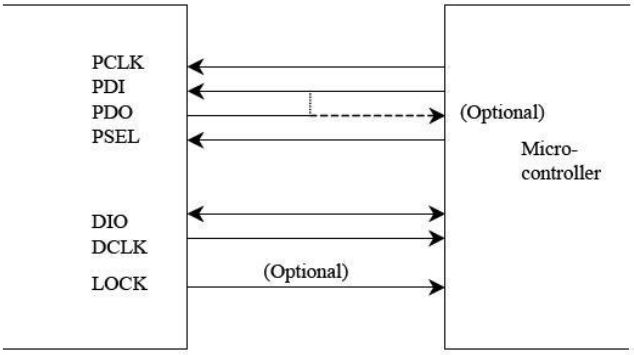

Figura 2.5. Interfaccia del CC1020 con un microcontrollore

Dopo varie ricerche, è stato scelto il microcontrollore della Texas Instrument MSP430F1121A, soprattutto per le sue dimensioni ridotte e i bassi consumi.

Durante la fase di test funzionale della scheda, questo tipo di componente si è rivelato essere parecchio inaffidabile, a causa dell'incompatibilità col compilatore 'IAR', descritta nel capitolo precedente.

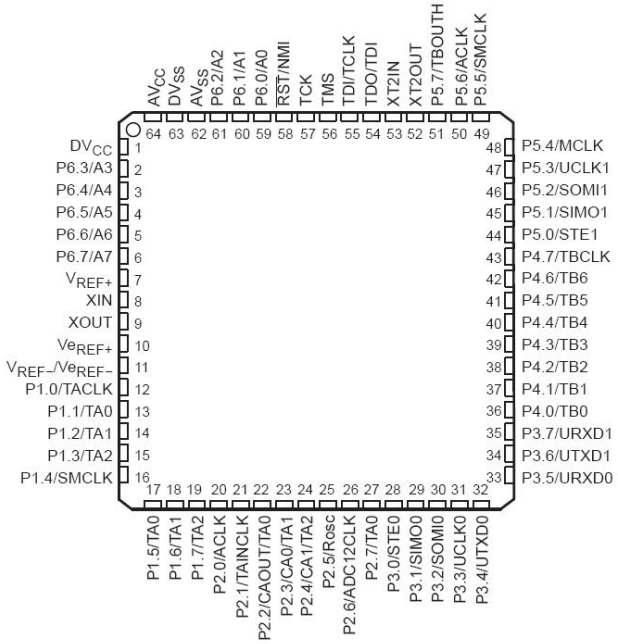

Figura 2.6. MSP430F149

La serie MSP430F1xx della Texas Instrument a basso consumo è caratterizzata da una CPU a 16 bit (altamente trasparente all'applicazione), 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.

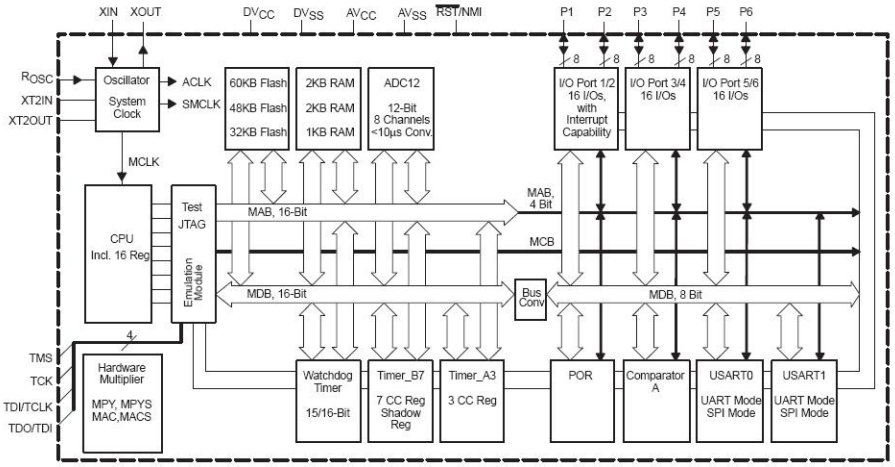

Figura 2.7. Schema a Blocchi del microcontrollore MSP430F149

Transceiver e microcontrollore hanno bisogno di un certo numero di componenti passivi, le cui caratteristiche sono elencate nei datasheet degli integrati scelti e devono essere categoricamente rispettate, affinché il tutto funzioni correttamente.

2.5 Schema a blocchi

La scheda Tx/Rx effettua uno scambio di dati con la stazione di terra, nella banda dedicata alle comunicazioni satellitari amatoriali (nella fattispecie 437MHz). I canali di comunicazione sono half-duplex e mutuamente esclusivi, in altre parole, la trasmissione e la ricezione sono alternate. Il segnale trasmesso dalla stazione di terra viene ricevuto dall'antenna del satellite, passa attraverso il relay e transita direttamente nel pin 'RFin' del CC1020; dopodiché viene demodulato e convertito in segnale digitale, per poi essere inviato al microcontrollore che a sua volta lo invierà alla scheda 'ProcA', che si occuperà dell'elaborazione dei dati. Per attivare la trasmissione dei dati, la scheda 'ProcA' invia tramite SPI, al microcontrollore, i bytes da trasmettere. Il microcontrollore configura il transceiver per abilitarlo alla trasmissione e gli invia i dati. Il CC1020 possiede internamente un PA, ma non è sufficientemente in grado di amplificare il segnale da trasmettere, per cui, tra gli obiettivi futuri, è prevista l'introduzione di un PA esterno di almeno 1W posto prima del commutatore d'antenna. Quando il CC1020 è pronto a trasmettere, invia un segnale di abilitazione (PAen) che, attraverso un pMos, permette al relay di commutare; tale segnale di abilitazione, sarà adoperato contestualmente a ciò, per l'abilitazione del PA esterno. Il relay utilizzato è di tipo single side stable, per cui è necessario ricordare, che questo deve essere alimentato correttamente, per tutta la durata della trasmissione.

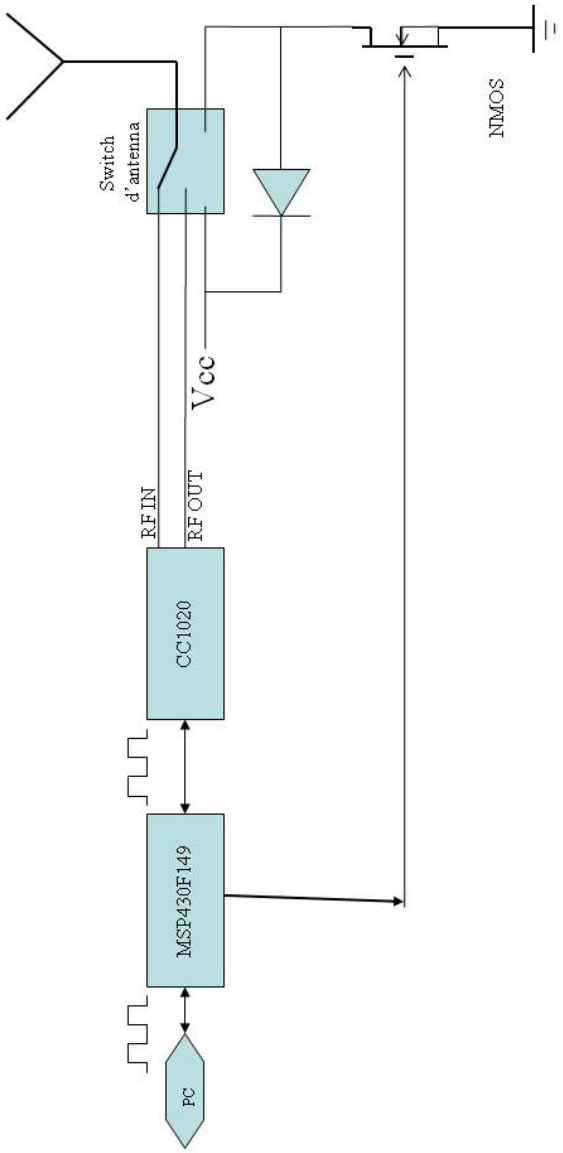

Figura 2.8. Schema a Blocchi della scheda Tx/Rx a 437MHz

Capitolo 3

Verifiche funzionali sulla scheda

La fase preliminare del lavoro, nonché il punto di partenza per la realizzazione del sistema di comunicazione in esame, è stato il test funzionale della scheda Tx/Rx a 437MHz, realizzata da un precedente tesista.

Le prove di laboratorio eseguite, hanno dato modo di riscontrare diversi malfunzionamenti, di varia natura, la cui graduale risoluzione ha permesso di ottenerne una versione nuova e funzionante.

Nel paragrafo successivo saranno illustrate le problematiche presentatesi e le successive risoluzioni.

3.1 Problemi Hardware

Come già accennato, durante il test funzionale della scheda, sono stati riscontrati e risolti nell'ordine che segue, diversi problemi di tipo hardware e software:

1. Mancata comunicazione fra micro-controllore MSP4430F1121A e transceiver CC1020;

2. Difficoltà nel reset del Transceiver CC1020;

3. Consumi eccessivi di corrente;

4. Difficoltà nell'aggancio del PLL del CC1020;

5. Mancata commutazione del relay d'antenna da ricezione a trasmissione;

6. Potenza del segnale in uscita pari a -45 dBm (su 10 dBm previsti).

La risoluzione di tali malfunzionamenti, riscontrati in questa fase, è avvenuta per passi successivi tramite un attento studio dello schema elettrico, dei componenti e del software di cui si disponeva.

3.2 Identificazioni delle soluzioni

Il primo passo ha riguardato la valutazione della correttezza delle connessioni tra MSP430F1121A/JTAG e successivamente quelle del CC1020/MSP430F1121A; in questa fase sono state apportate alcune modifiche circuitali (effettuate tagliando alcune piste e connettendole diversamente), che hanno consentito sia di effettuare la programmazione del microcontrollore, sia di stabilire la comunicazione tra microcontrollore e transceiver.

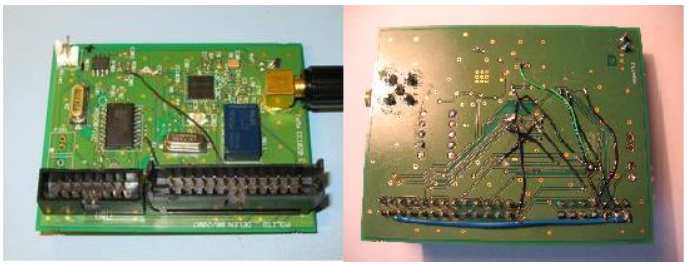

Figura 3.1. Prime modifiche della scheda

Tramite l'ausilio della seriale RS232 e del programma 'RealTerm', è stato possibile visualizzare e verificare costantemente sullo schermo del pc, la presenza di alcuni

messaggi inseriti nel codice di programmazione; con questo metodo, è stato possibile individuare, valutare e correggere alcuni errori, attraverso la modifica di alcune linee di codice, (in particolare quelle inerenti al settaggio delle porte di comunicazione del microcontrollore e quelle inerenti ai valori dei registri del CC1020).

Le modifiche eseguite, hanno permesso sia di stabilire una corretta comunicazione tra i due componenti, sia di resettare il transceiver; dunque, si è cominciato a sviluppare il codice di programmazione (che verrà trattato nel capitolo successivo), per il settaggio del CC1020 in modalità di trasmissione, generare una portante e trasmetterla.

In tali circistanze, si sono presentati i seguenti tre inconvenienti:

- Consumi eccessivi di corrente;
- Mancata commutazione del relay d'antenna (TQ2-3V) dalla modalità di ricezione a quella di trasmissione;
- Difficoltà nell'aggancio del PLL del CC1020;

Osservando lo schema elettrico e verificandone la corrispondenza con il PCB, ci si è resi conto che le connessioni delle linee 'Rx' e 'Tx' al relay d'antenna, erano invertite; pertanto la commutazione non poteva avvenire, poiché nella realtà il transceiver era posto permanentemente in modalità di trasmissione; ciò giustificava, peraltro, anche i consumi estremamente eccessivi di corrente.

Non potendo effettuare dei tagli sulle piste delle radiofrequenze, si è ovviato al problema temporaneamente, eliminando il relay e 'forzando' la scheda in modalità 'Tx', al fine di poter comunque proseguire con la programmazione e testare la catena di trasmissione.

La risoluzione del terzo problema è stata meno semplice, poiché il mancato aggancio del PLL, era dovuto ad un problema software; inoltre, il microcontrollore

MSP430F1121A si presentava poco affidabile e difficilmente gestibile, poiché mal supportato da IAR, che non consentiva di effettuare un 'debug' del codice.

In tali condizioni riuscire a trovare la causa è stato lungo e laborioso, quanto trovarne la risoluzione. Il problema era dovuto alla durata dei tempi di attesa (per l'accensione del sintetizzatore di frequenza e della calibrazione del PLL del CC1020), che dovevano essere settati manualmente e verificati continuamente tramite svariate misure effettuate con l'oscilloscopio.

A causa di tale inconveniente, si è deciso di procedere con la sostituzione del microcontrollore MSP430F1121A, con la versione MSP430F149, meno recente, ma più affidabile; attraverso questa modifica è stato possibile risolvere immediatamente il problema.

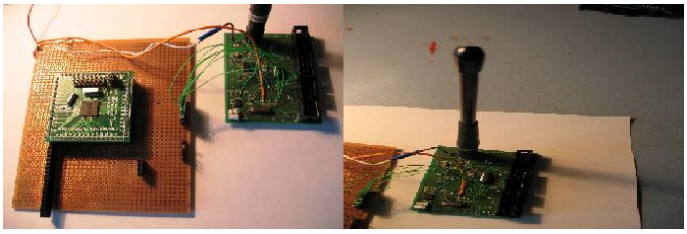

Figura 3.2. Scheda Tx/Rx dopo la sostituzione del microcontrollore

Il passo seguente è stato quello di generare una portante e trasmetterla. Per generare la portante è necessario settare correttamente i registri del transceiver, ciò è risultato essere abbastanza semplice, avvalendosi del programma 'SmartRFStudio' della Texas Instrument, che permette di ottenere i valori corretti da inserire nei registri, selezionando il tipo di modulazione voluto ed altri parametri.

Una volta risolti tutti i problemi trattati, settati correttamente tutti i registri e programmata la scheda, si è passati a verificare il funzionamento della scheda, tramite l'analizzatore di spettro. Durante questo test funzionale, si è riscontrato che la potenza del segnale in uscita era eccessivamente bassa (-45 dBm circa).

Dopo aver effettuato diverse verifiche nel software, è stato eseguito un controllo accurato e dettagliato sulla linea di trasmissione della scheda, al fine di valutare l'andamento della potenza dall'uscita dal CC1020 fino all'antenna e comprenderne le cause del malfunzionamento.

Il problema è stato trovato nel partitore R7-L14 evidenziato in figura:

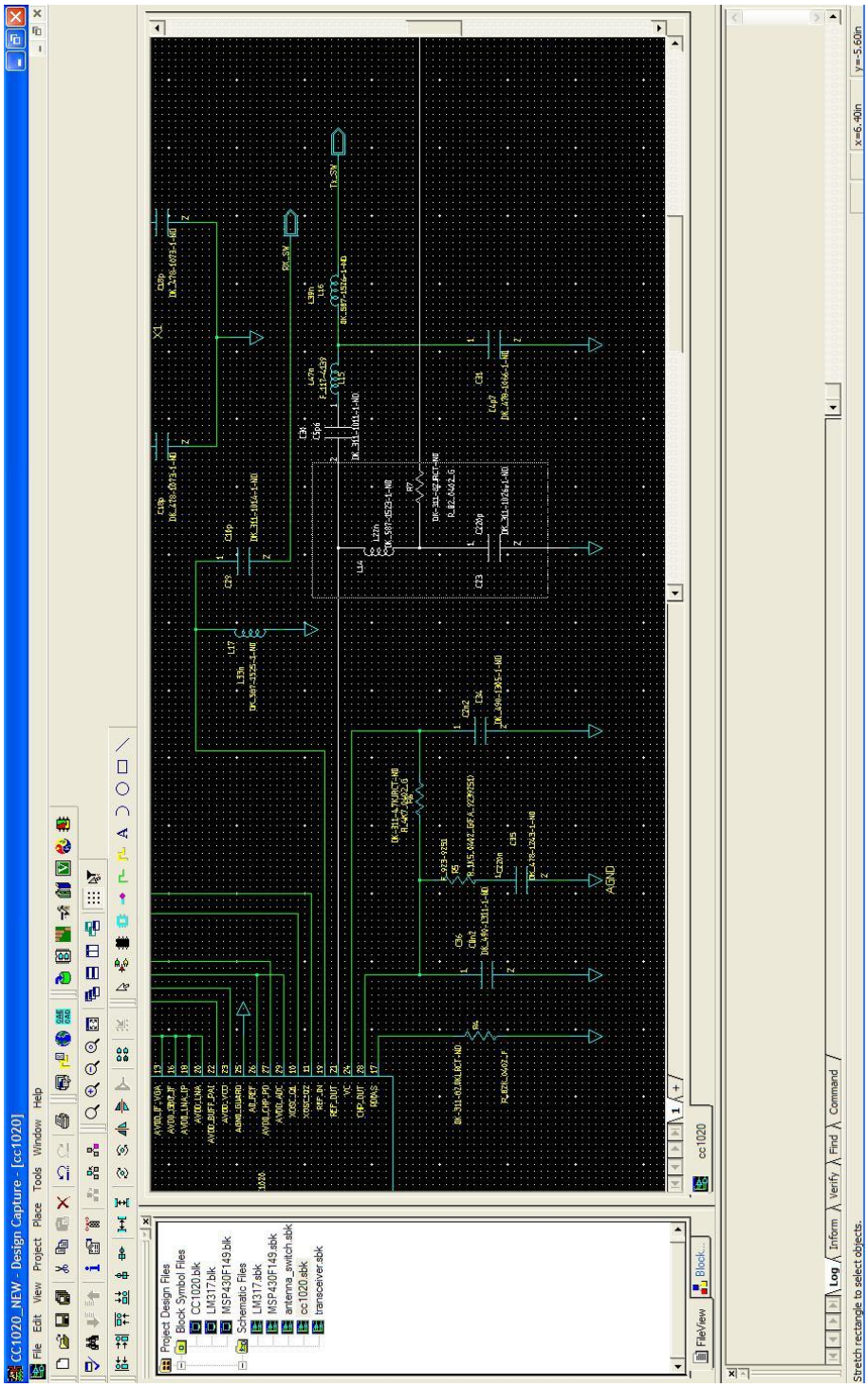

Figura 3.3. Partitore R7-L14

Un errato valore della resistenza R7, dieci volte più grande di quello previsto, provocava un'inversione della polarizzazione, che annullava il segnale. Attraverso la sostituzione della resistenza R7, con una di valore adeguato (82Ω), la potenza del segnale in uscita, è risultata essere quella programmata (10dBm).

Nello schema circuitale, inoltre, sono stati inseriti i seguenti componenti:

- Un diodo schottky nel blocco 'Antenna Switch' (Appendice A, Figura A.4), per evitare che durante l'accensione o lo spegnimento del sistema, gli over/undershoot di tensione danneggino l'interruttore;

- Un nMos per l'azionamento del relay d'antenna (anzichè un pMos), pilotato direttamente dal microcontrollore, al fine di evitare commutazioni spurie durante l'accensione del sistema; inoltre, comandando direttamente il relay d'antenna con il microcontrollore, è possibile introdurre un ritardo nella commutazione dello stesso, compensandone in questo modo il ritardo di commutazione.

Capitolo 4

Protocolli di comunicazione

Nel seguente capitolo, si farà riferimento alla suddivisione standard dei protocolli di trasmissione adoperati per le comunicazioni satellitari amatoriali.

4.1 Il protocollo AX-25

4.1.1 Introduzione

Questo protocollo è conforme alle raccomandazioni ISO 3309, 4335 (includendo DAD 1&2) e 6256 high-level data link control (HDLC) ed è stato creato per poter lavorare egualmente bene in ambienti radioamatoriali sia half che full duplex.

La maggior parte dei protocolli di link-layer a livello due, danno per scontato che la comunicazione avvenga tra un'apparecchiatura principale, detta DCE (Data Circuit-terminating Equipment), collegata a diverse altre più piccole, dette DTE (Data Terminating Equipment). Una tale assegnazione di funzioni, non è auspicabile in un mezzo condiviso qual'è il tipico canale radioamatoriale.

AX.25 considera una nuova classe di apparecchiature (devices) i DXE, capaci di svolgere sia i compiti di un DCE che di un DTE in egual misura.

4.1.2 Definizione dei campi

Le trasmissioni a livello due in Packet Radio (d'ora in poi PR), sono costituite da piccoli "blocchi" detti (frames). Ogni frame è l'insieme di parti più piccole unite in sequenze logiche univoche, detti fields (campi). Ogni pacchetto è formato da più campi. Ogni campo, field, è composto da gruppi di otto bit, octets o bytes, e ha specifiche funzioni.

I possibili pacchetti sono illustrati qui di seguito:

Flag	Address	Control	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	N*8 Bits	16 Bits	01111110

Figura 4.1. *Pacchetto U o S*

Flag	Address	Control	PID	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	8 Bits	N*8 Bits	16 Bits	01111110

Figura 4.2. *Pacchetto I*

Si descriveranno adesso le funzioni di ciascun campo:

Flag

Il P.R. è un protocollo orientato al bit; per evitare overrun e perdite di dati, l'unico modo che si ha per avvertire il corrispondente del termine di un pacchetto e della partenza di un altro,è l'invio di un campo di delimitazione per l'inizio e la fine di ogni frame. Questo campo viene chiamato FLAG field e consiste di un bit a "0" seguito da sei bit a "1" e di nuovo da uno "0", ovvero dal byte: 01111110 (7E esadecimale).

Le uniche volte in cui questa sequenza è valida, sono l'inizio e la fine di ogni pacchetto. Per evitare il ripetersi di questa sequenza all'interno di un frame, come si vedrà di seguito, vi sono dei meccanismi che eseguono il bit stuffing.

Esiste la possibilità di utilizzare questo segnale come idle (mantenimento della portante modulata dal segnale); infatti, il corrispondente ogni volta che termina di decodificare un byte così composto, attende il prossimo come inizio valido del pacchetto a seguire. Se il prossimo byte sarà di nuovo un flag viene mantenuto lo stato iniziale di attesa della parte utile del pacchetto e questo ciclo può esser ripetuto.

Address Field

Il campo d'indirizzamento, Address, viene usato per conoscere origine e destinazione di ogni singolo frame.

Nella raccomandazione CCITT X.25, questo campo è lungo al massimo un byte. Quindi permette al massimo 256 utilizzatori per ogni canale PR livello 2. Poiché alcuni bit di questo campo hanno altri usi, il numero di utilizzatori contemporanei potrebbe scendere al livello di 30 per ogni canale. Entrambe le raccomandazioni HDLC e ADCCP permettono l'estensione del campo d'indirizzamento. Il protocollo amatoriale AX.25 prevede quindi, per comune decisione, l'estensione dell'ADDRESS field per permettere l'inserzione dei nostri nominativi, sia del destinatario (destination) che del mittente (source)in ogni pacchetto.

Control Field

Il campo di controllo viene usato per identificare il tipo di frame e altri attributi della connessione a livello 2 ed è lungo un byte. Permette il mantenimento del controllo degli eventi di link (collegamento-connessione) tra le stazioni. La costruzione del campo di controllo di AX.25 è ripresa dalla raccomandazione CCITT X.25 per

operazioni bilanciate (LAPB), con in più l'aggiunta di un sottocampo di controllo ulteriore ripreso da ADCCP per permettere trasmissioni circolari e a stazioni non connesse.

Campo PID

Protocol Identifier field: campo identificatore di protocollo. È usato solo nei pacchetti d'informazione ed identifica che tipo di livello 3 è in uso, ammesso che ve ne sia uno. La sua decodifica ha i valori definiti in fig.4.3:

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	

Figura 4.3. *Definizioni di PID*

Dove: y, indica un valore qualunque del bit.

Information Field

Questo è l'unico campo del pacchetto che contiene le informazioni utili.

Il resto dei campi introduce overhead, in quanto non trasporta informazioni degli utilizzatori ma "solo" dati di funzionamento del protocollo. Il campo d'informazione è valido solo per tre tipi di pacchetti: I frames, UI, frames e FRMR frames; ovvero: Information frames; Unnumbered Information frames; FRaMe Rejected frames.

Il campo I può essere lungo sino a 256 bytes, comunque la sua lunghezza deve essere un multiplo pari di bytes, al minimo due bytes. Tutti i dati che vengono inseriti all'interno dell'I field vengono passati in modo "trasparente" dal mittente al destinatario lungo il link, salvo per l'inserzione di un eventuale bit a "0" necessaria per evitare che un FLAG compaia all'interno del campo dati (bit stuffing).

Frame Check Sequence

Il campo di controllo del pacchetto (FCS o CRC) è un numero su 16 bit e viene calcolato sia dal destinatario che dal mittente di ogni pacchetto. Il mittente lo calcola e lo inserisce nel pacchetto da spedire, il destinatario lo ricalcola (in base ai dati ricevuti) e lo controlla con quello speditogli nel frame. 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).

Le proprietà del CRC - 16 sono riassunte nella tabella 4.1.
Si ricorda che è necessario trasmettere prima il byte meno significativo del FCS (in quanto contiene i coefficienti del termine più grande).

L'FCS viene calcolato su tutti i bits di Address, Control, Protocol, Information and Padding fields, tranne che sui flag di inizio e fine frame e sullo stesso FCS.

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

Tabella 4.1. *Controllo di flusso del protocollo*

4.1.3 Procedure di trasmissione

Si illustrano in questa sezione le principali regole richieste dal protocollo per trasmettere i frame correttamente.

Bit Stuffing

Dopo il calcolo del FCS, per assicurarsi che la sequenza di bits del campo di FLAG non compaia all'interno degli altri campi, prima che ogni singolo frame venga trasmesso, ne viene eseguita una scansione, con una finestra mobile di cinque bit lungo il pacchetto.

Se vengono trovati 5 bit consecutivi a "1" la sequenza viene interrotta e viene inserito uno "0" per evitare appunto il ripetersi del carattere FLAG.

Bit Order of Transmission

Il frame viene trasmesso da sinistra vero destra. Si ricorda che con l'eccezione dell'FCS (Frame Check Sequence), tutti gli altri campi di ogni pacchetto AX.25 vengono trasmessi partendo dal Least Significant Bit (LSB), bit meno significativo.

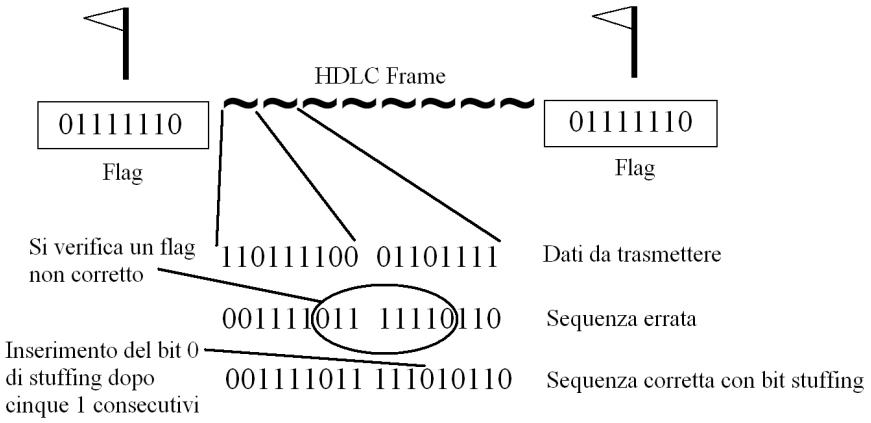

Figura 4.4. *Bit stuffing*

Frame Abort

Se un pacchetto deve essere terminato prematuramente, devono essere trasmessi almeno quindici bit consecutivi a "1", senza bit stuffing.

Invalid Frame

Ogni pacchetto più corto di 136 bits, o non incluso nei campi di FLAG (un byte con valore binari "011111"), o non composta da byte allineati (bytes incompleti, numero di bits non divisibile per 8 senza resto) o con una sequenza di sei o più "1" consecutivi, verrà considerato non valido.

4.1.4 Implementazione software

Nel campo delle trasmissioni digitali via radio i radioamatori fanno uso del TNC (Terminal Node Controller) che per l'appunto costituisce l'interfaccia tra PC e radioricetrasmittente (RTX), ma effettua in più la codifica e la decodifica dei messaggi in pacchetti.

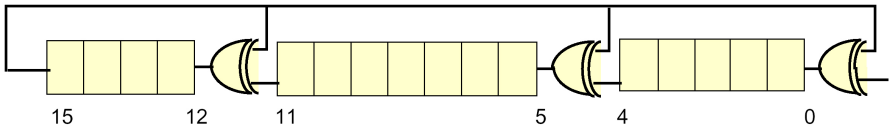

Figura 4.5. *CRC-hardware*

Il TNC al suo interno contiene poi anche un modem, che trasforma i segnali logici in segnali audio.

Ovviamente il PC per comunicare con il TNC (attraverso porta seriale tipo RS232) dovrà essere provvisto di un programma emulatore di terminale.

Il TNC è quindi sostanzialmente un modem intelligente; al suo interno si trova una CPU, una EPROM che contiene il programma di lavoro e delle memorie RAM. La CPU è in genere un processore tipo Z80. Il programma di gestione del TNC si occupa solamente della gestione degli errori di trasmissione; il PC si occupa di tutto il resto.

Nel nostro caso invece si è fatto uso di un microcontrollore quale l'MSP430 per gestire le trasmissioni che avverranno tramite il Chipcon CC1020.

Nel paragrafo successivo, verrà descritta la funzione che effettua il calcolo del FCS.

Calcolo del FCS

In figura 4.5 si riporta una possibile implementazione hardware dell'algoritmo del calcolo del FCS.

Risulta semplice quindi dedurne da questo schema una realizzazione che lavora sul bit, ma è molto inefficiente per un calcolo software.

Via software invece di calcolare il FCS per ogni bit, una lookup table di 256 elementi può essere usata per effettuare l'equivalente di 8 operazioni alla volta.

Essendo il polinomio da implementare $X^{16}+X^{12}+X^5+X^0$ ovvero 0x8404 la lookup table che ne deriva è la seguente:

```
static 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
    };
```

(questo è descritto in (Byte-wise CRC Calculations in IEEE Micro, June 1983, pp. 40-50).

A questo punto basta seguire questo semplice algoritmo per calcolare il FCS:

1. Inizializzare il FCS a FFFF
2. EX-OR del nuovo byte in ingresso con il byte meno significativo del FCS per ottenere l'indice della lookup table
3. Shiftare il FCS a destra di 8 bit
4. EX-OR del FCS con il byte ottenuto in precedenza dalla lookup table
5. Ripetere i passi dall 1 al 3 per tutti i byte

Che è rappresentato dalla seguente implementazione

$$FCS = Hi(FCSvalue)XORTable[NewByteXORLo(FCSvalue)] \qquad (4.1)$$

La funzione che esegue tale algoritmo è:

```
unsigned short pppfcs(unsigned short fcs, unsigned char *cp,
int len)
      { while (len--)
        fcs = (fcs >> 8) ^ fcstab[(fcs ^ *cp++) & 0xff];
return (fcs); }
```

Capitolo 5

Programmazione della scheda

Nel seguente capitolo saranno illustrati i criteri e le modalità di sviluppo del software di programmazione della scheda, per la trasmissione e per la ricezione, quali obiettivi fondamentali della tesi in esame; inoltre ci si soffermerà sulla descrizione delle funzioni generate e su alcune caratteristiche delle stesse, alle quali è necessario porre particolare attenzione.

Il software è stato sviluppato con la finalità, sia di rendere comprensibili in maniera univoca le operazioni che devono essere effettuate sul microcontrollore e sul transceiver, sia di poter essere riutilizzato in progetti futuri.

Nei paragrafi successivi, si discuterà brevemente dell'interfaccia CC1020-MSP430F149 e del principio di funzionamento del software, come punto di partenza per tutto il lavoro a seguire.

5.1 Interfaccia CC1020-Msp430F149

Prima di potere implementare le diverse funzioni software, è stato necessario capire la modalità con cui avviene la comunicazione fra il microcontrollore e il transceiver.

La comunicazione fra MSP40F149/CC1020, avviene attraverso la gestione di quattro segnali (PCLK, PDI, PDO, PSEL), collegati alla porta numero due del microcontrollore e gestiti dai primi quattro bit (pin 20, 21, 22, 23); dunque, in primis, sono stati definiti nel software i collegamenti Pin/segnali.

I dati, vengono scambiati attraverso l'utilizzo del pin DIO, (bidirezionale) e attraverso il pin DCLK di output, che provvede al sincronismo con i dati.

Il microcontrollore configura e controlla il CC1020 attraverso un bus SPI ad alta velocità e a sua volta è programmato attraverso un connettore JTAG.

Il CC1020 è provvisto di registri a 8 bit, ognuno dei quali è individuato da un indirizzo a 7 bit; inoltre, è previsto un bit di Read/Write per iniziare le operazioni.

Una completa configurazione del CC1020, richiede l'invio di 33 data frame di 16 bit ciascuno (7 bit d'indirizzo, uno di R/W e 8 bit di dati). Il tempo necessario per effettuare la configurazione completa, dipende dalla frequenza del segnale PCLK.

I cicli di lettura/scrittura, avvengono nelle seguenti modalità:

- Ciclo di scrittura: i 16 bit sono inviati su PDI; i primi 7 bit significativi di ogni data frame sono quelli d'indirizzo, mentre l'ottavo bit è il bit di R/W (alto per la fase di scrittura e basso per quella di lettura) e i successivi 8 bit sono quelli dei dati. In tale fase, il segnale di Program SELect (PSEL), va tenuto basso.

- Ciclo di lettura: Vengono inviati i primi 7 bit d'indirizzo, seguiti dall'ottavo bit di R/W (basso per la fase di lettura). Ricevuto il comando di lettura, il CC1020 restituisce i dati dal registro, attraverso il segnale PDO che dev'essere settato come input dal microcontrollore.

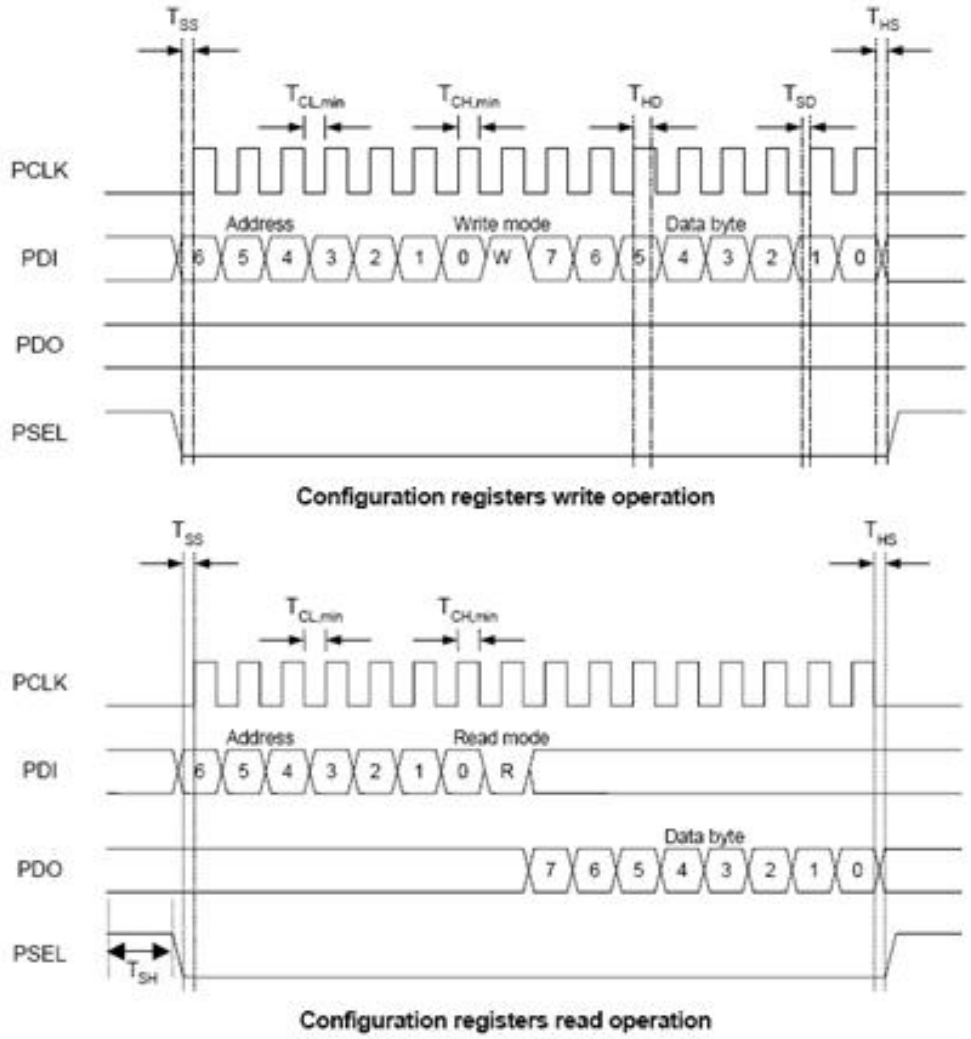

Figura 5.1. Segnali che gestiscono la comunicazione tra CC1020 e MSP430F149 in lettura e scrittura

Il principio di funzionamento della scheda, nonché del software di programmazione, è quello descritto in figura:

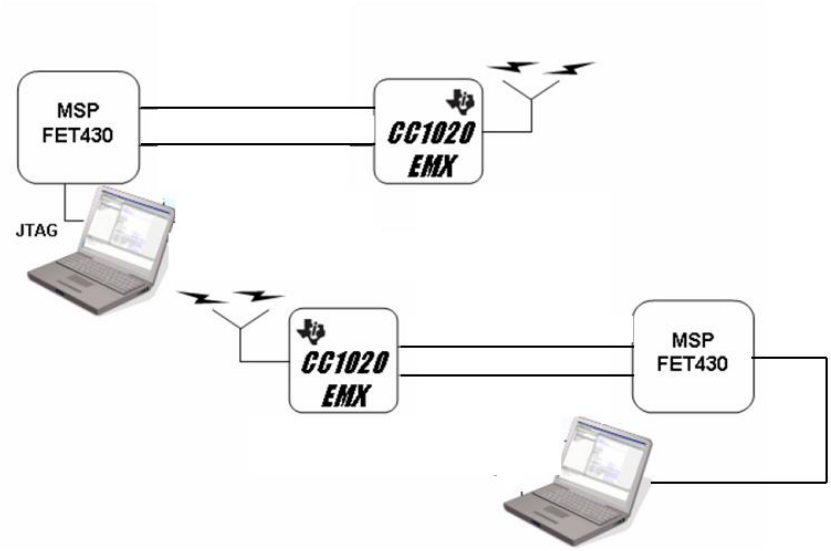

Figura 5.2. Schema della comunicazione Ground Station/scheda in trasmissione e ricezione

Nella modalità di ricezione, il processore di bordo accenderà il sistema e fornirà il comando di ricezione; il microcontrollore provvederà alla configurazione del transceiver per la modalità prevista e tutti i dati che dovranno essere inviati dalla Ground Station saranno ricevuti. Una volta ricevuto il pacchetto, il transceiver sarà portato nuovamente nella condizione iniziale (Power Down).

Analogamente, in fase di trasmissione, il processore di bordo accenderà il sistema e fornirà il comando di trasmissione; il microcontrollore provvederà alla configurazione del transceiver per la modalità trasmissione, lo abiliterà alla trasmissione dati e, una volta inviato il pacchetto, lo riporterà nella condizione iniziale (Power Down). Le funzioni che devono essere implementate durante queste due fasi, sono schematizzate in figura 5.3:

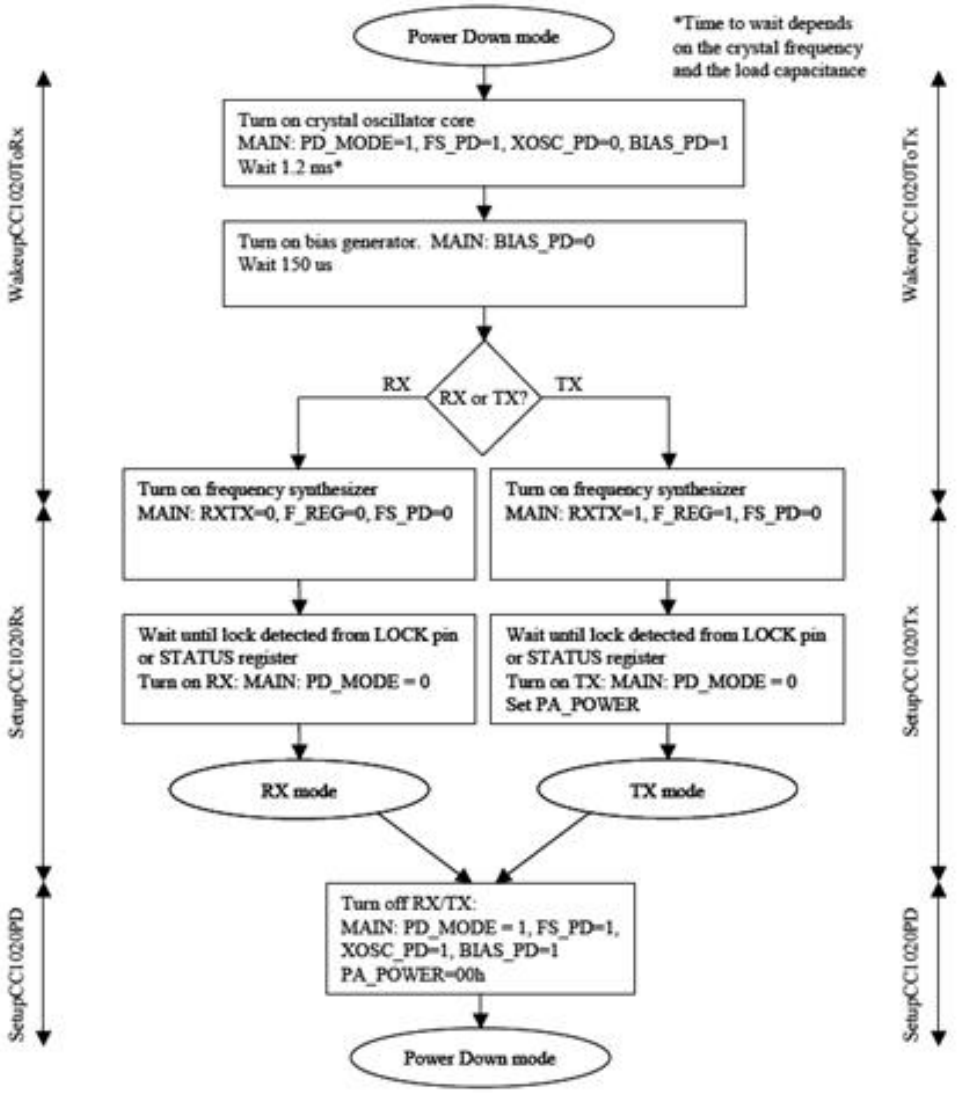

Figura 5.3. Schema per lo sviluppo del software

5.2 Struttura del Software

Il software sviluppato per il microcontrollore MSP430, è scritto in C per il compilatore IAR per MSP430 e permette di inviare caratteri ASCII da un pc all'altro. L'operazione di altissima priorità del software, viene effettuata tramite il controllore degli interrupt esterni, che è innescato dalle transizioni del DCLK che proviene dal CC1020.

Il software consiste di diversi file di codice sorgente, nella seguente tabella, è fornita una breve descrizione delle sorgenti allegate:

File	Descrizione
Main.C	sorgente del programma principale
CC1020.c	Contiene funzioni di configurazione del CC1020
Uart.c	Inizializza l'UART e definisce le funzioni di tx caratteri su pc
SPI.c	Inizializza l'SPI, definisce funzioni di tx caratteri e raggruppamento bit per l'invio degli stessi in bytes
APRS.c	Definisce prologo, funzioni di scrambling-descrambling, calcolo FCS, invio e ricezione pacchetti

Tabella 5.1. File sorgente del software creto

Il 'main', effettua le chiamate delle funzioni, secondo il flusso mostrato in figura:

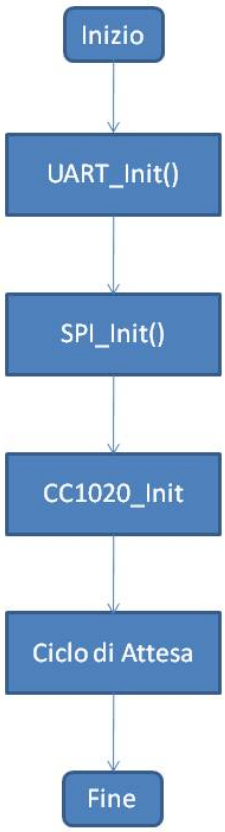

Figura 5.4. Flusso delle funzioni chiamate nel main

I primi tre blocchi rappresentano le funzioni di inizializzazione della Uart, della SPI e del CC1020, mentre l'ultimo rappresenta il ciclo di attesa del comando, il cui diagramma di flusso è mostrato in figura:

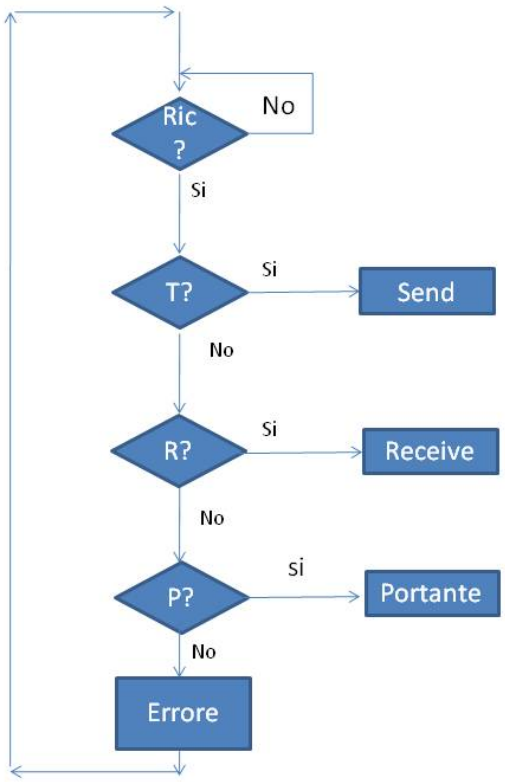

Figura 5.5. Flow Chart del ciclo di attesa dei comandi

Durante questo ciclo, il sistema rimane nell'attesa di ricevere un comando.

Una volta ricevuto il comando, ne verifica la correttezza e se non dovesse corrispondere ad uno di quelli predefiniti, lo segnala attraverso un messaggio d'errore; in caso contrario, si verificano tre possibilità:

1. Comando 'T': il sistema entra in modalità di trasmissione;
2. Comando 'R': il sistema entra in modalità di ricezione;
3. Comando 'P': il sistema genera una portante;

Nelle seguenti figure, sono mostrati i flussi dei blocchi send, receive e portante:

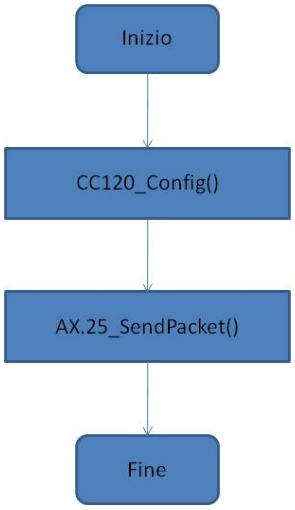

Figura 5.6. Diagramma di flusso del blocco Send

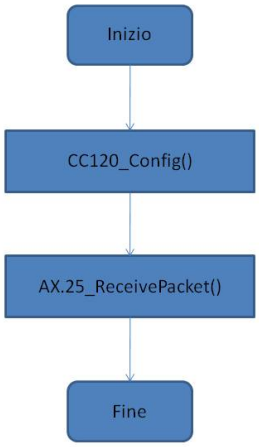

Figura 5.7. Diagramma di flusso del blocco receive

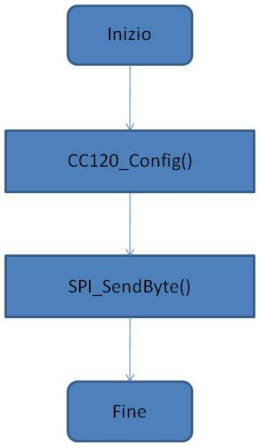

Figura 5.8. Diagramma di flusso del blocco portante

5.3 Descrizione delle funzioni create

In questo paragrafo saranno descritte le funzioni create, presenti nelle sorgenti (allegate nell'appendice B), Main.c, CC1020.c, SPI.c, Uart.c, APRS.c.

5.3.1 CC1020.c

In questa sorgente sono presenti le seguenti funzioni:

- `CC1020_Init()`

  Configura le porte del CC1020, inerenti ai quattro segnali PSEL, PDI, PDO e PDCLK, per poter procedere alla sua configurazione.

- `char CC1020_Reset(void)`

  Configura i registri del CC1020, con i valori iniziali predefiniti.

- `CC1020_WakeUpToTX(char TXANALOG)`

  Imposta i bit del registro 'ANALOG' del CC1020; in paricolare setta ad 1 i bit

  `'LO_DC, PA_BOOST, DIV_BUFF_CURRENT_3'`

  e setta a '0' tutti gli altri. E' una delle funzioni più importanti, poichè provvede a 'svegliare' il CC1020, accendendo rispettivamente quarzo, generatore di bias e sintetizzatore di frequenza. Nella funzione sono presenti due cicli di attesa, il primo rappresenta il tempo necessario di cui ha bisogno il quarzo per stabilizzarsi, teoricamente un attesa di 2-5 ms, prima di poter accendere il generatore di bias; nella realtà tale tempo di attesa è stato portato ad 8 ms, poichè il quarzo da noi adoperato non riusciva a stabilizzarsi in quel range temporale. Trascurare questa attesa può comportare un mancato aggancio

del PLL. Il secondo ciclo di attesa, 150ms, rappresenta il tempo che deve obbligatoriamente intercorrere tra l'accensione del generatore di bias e quella del sintetizzatore; anche in questo caso, riducendo il ciclo di attesa, il PLL potrebbe non agganciare.

- `char CC1020_Config(void)`

  Funzione che provvede alla configurazione dei registri del CC1020.

  Per la configurazione dei registri, ci si è avvalso del programma'SmartRfStudio' della Texas Instrument; a seconda delle frequenze di utilizzo, della potenza voluta in uscita, del baud rate richiesto, della modulazione e altri parametri, è possibile ottenere i valori dei registri che saranno poi inseriti nella funzione in oggetto.

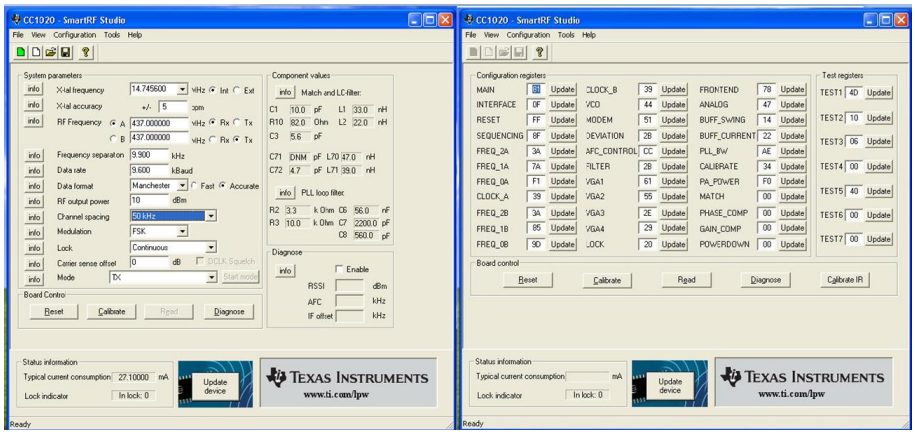

Figura 5.9. Settaggio dei registri tramite il programma SmartrfStudio

- `char CC1020_Calibrate(char pa_power)`

  Provvede all'aggancio del PLL ed eventualmente, se al primo tentativo non dovesse agganciarsi, riprova. In questa fase il PA interno al transceiver, viene spento.

- `CC1020_SetupTX(char TXANALOG, char PA_POWER)`

  E' la funzione che provvede a gestire l'aggancio del PLL e al settaggio della potenza in uscita,i parametri impostati sono: il contenuto del registro 'Analog' e il contenuto del registro 'PA Power'. Il sistema, dopo la chiamata di questa funzione, è pronto a trasmettere.

- `void CC1020_WakeUpToRX(char RXANALOG)`

  Imposta i bit del registro 'ANALOG' del CC1020; in particolare, setta ad 1 i bit 'LO_DC, PA_BOOST, DIV_BUFF_CURRENT_3'e setta a '0' tutti gli altri. E' una delle funzioni più importanti, poichè provvede a 'svegliare' il CC1020, accendendo rispettivamente quarzo, generatore di bias e sintetizzatore di frequenza. Nella funzione sono presenti due cicli di attesa, il primo rappresenta il tempo necessario di cui ha bisogno il quarzo per stabilizzarsi, teoricamente un attesa di 2-5 ms, prima di poter accendere il generatore di bias; nella realtà tale tempo di attesa è stato portato ad 8 ms, poichè il quarzo da noi adoperato non riusciva a stabilizzarsi in quel range temporale. Trascurare questa attesa può comportare un mancato aggancio del PLL. Il secondo ciclo di attesa, 150ms, rappresenta il tempo che deve obbligatoriamente intercorrere tra l'accensione del generatore di bias e quella del sintetizzatore; anche in questo caso, riducendo il ciclo di attesa, il PLL potrebbe non agganciare.

- `char CC1020_SetupRX(char RXANALOG, char PA_POWER)`

  E' la funzione che provvede a gestire l'aggancio del PLL e al settaggio della potenza del PA interno al CC1020. I parametri impostati sono: il contenuto del registro 'Analog' e il contenuto del registro 'PA Power'. Il sistema, dopo la chiamata di questa funzione, è pronto a ricevere.

- `void CC1020_SetupPD()`

Una volta trasmesso il pacchetto dati, sintetizzatore di frequenza, generatori di bias e quarzo vengono spenti. Tale funzione rappresenta la modalità di 'stand-by', nella quale viene posto il CC1020 al termine della trasmissione/ricezione dei pacchetti di dati.

5.3.2 Uart.c

- `void printUART (unsigned char *message)`

  Funzione il cui scopo è quello di stampare a video i vari messaggi tramite seriale RS232 e il programma 'RealTerm'. Il parametro 'message', è un puntatore alla stringa da inviare.

- `void UART_Init()`

  Funzione che abilita l'USART del microcontrollore MSP430F149 e configura clock e Baud Rate. Nel caso specifico è stata adoperata ed abilitata l'USART0 con clock pari a 2MHz e un BR pari a 9.6 kbps.

- `void UART_SendByte (unsigned char data)`

  Funzione che consente di trasmettere un byte alla Uart. Data è il byte che dev'essere inviato.

- `unsigned char UART_ReceiveByte ()`

  Funzione che consente di ricevere un byte dalla Uart.Data è il buffer su cui vengono scritti i byte.

- `unsigned char UART_ReceivePacket(unsigned char *pDato)`

  Funzione che consente di ricevere un pacchetto dalla Uart. Il parametro 'pDato' è un puntatore al buffer in cui il pacchetto viene scritto.

- `unsigned char UART_SendPacket(unsigned char *pDato)`

  Funzione che consente di trasmettere un pacchetto alla Uart. Il parametro 'pDato' è un puntatore al buffer in cui è contenuto il pacchetto da trasmettere.

5.3.3 SPI.c

- `void SPI_Init()`

  Abilita la SPI del MSP430F149 e ne imposta il tempo di clock.

- `void SPI_SendByte (unsigned char data)`

  Funzione che consente di trasmettere un byte alla SPI. Il parametro Data rappresenta il Byte da inviare.

- `void SPI_ResetBit ()`

  Inizializza il contatore, per il successivo conteggio dei bit.

- `void SPI_SendBit (unsigned char data)`

  Il suo compito è quello di trasmettere i singoli bit, ma solo dopo averli raggruppati in gruppi di otto. Il parametro Data, può essere 1 o 0.

5.3.4 AX.25.c

- `void pppfcs(char *cp, int len)`

  Funziona che implementa l'algoritmo per il calcolo del FCS, descritto nel capitolo precedente.

- `void AX25_ComputeFCS(char * packet, int packet_len)`

  Effettua il calcolo del FCS su un pacchetto. I parametri 'packet' e 'packet_len' rappresentano rispettivamente, il puntatore al pachetto da trasmettere e la sua lunghezza.

- `void AX25_SendByte(unsigned char byte, int flag)`

  Consente di trasmettere un byte ed effettua il controllo del massimo numero di bit consecutivi 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 messi a 1.

- `void AX25_SendPacket(unsigned char * packet, unsigned int packet_len)`

  Consente la trasmissione del pacchetto dati (AX.25). I parametri 'packet' e 'packet_len', rappresentano rispettivamente, il puntatore al pacchetto da trasmettere e la sua lunghezza.

5.3.5 Timer.c

In questa sorgente sono state create le funzioni, che si occupano della temporizzazione del sistema:

- `void TIMER_SetupTimer_ms(short volatile *semaforo)`

  E' un contatore in ms;

- `void TIMER_Wait_us(short volatile semaforo)`

  E' un ciclo di attesa in μs.

Capitolo 6

Realizzazione del nuovo circuito stampato

Di seguito saranno descritte le fasi di progettazione del circuito stampato, realizzato con l'ausilio del software Mentor Graphics, del quale se ne accennerà brevemente il funzionamento.

La realizzazione di qualsiasi circuito stampato, comporta il passaggio dai seguenti punti:

- Realizzazione di una libreria centrale, attraverso il software Library Manager;
- Progetto e verifica dello schematico, attraverso il software Design Capture;
- Realizzazione del PCB attraverso il software Expedition PCB;
- Realizzazione del PCB 'fisico', ovvero, il file gerber generato dal software Expedition PCB, viene inviato ad una ditta che si occuperà di produrre lo stampato.

Nella libreria sono definiti i simboli e le celle di ogni componente, che servono rispettivamente per la realizzazione degli schemi elettrici e del layout. Nel nostro

caso non è stato necessario crearne una, poiché si disponeva già di una libreria adatta allo scopo (AraMiS Mentor Lib); dunque ci si è solamente limitati ad aggiungere pochissimi componenti e a verificare che le celle rispettassero i requisiti richiesti dalla ditta che produrrà lo stampato.

6.1 Progetto dello schema elettrico

Lo schema elettrico della scheda è stato realizzato con l'ausilio del programma Design Capture, seguendo i criteri descritti di seguito.

In primis sono stati creati separatamente gli schemi elettrici dei singoli componenti della scheda, successivamente, si è proceduto con la realizzazione dei blocchi degli schemi elettrici progettati e con l'interconnessione degli stessi (allegati nell'appendice A).

L'aver adoperato dei blocchi interconnessi fra loro, oltre a consentire la realizzazione e la verifica della correttezza delle interconnessioni in maniera più speditiva, comporta l'enorme vantaggio di poter apportare modifiche più o meno significative alla scheda con maggiore semplicità; infatti, se si volesse sostituire un componente qualunque con uno più efficiente o di nuova generazione, occorrerebbe solamente creare un nuovo schema elettrico, il suo relativo blocco e sostituirlo ad uno di quelli già presenti.

6.2 Realizzazione PCB

Una volta completato lo schema elettrico, è necessario eseguire sullo stesso, tutti i controlli specificati dall'utente al fine di verificare la presenza di errori nel progetto, per far ciò ci si avvale del comando 'verify':

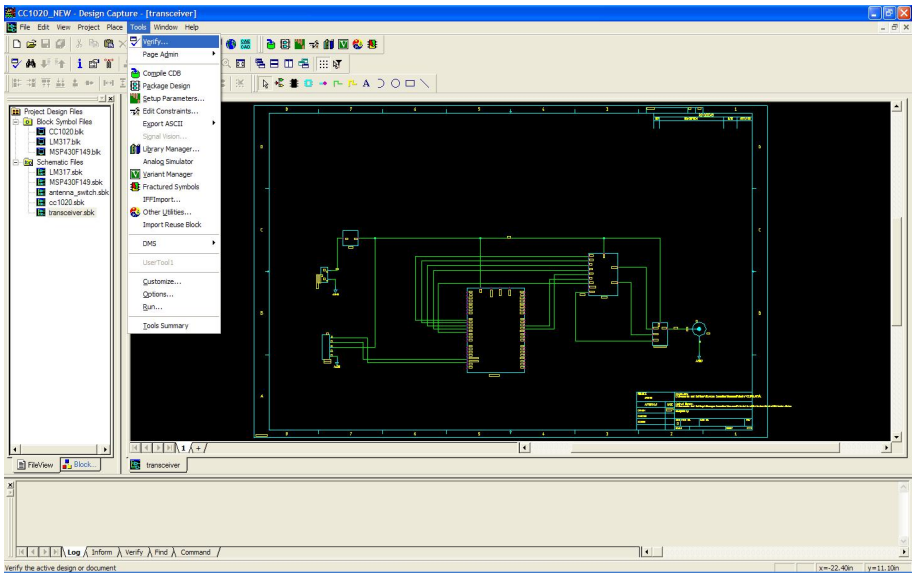

Figura 6.1. Comando che consente il riscontro di errori nel progetto dello schema elettrico

Successivamente a questa verifica, corretti eventuali errori, si passa alla compilazione del progetto mediante il comando 'Compile CDB':

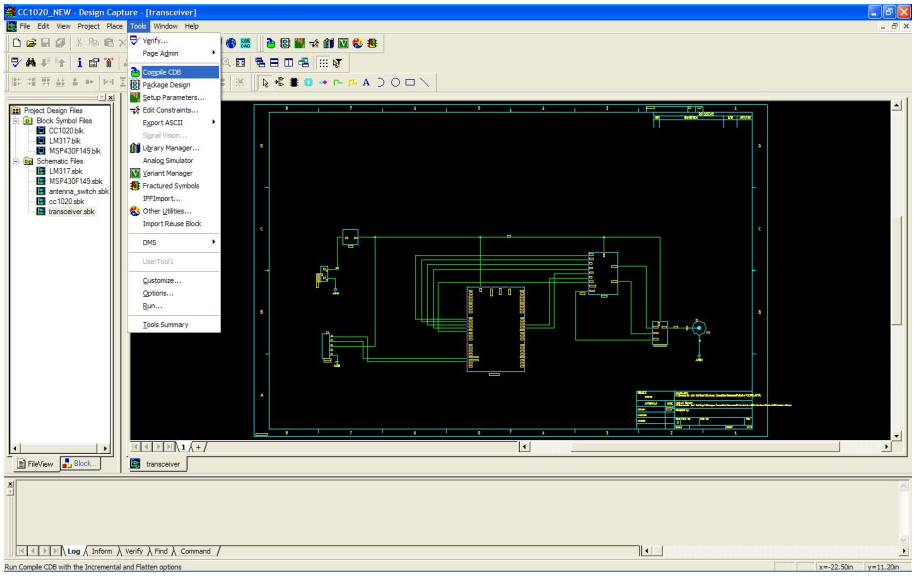

Figura 6.2. Comando Compile CDB

In tale fase vengono create delle netlists, contenenti i dati relativi alla connettività strutturale e gerarchia del design.

Terminata la compilazione e corretti eventuali errori, si procede con il 'packaging', il cui obiettivo consiste nel 'mapping' dei simboli logici nelle relative celle fisiche; ovvero tutte le informazioni relative ai simboli e alle reti del progetto dal file generato durante la compilazione, vengono localizzate sulle corrispondenti celle fisiche e vengono assegnate ai simboli riportati sullo schematico.

Dopo aver completato le suddette operazioni, si è passati alla creazione del PCB mediante il software 'Expedition PCB'.

Il circuito stampato che si andrà a realizzare, è stato pensato per essere realizzato su due strati, dei quali, il primo (top) è un piano di segnale, mentre il secondo (bottom) è un piano di massa. Nella realtà, non si è riusciti, per questioni di dimensioni, a porre tutte le linee di segnale sul piano TOP.

Dopo aver settato 'l'Editor Control', si è passati all'inserimento delle celle sullo stampato, mediante la finestra di dialogo 'Place Part'. Le piste possono essere posizionate automaticamente (comando 'Auto Route'), o manualmente. La modalità automatica, purtroppo, non ha permesso di tracciare tutte le linee, per cui si è dovuto procedere obbligatoriamente con la modalità manuale.

Infine, tramite il comando 'Design Rule Check', si è verificato che non fosse stata violata alcuna regola specificata all'inizio del progetto ed è stato generato il 'Gerber Output file', che è stato inviato alla ditta per il plotting del PCB fisico.

6.3 Collaudo

Le ultime due fasi del progetto, dovrebbero riguardare il montaggio dei componenti e il collaudo della scheda. In realtà, a causa dei tempi di realizzazione del circuito

stampato relativamente lunghi, in questo ultimo capitolo si descriverà il collaudo effettuato sulla scheda modificata, che comunque, risulta essere funzionante.

Il primo test ha riguardato la verifica, tramite l'analizzatore di spettro, del segnale trasmesso dal CC1020. In questo test, è di primaria importanza constatare che i parametri frequenza e potenza del segnale in uscita, siano effettivamente quelli previsti, ovvero 437MHz e 10dBm.

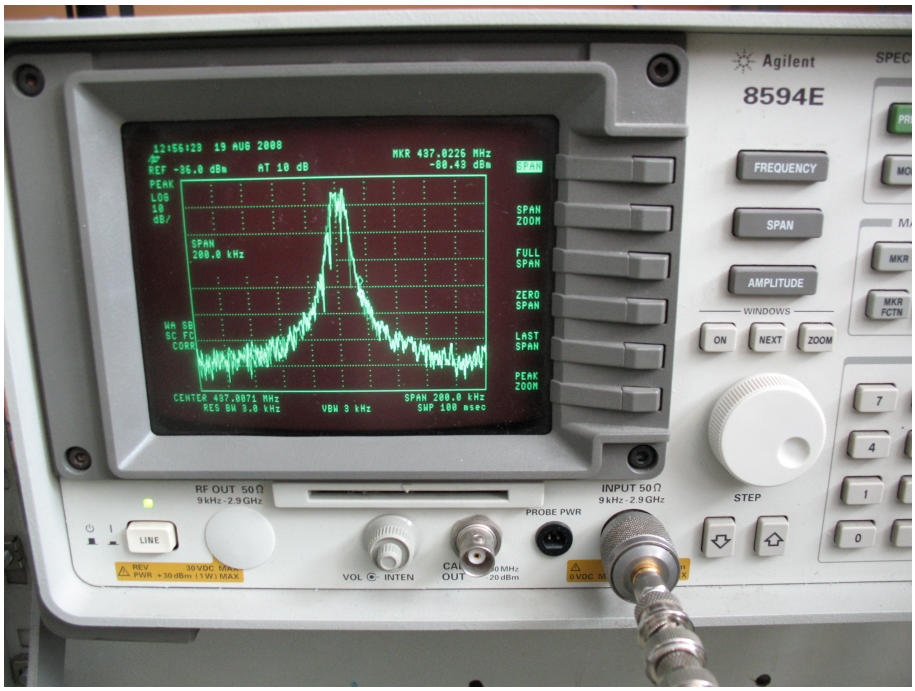

Figura 6.3. Segnale trasmesso dal CC1020, visto all'analizzatore di spettro

Una volta verificata la correttezza di questi due parametri, si è passati alla verifica della correttezza dei dati trasmessi.

Tale test è stato effettuato attraverso una catena di dispositivi, che consentono la trasmissione dei comandi e la ricezione delle telemetrie, simulando il canale di trasmissione.

In particolare, sono stati adoperati:

- Un pc (pc1), collegato tramite seriale RS232 alla scheda in esame;
- Un TNC (DSP 2232), prodotto dalla Timewave;
- Una radio FT-847, prodotta dalla Yaesu;
- Un pc (pc2), collegato al TNC, rappresentante la stazione di terra.

In trasmissione, i dati vengono inviati tramite cavo seriale RS-232 dal PC al TNC, al quale è affidato il compito di gestire il protocollo AX.25, calcolarne il FCS e modulare il segnale in FSK, con bitrate pari a 9.6kbps. L'uscita viene fornita alla radio Yaesu, che ha il compito di trasmetterla, con una potenza massima pari a 50W.

In ricezione il segnale viene inviato al LNA e quindi demodulato dalla radio ed inviato al TNC. Quest'ultimo converte il segnale in una sequenza di bit, quindi effettua un controllo sul FCS ed in caso di assenza di errori, invia il flusso dati al pc.

Dunque, dopo avere collegato Yaesu, TNC e pc2 (Ground station) e dopo aver connesso la scheda al pc1, è stato dato dal pc1 il comando 'Send',che comanda la trasmissione dati del sistema,inviando un certo numero di byte (il messaggio è definibile dall'utente):

In questa fase è stato verificato che il pacchetto dati trasmesso, corrispondeva esattamente a quello inviato, il che implica che la scheda è correttamente programmata e funzionante.

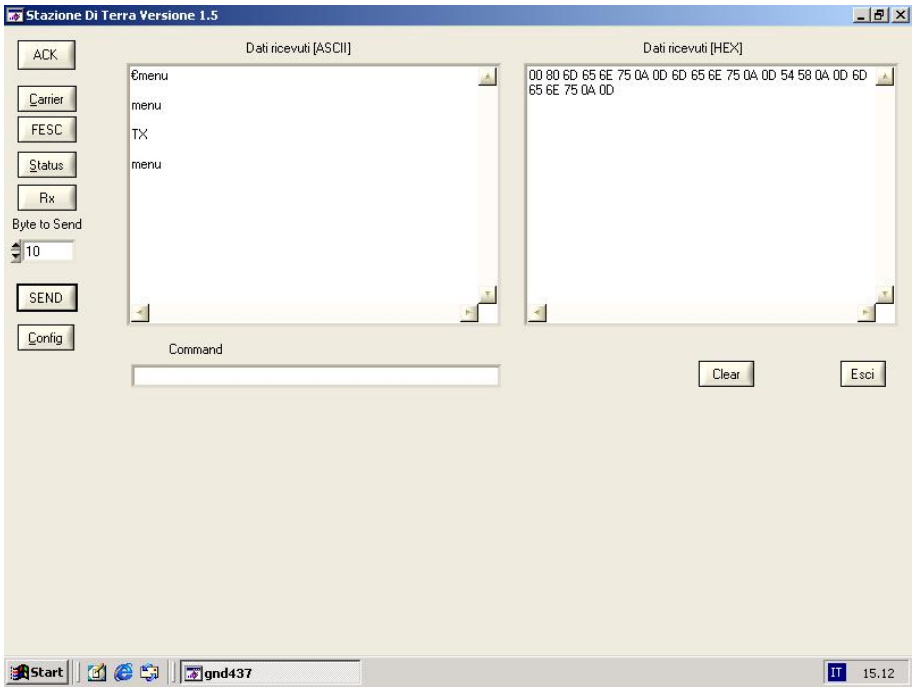

Figura 6.4. Comando Send

Figura 6.5. Yaesu e TNC

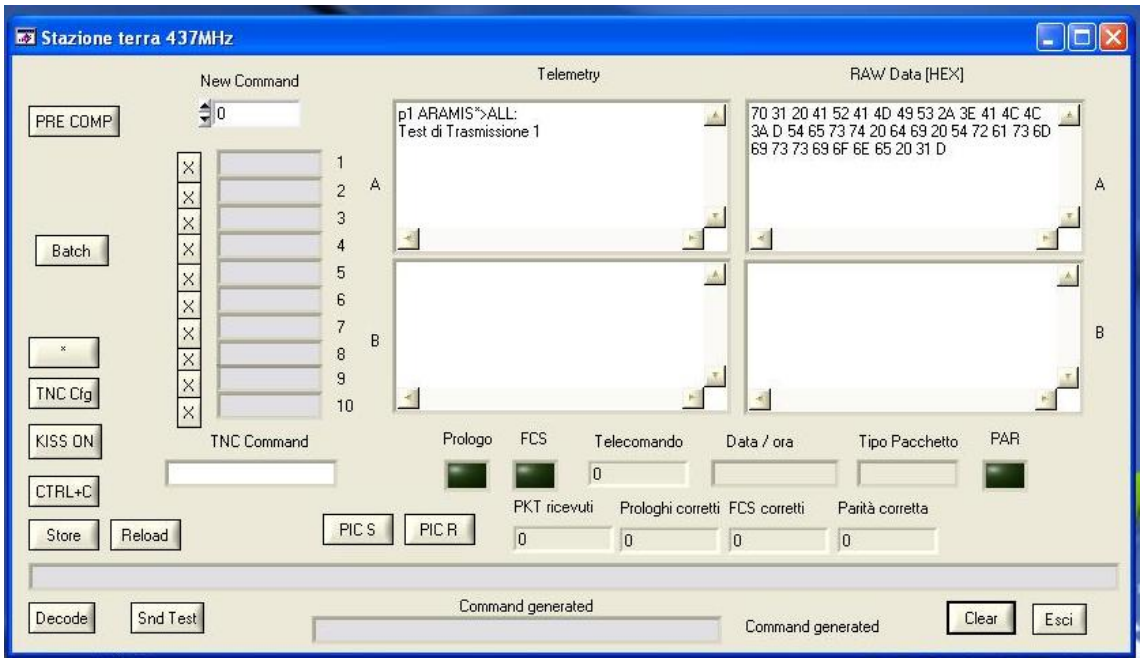

Figura 6.6. Segnale trasmesso dal CC1020, ricevuto dalla radio Yaesu, elaborato dal TNC e poi inviato alla Ground Station

Capitolo 7

Conclusioni

In questo lavoro sono stati sviluppati il progetto, l'analisi e la realizzazione, fino al livello del layout finale, della scheda di trasmissione e ricezione a 437MHz per applicazioni spaziali. Dopo una panoramica sul conteso nel quale la suddetta scheda si inquadra, è stato realizzato un modello funzionante della stessa, che dovrà essere utilizzato sul satellite AraMiS.

La partecipazione al progetto AraMiS in qualità di tesista, sotto la vigile supervisione di professori e dottorandi del dipartimento di elettronica, che mi hanno seguito costantemente durante tutto il lavoro svolto, oltre a consentirmi di mettere in atto tutte le conoscenze teoriche acquisite nel corso degli studi, di approfondire l'utilizzo della strumentazione di laboratorio, le conoscenze nel campo dell'elettronica delle telecomunicazioni e di acquisire nozioni di applicazioni aerospaziali, mi ha dato soprattutto la possibilità d'imbattermi in tutte le svariate problematiche, che si incontrano durante la progettazione di un sistema elettronico e nell'interfacciare diversi componenti hardware .

La scheda elettronica finale è stata senz'altro il frutto delle conoscenze specifiche nel campo delle comunicazioni satellitari, della conoscenza della teoria delle linee di trasmissioni e delle conoscenze base sull'elettronica delle telecomunicazioni, ma

sopratutto dell'esperienza acquisita da tutto il personale impegnato nel progetto, che ha voluto e saputo condividere con me; infatti, l'elevato livello di sperimentazione e la mancanza di esperienza pratica in tale campo non mi avrebbero permesso di realizzare il progetto, senza l'ausilio di una valida linea guida.

La complessità del progetto nonché delle problematiche riscontrate, hanno dettato la necessità di suddividere il progetto dell'intera scheda in più parti, che potessero essere testate singolarmente, al fine di poter capire le cause dei vari malfunzionamenti e risolverli.

Gli obiettivi posti in questa tesi, progetto hardware e software della scheda del sistema di comunicazione del satellite, sono stati raggiunti e il risultato finale è stato quello di ottenere una scheda funzionante testata e collaudata.

A causa delle tempistiche necessarie per la produzione del PCB realizzato, non è stato possibile collaudare e testare la nuova 'versione' del sistema; dunque, per chi riprenderà questo lavoro, gli obiettivi saranno:

- Montaggio della scheda e conseguente collaudo;
- Realizzazione di una versione della stessa, con PA di potenza superiore;
- Integrarla sul satellite AraMiS.

Appendice A

Schemi elettrici

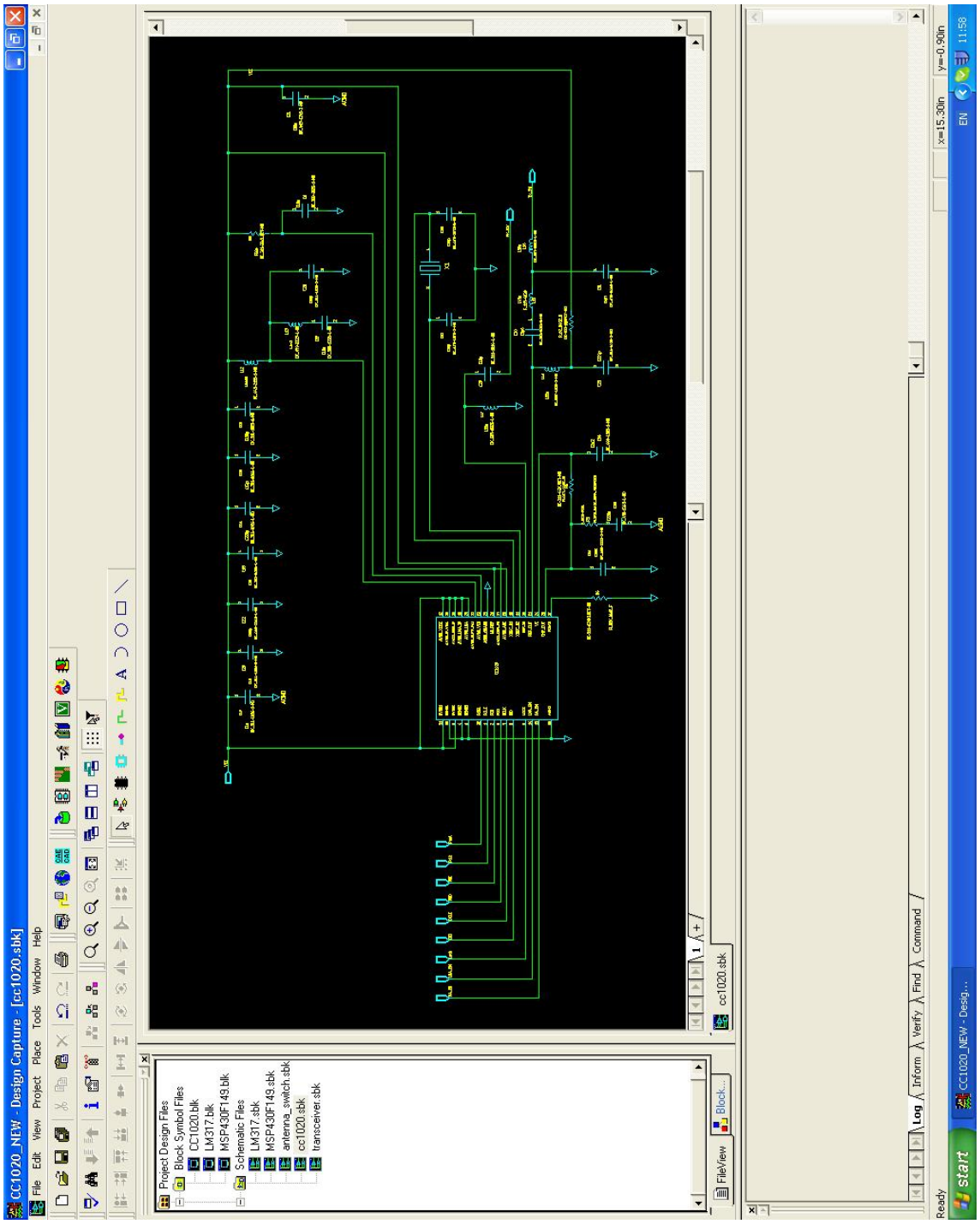

Figura A.1. Schema elettrico del CC1020

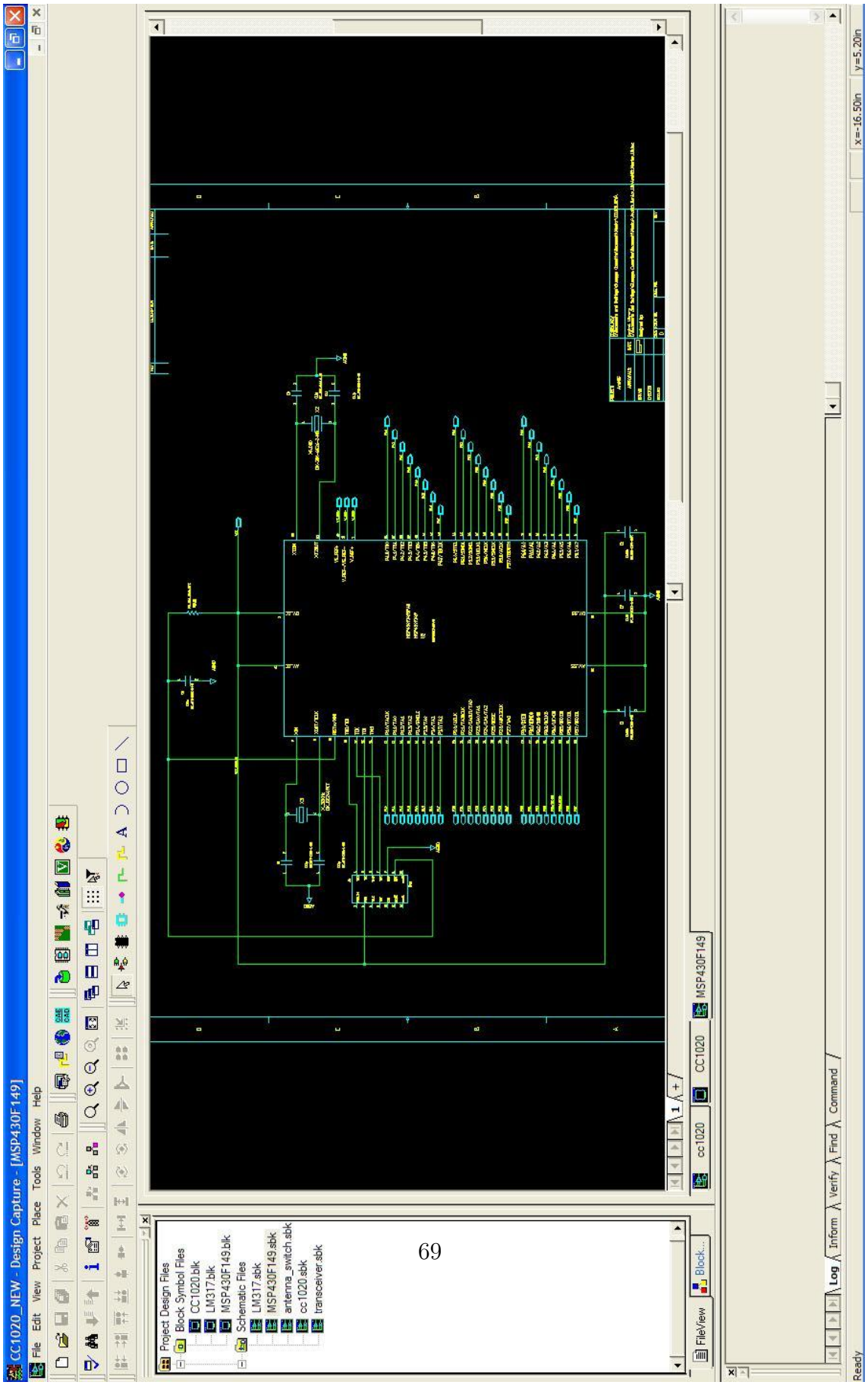

Figura A.2. Schema elettrico del microcontrollore MSP430F149

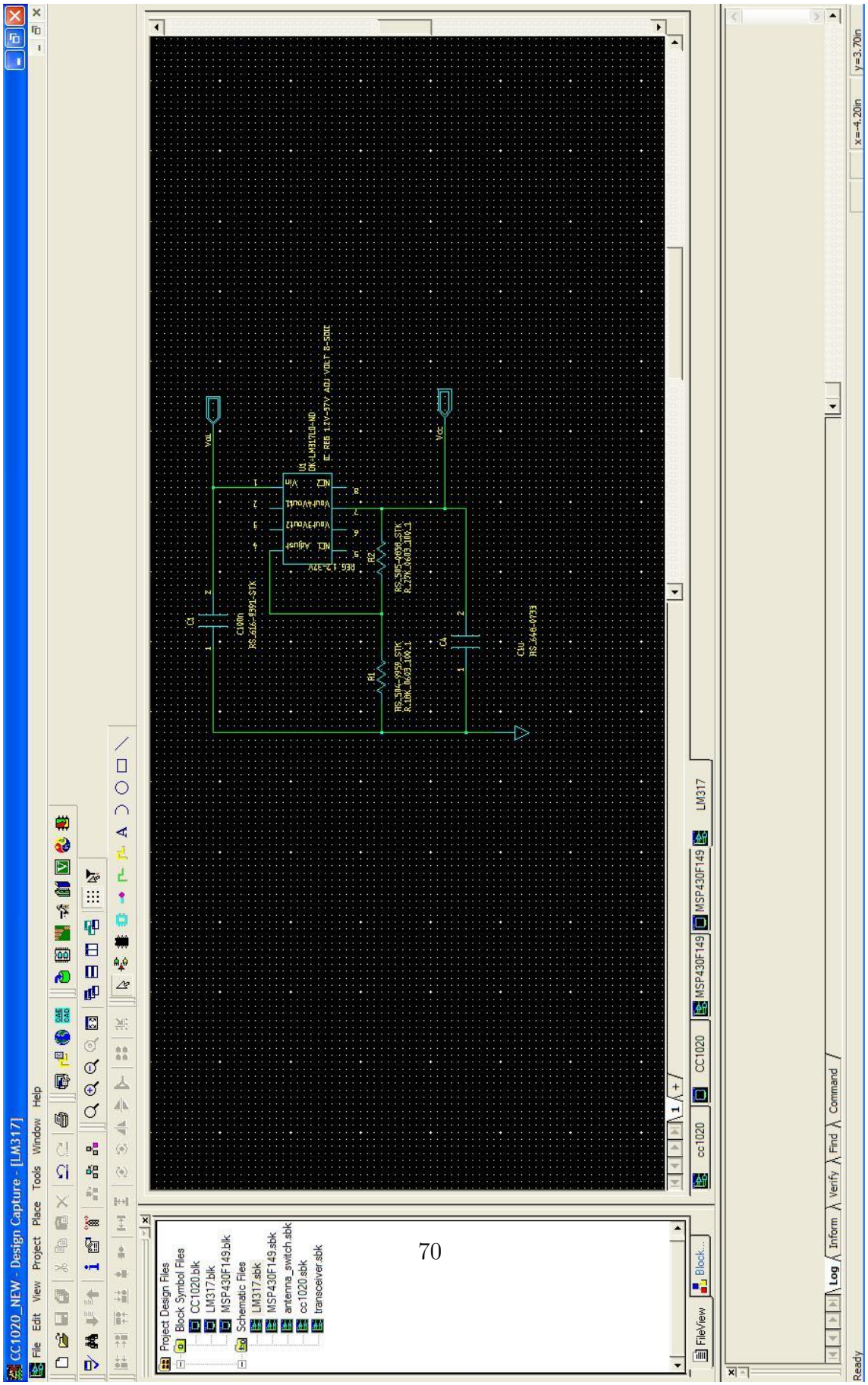

Figura A.3. Schema elettrico del regolatore di tensione LM317

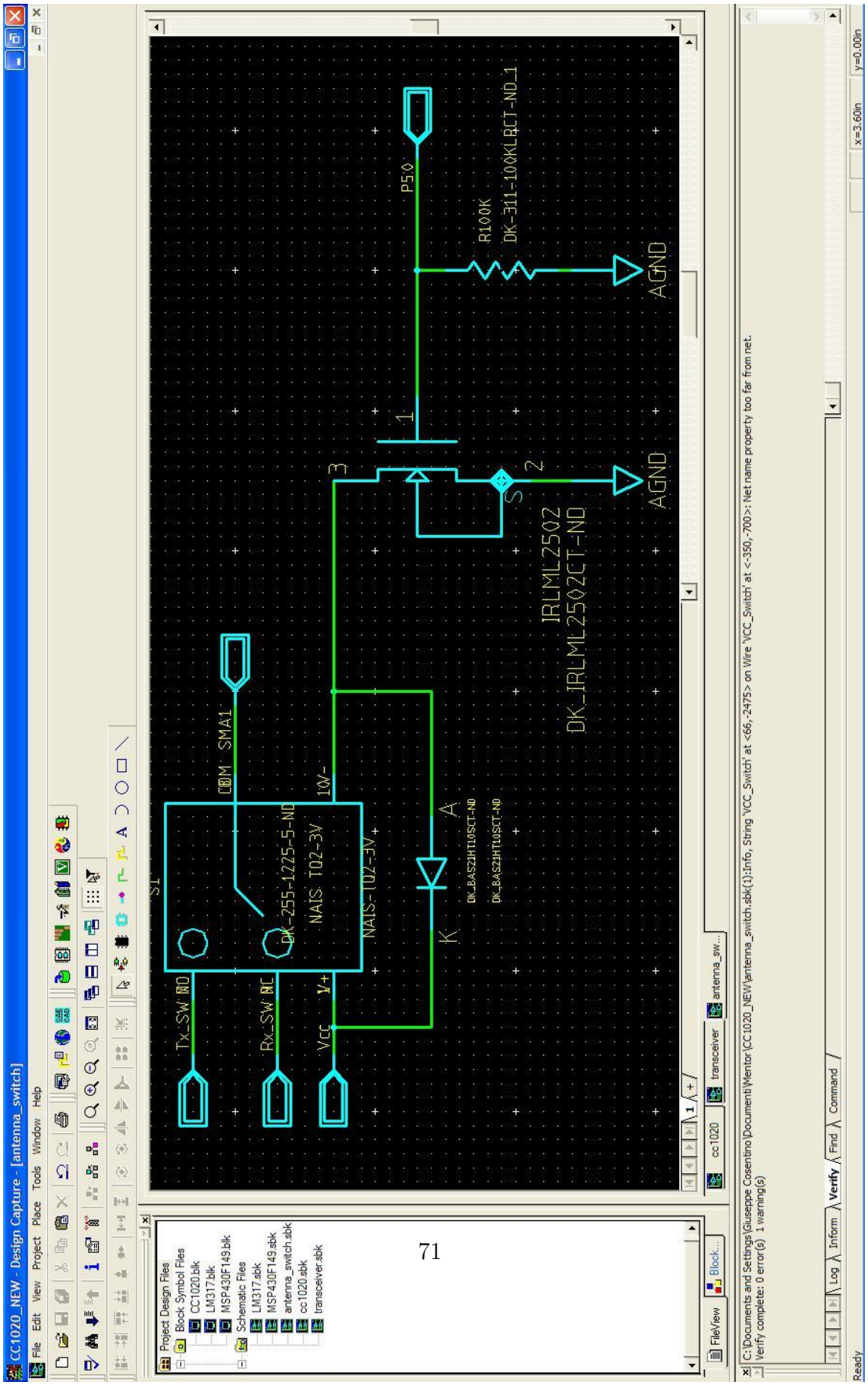

Figura A.4. Schema elettrico del Switch d'antenna

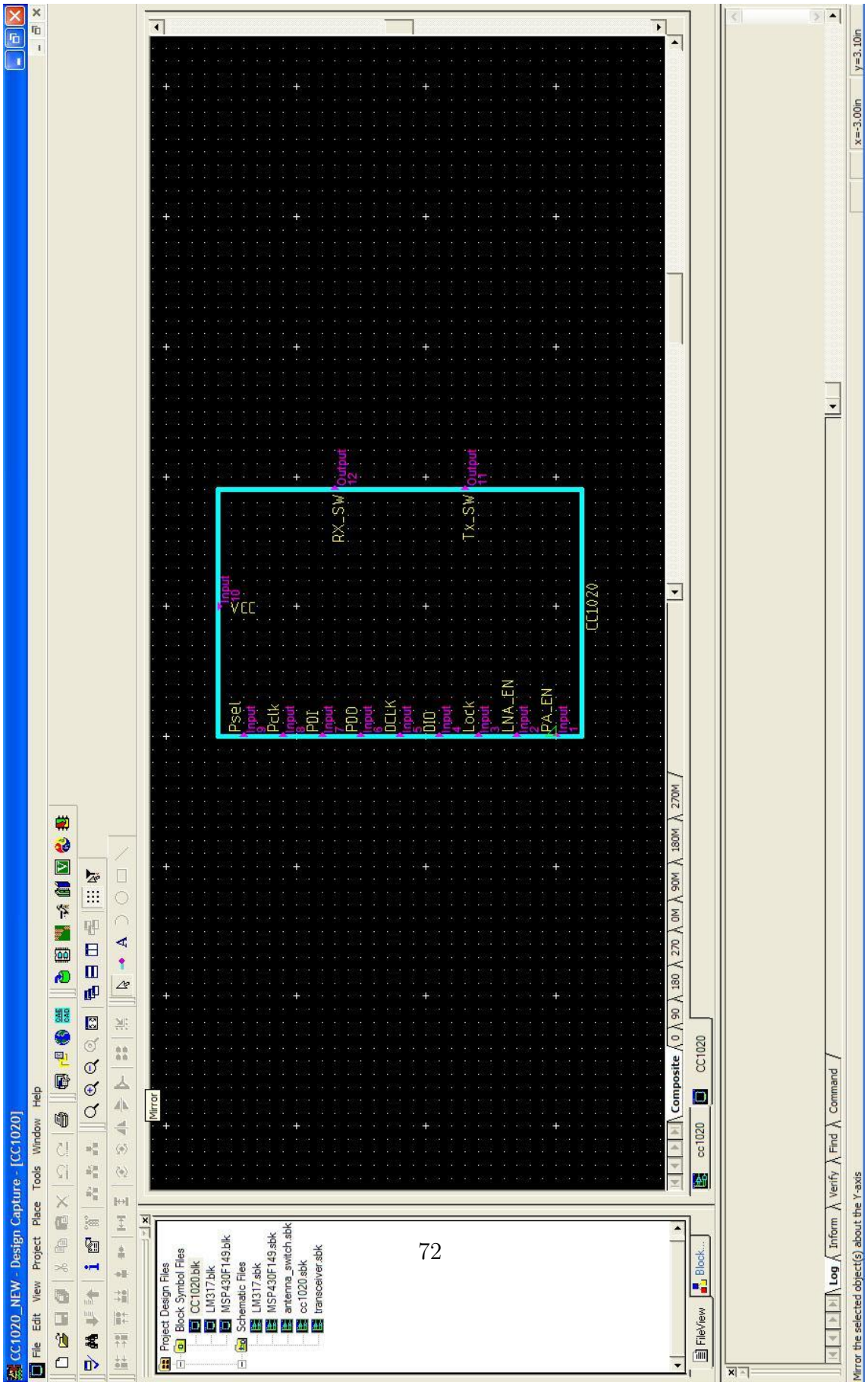

Figura A.5. Blocco rappresentante lo schema elettrico del CC1020

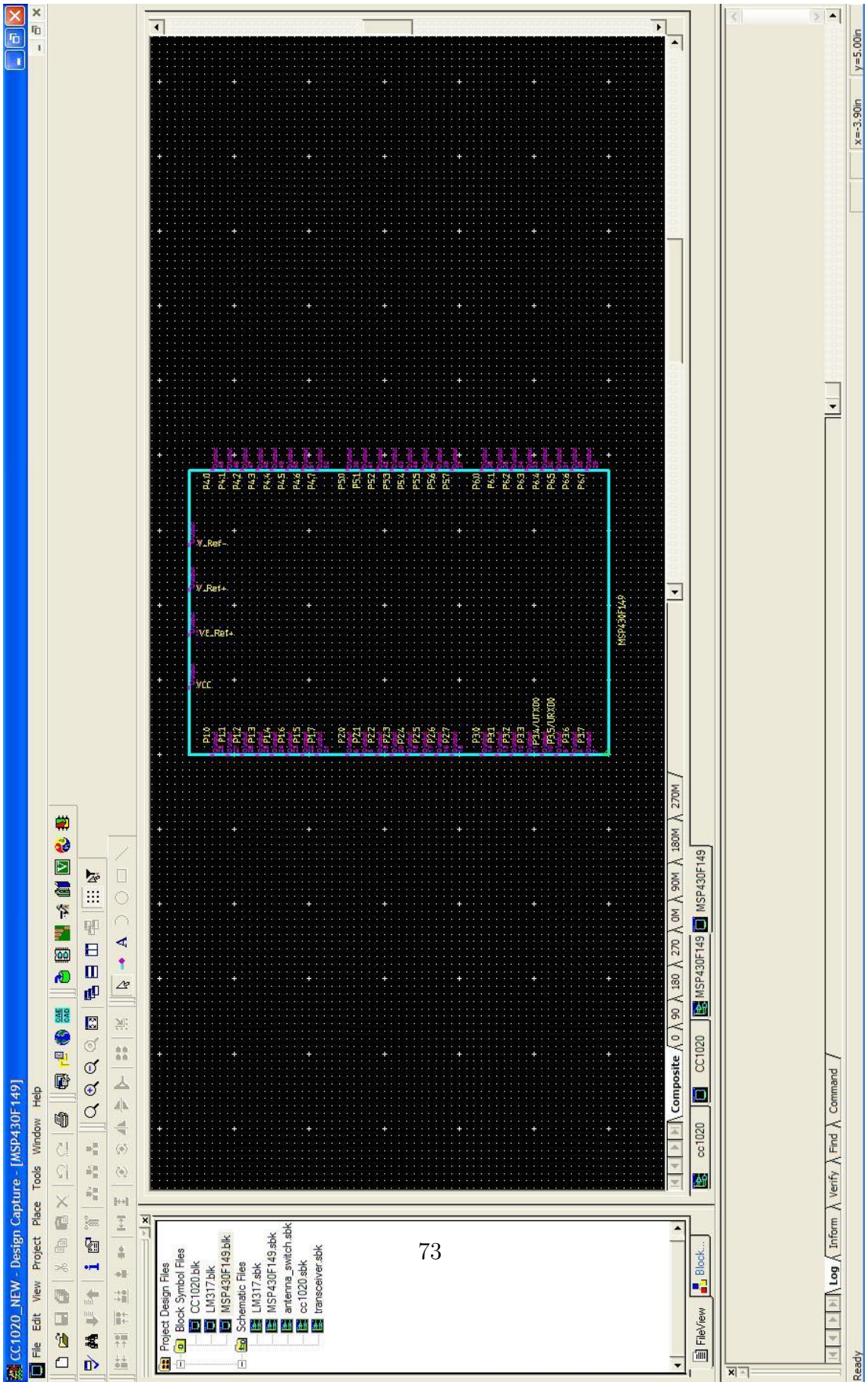

Figura A.6. Blocco rappresentante lo schema elettrico del microcontrollore MSP30F149

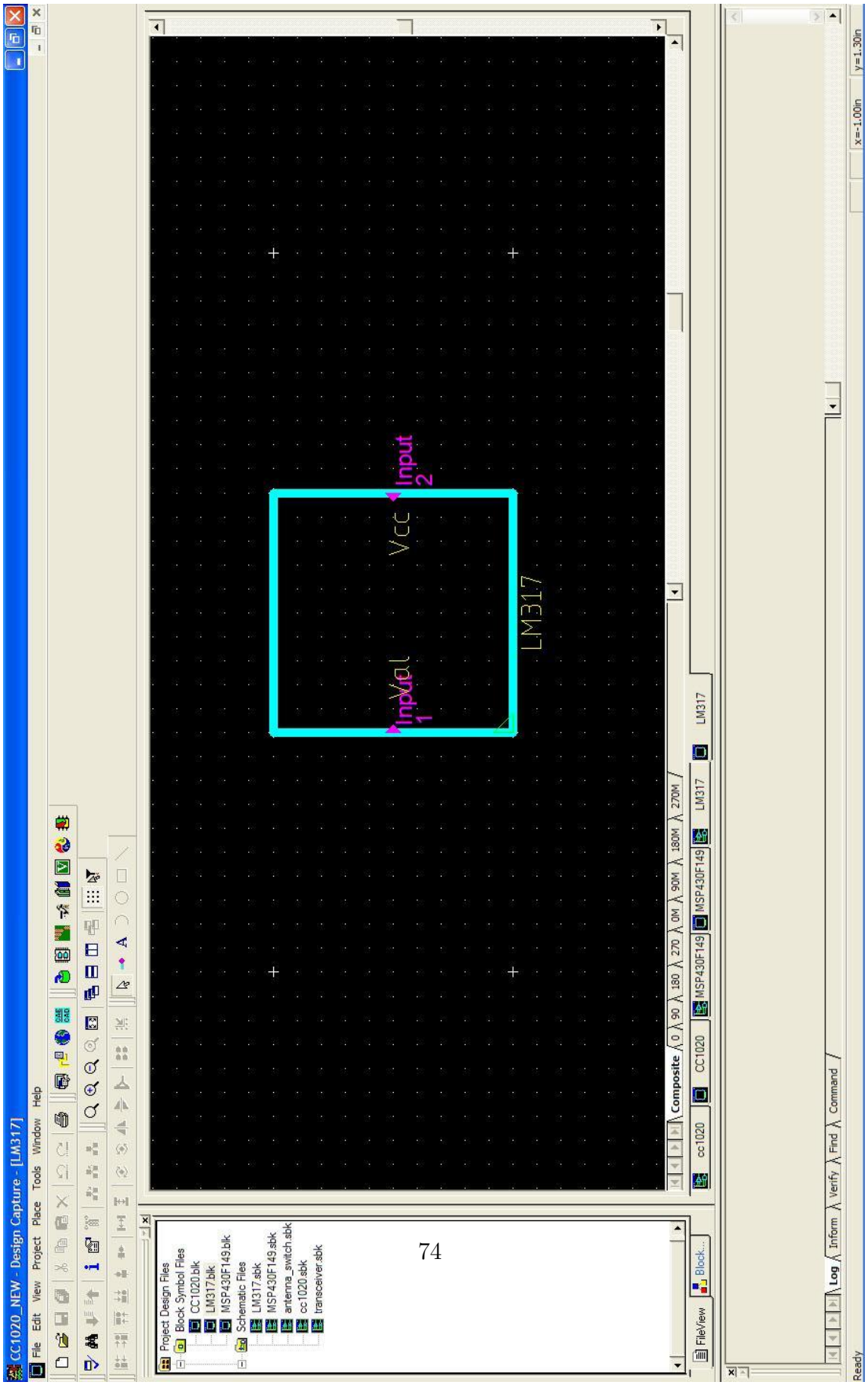

Figura A.7. Blocco rappresentante lo schema elettrico del regolatore di tensione LM317

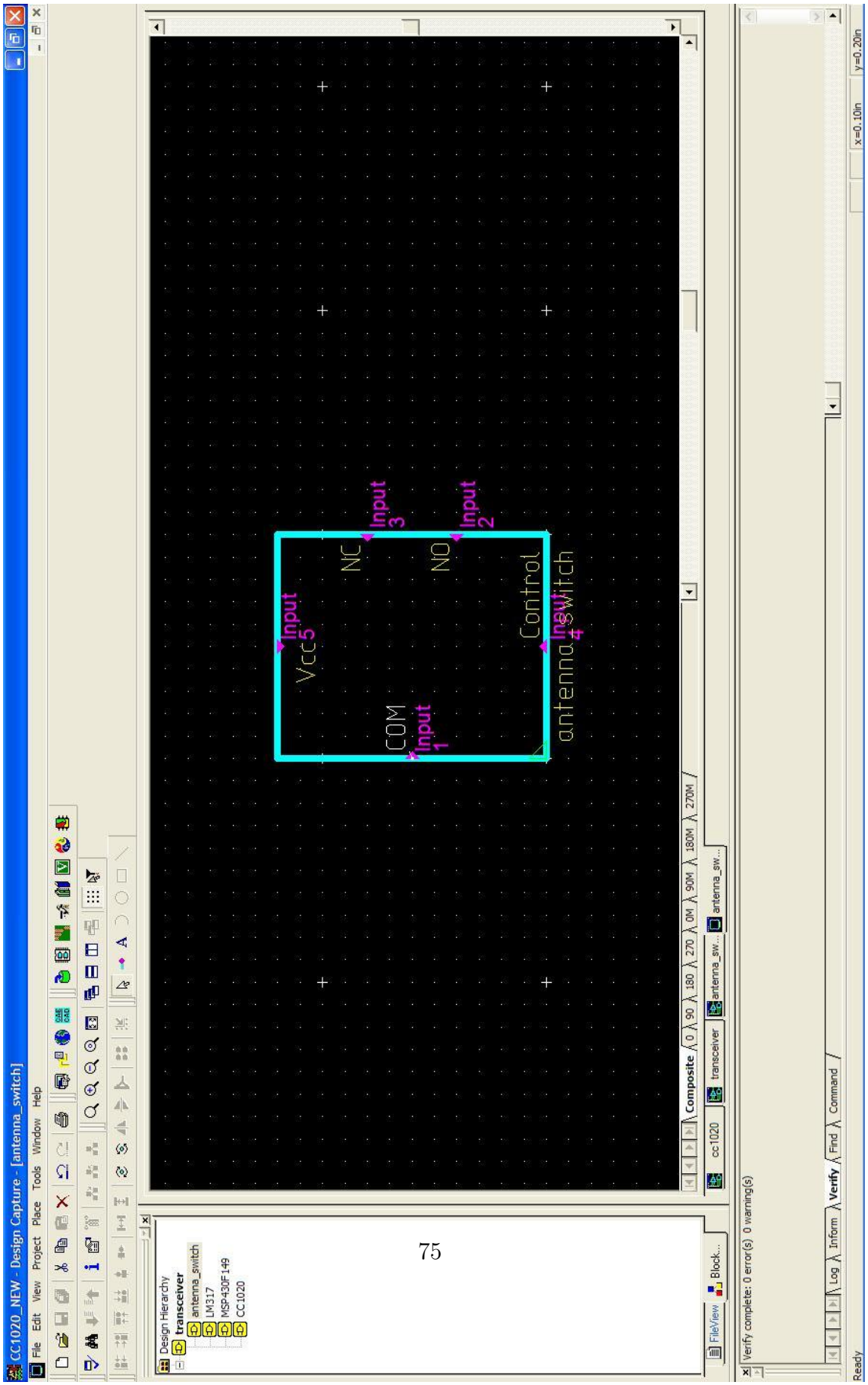

Figura A.8. Blocco rappresentante lo schema elettrico del switch d'antenna

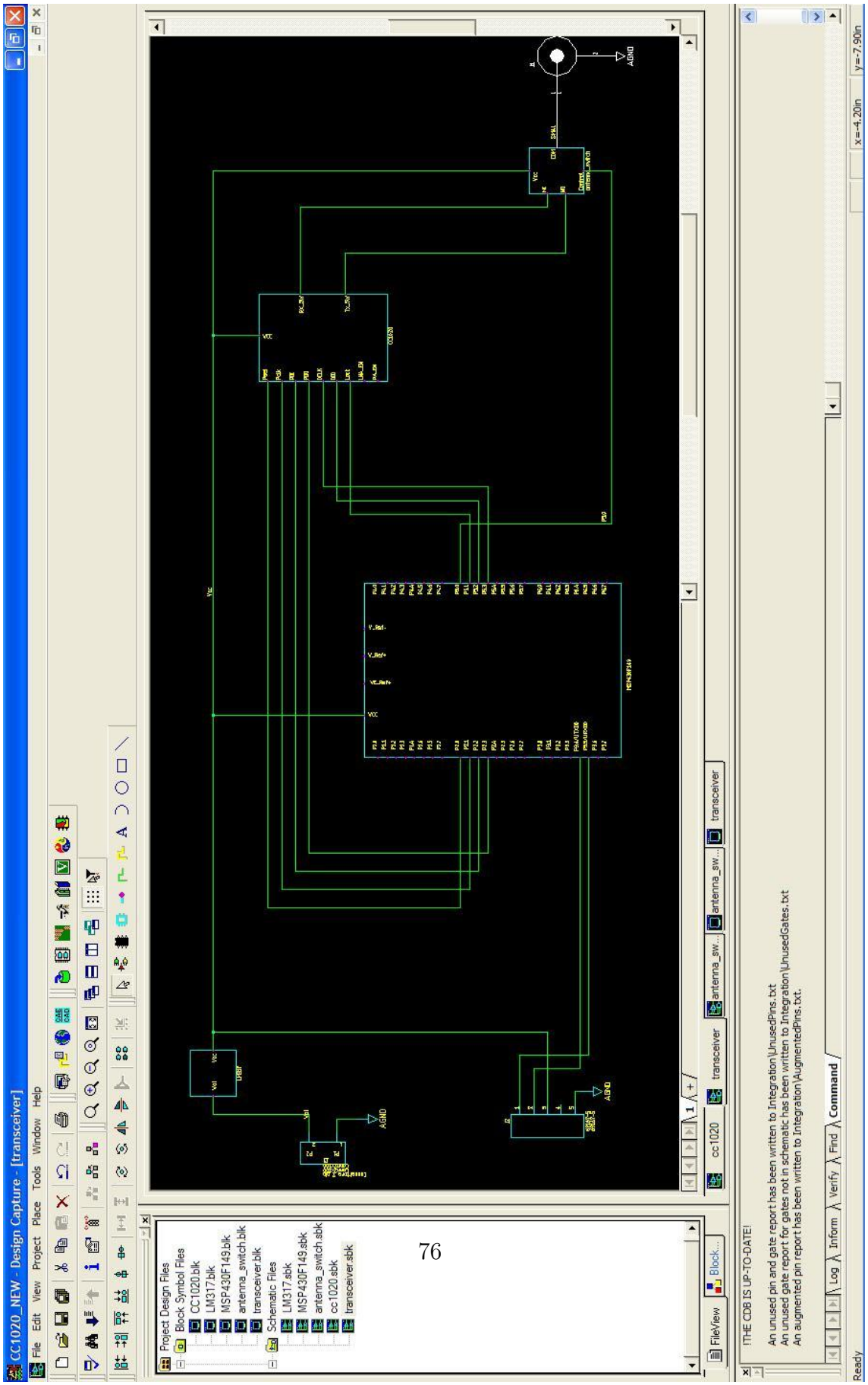

Figura A.9. Interconnessione dei blocchi realizzati, rappresentante lo schema elettrico della scheda

Appendice B

PCB Layout

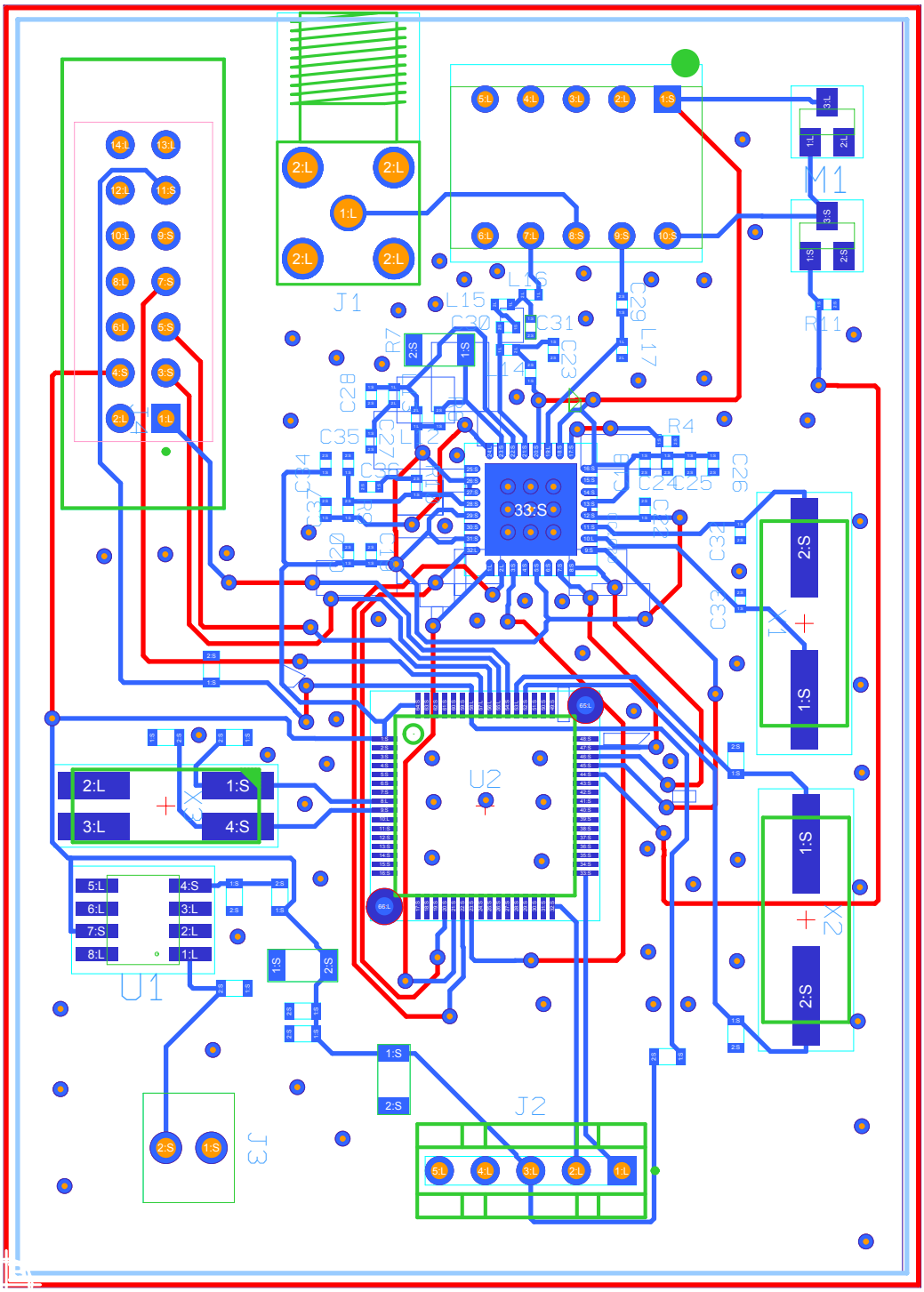

Figura B.1. Generazione del PCB

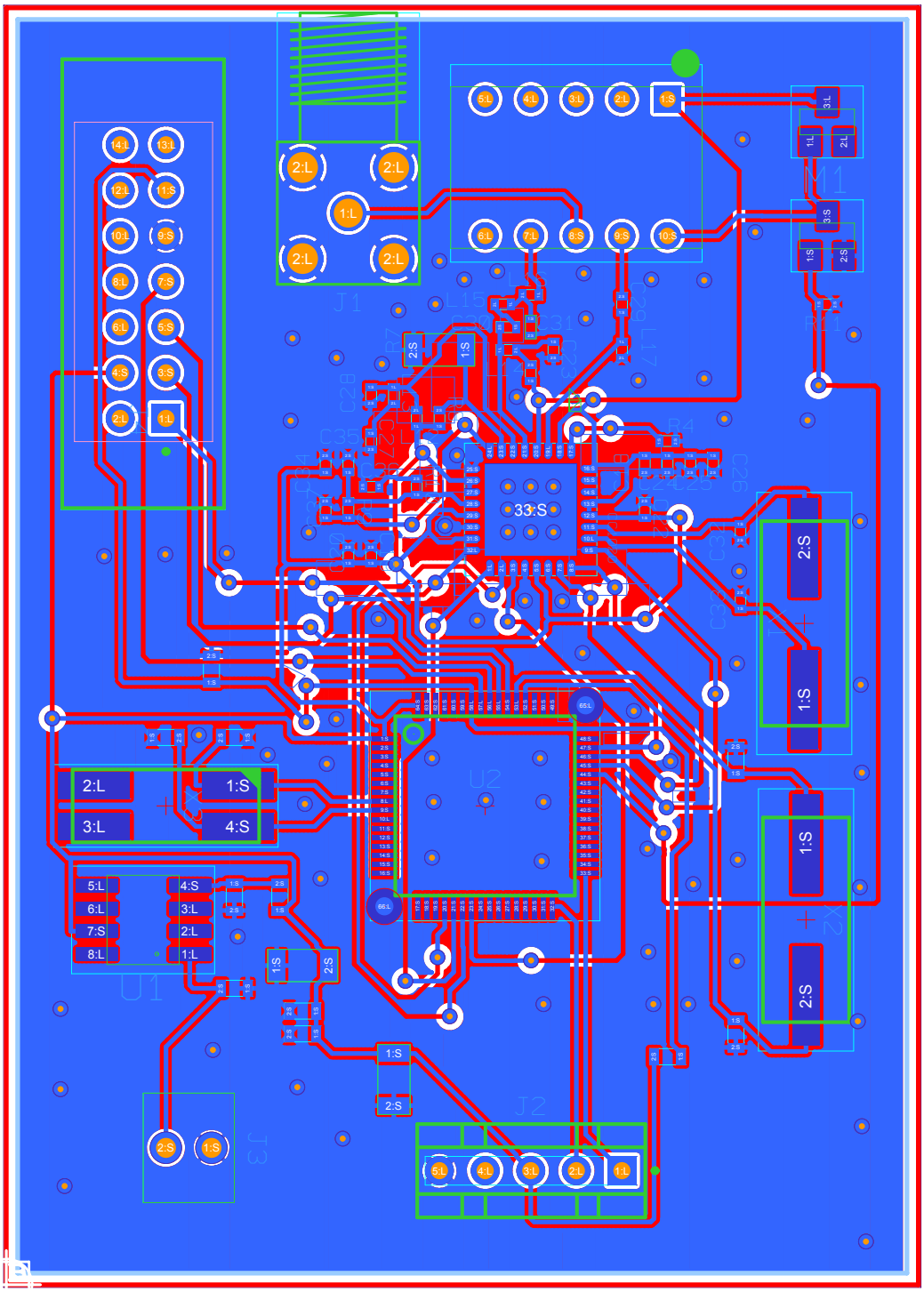

Figura B.2. Top del PCB

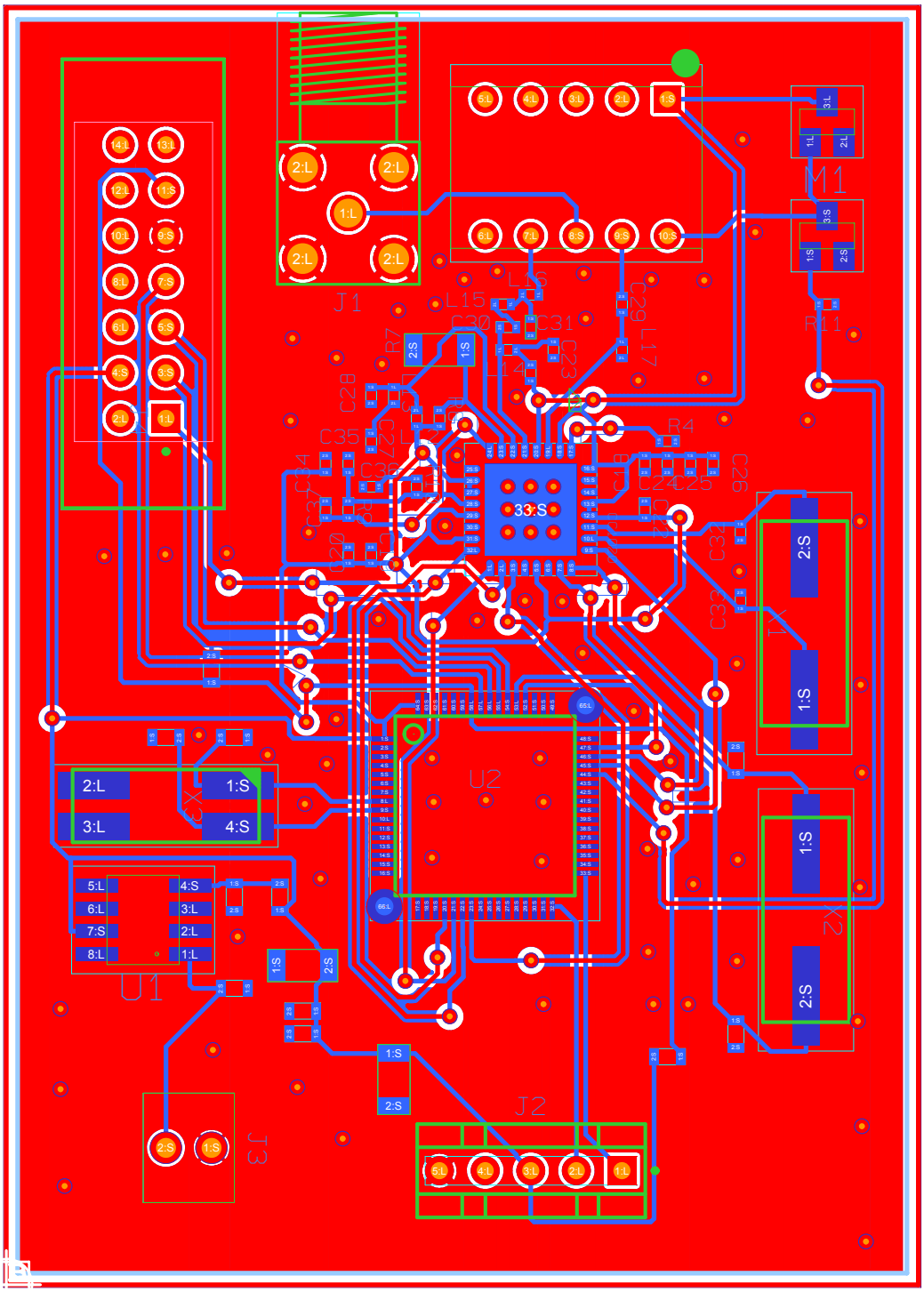

Figura B.3. Bottom del PCB

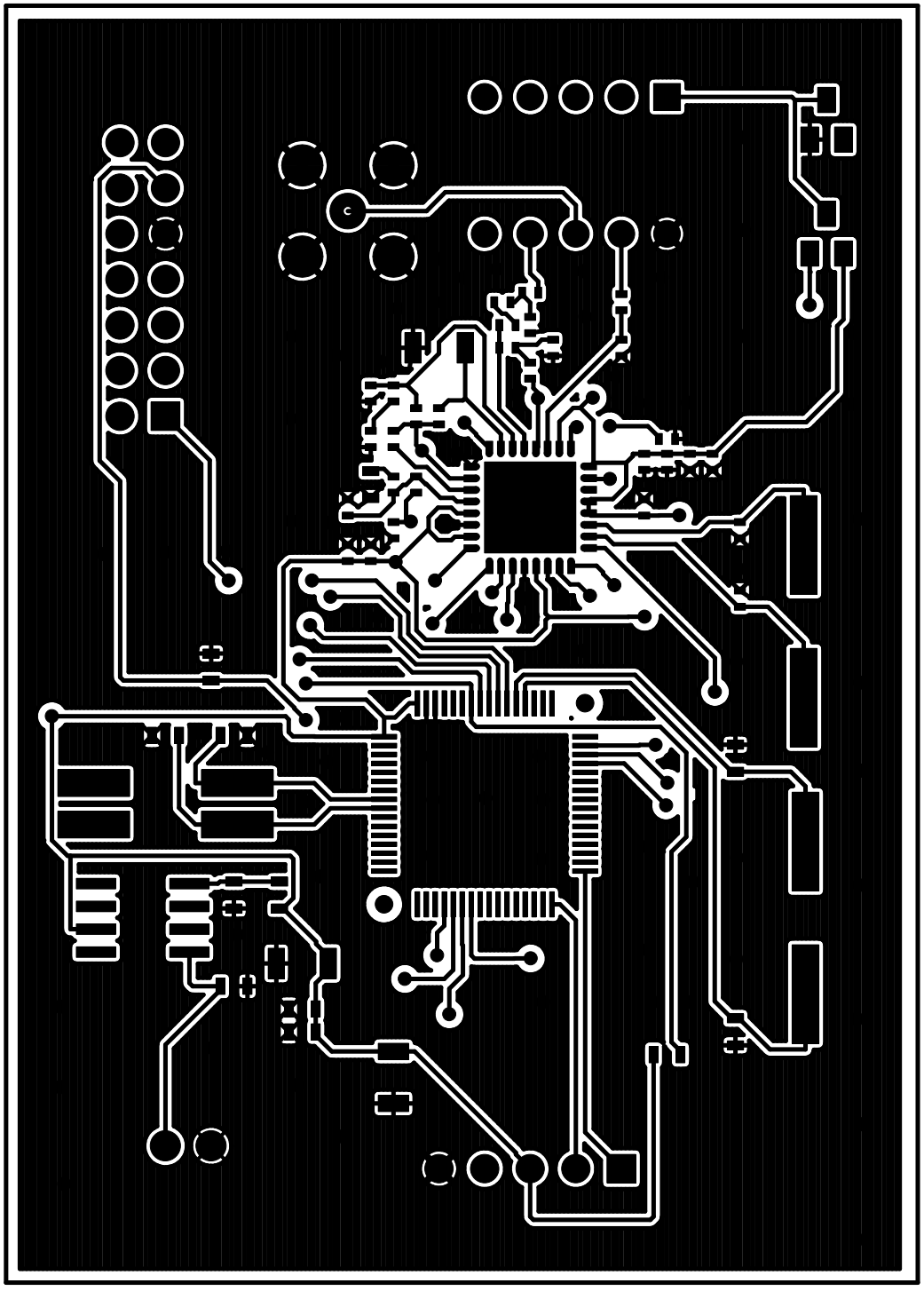

Figura B.4. Top file Gerber

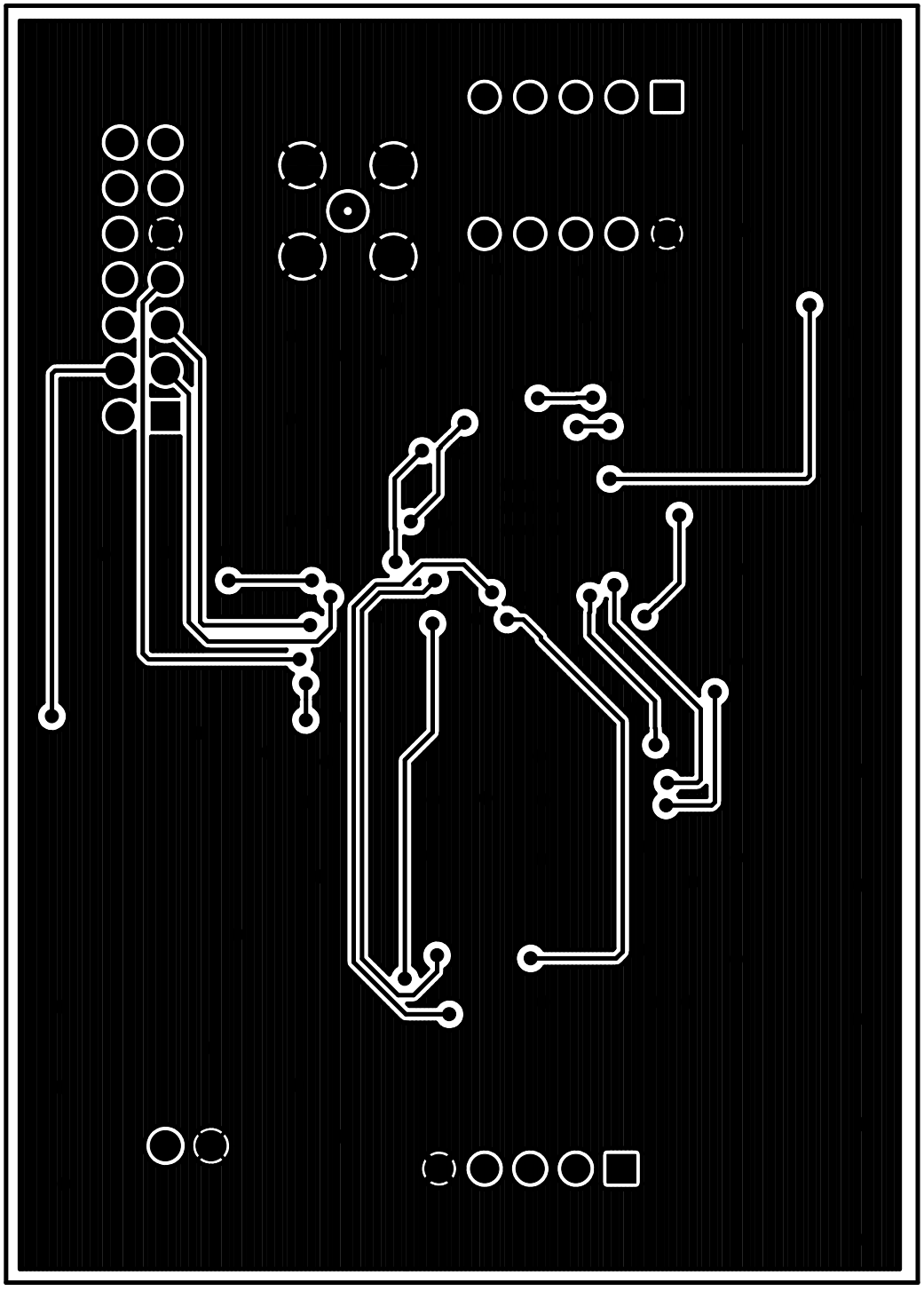

Figura B.5. Bottom del file Gerber

Appendice C

Source Files

C.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

unsigned 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_Tx();

              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);

SPI_Disable();

CC1020_SetupPD();

break;

case TX_CMD:
printUART("TX\n\r");

SPI_Init();

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]);


printUART("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\r

SPI_Disable();
CC1020_SetupPD();

break;
```

```
case RX_CMD:
printUART("RX\n\r");

CC1020_Config_Rx();

SPI_Disable();

CC1020_WakeUpToRX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3);

CC1020_SetUpToRX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3, 0x00);

 /*tmp1=0;

printUART("RSSI:\n\r");
 while(1)
 {
 tmp=CC1020_ReadReg(CC1020_RSSI);
 if ((tmp-tmp1) > 20 ) UART_SendByte (tmp);
 tmp1 = tmp;
 }
 */
 AX25_ReceivePacket(&(test[2]));

 test[0]='R';
 test[1]=10;

 uart_SendPacket(test);
 printUART("Prova\n\r");

 CC1020_SetupPD();
```

```
                break;



                default:
                      printUART("NO\n\r");



            }
        }
        else
        {
            printUART("NO CRC\n\r");

         }


  }


}
```

C.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
```

C.3 CC1020.c

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

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_Tx(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);
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);
CC1020_SetReg(CC1020_DEVIATION,0x1D);
CC1020_SetReg(CC1020_AFC_CONTROL,0xC7);
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);
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);          //IMPORTANT:  SEP_DI_DO = 1 era 4f o 5f
  CC1020_SetReg(CC1020_RESET,0xFF);
  CC1020_SetReg(CC1020_SEQUENCING,0x8F);
  CC1020_SetReg(CC1020_FREQ_2A,0x3A);
  CC1020_SetReg(CC1020_FREQ_1A,0x85);
```

```
CC1020_SetReg(CC1020_FREQ_0A,0x9D);

CC1020_SetReg(CC1020_CLOCK_A,0x39);

CC1020_SetReg(CC1020_FREQ_2B,0x39);
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);
CC1020_SetReg(CC1020_DEVIATION,0x0B);
CC1020_SetReg(CC1020_AFC_CONTROL,0xCC);
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);  //era 20
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,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;
```

```
}

   /*
   Important note:
   In active mode the CC1020 should theoretically
   initiate an internal action/process more or less
   instantly upon receiving any command from e.g. an MCU.
   However, upon sending a [calibration start]
   command to CC1020, tests shows that the [STATUS.CAL_COMPLETE]-signal
   sometimes remains asserted (or idle) close to 100 usec after the command
   has been originally issued. Consequently this process
   must be carefully monitored to avoid premature PLL LOCK monitoring;
   do not proceed with subsequent
   PLL LOCK monitoring before the calibration has actually completed
   inside the CC1020! Errata Note 04 suggests that [calibration start]
   should be monitored by a fixed timeout > 100 usec. However, the
   above method implements an adaptive monitoring of [calibration start],
   which basically waits for the [STATUS.CAL_COMPLETE]-signal to
   initialise/deassert (indicating calibration has actually started)
   before proceeding with monitoring calibration complete and PLL LOCK.
   Chipcon considers both methods safe, and thus leaves it up to the user,
   which one to use.
   */

// Monitor calibration complete
// Monitor lock

TimeOutCounter = 300;

while((CC1020_ReadReg(CC1020_STATUS) & LOCK_CONTINUOUS) !=  LOCK_CONTINUOUS )
{
  TimeOutCounter--;
  if ( TimeOutCounter <= 0 )
```

```
        return LOCK_NOK;
    }

    // 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)

  // 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); Ti

  // 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;
}
```

C.4 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_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
```

C.5 Uart.c

```
//******************************************************************************
//  MSP430x11x1 Demo - Timer_A, UART 115200 Echo, HF XTAL ACLK
//
//  Description: Use Timer_A CCR0 hardware output modes and SCCI data latch
//  to implement UART function @ 115k baud. Software does not directly read and
//  write to RX and TX pins, instead proper use of output modes and SCCI data
//  latch are demonstrated. Use of these hardware features eliminates ISR
//  latency effects as hardware insures that output and input bit latching and
//  timing are perfectly synchronised with Timer_A regardless of other
//  software activity. In the Mainloop the UART function readies the UART to
//  receive one character and waits in LPM0 with all activity interrupt driven.
//  After a character has been received, the UART receive function forces exit
//  from LPM0 in the Mainloop which echo's back the received character.
//  ACLK = MCLK = TACLK = HF XTAL = 8MHz
// //* An external 8MHz XTAL on XIN XOUT is required for ACLK *//
// //* Vcc needs to be 3.6 V for 8MHz MCLK operation *//
//
//                MSP430F1121
//            -----------------
//        /|\|              XIN|-
//         | |                 | 8MHz
//         --|RST          XOUT|-
//           |                 |
//           |   CCI0A/TXD/P1.1|---------->
//           |                 | 115200 8N1
//           |   CCI0B/RXD/P2.2|<----------
//
//  M. Buccini
//  Texas Instruments Inc.
//  Feb 2005
//  Built with IAR Embedded Workbench Version: 3.21A
```

```
//*****************************************************************************
//#include <msp430x11x1.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;
  UBR00 = 0xA0;                             // 2MHz 9600
  UBR10 = 0x01;                             //
  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;
}
```

C.6 Uart.h

```
#ifndef 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
```

C.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
  U1BR0 = 0x00;                          // SPICLK = SMCLK/2
  U1BR1 = 0x00;
  U1MCTL = 0x00;
  ME2 |= USPIE1;                         // Module enable
  U1CTL &= ~SWRST;                       // SPI enable

}

void SPI_Disable(void)
{

  P5SEL &= ~(DIO | DCLK | LOCK_BIT);     // P5.1,2,3 SPI option select
  P5DIR &= ~(DIO | DCLK | LOCK_BIT);      //set SPI pin as input

  ME2 &= ~USPIE1;                         // Module disable
  U1CTL |= SWRST;                         // SPI disable
```

```
}

// 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));                // USART0 TX 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;

  }

}
```

C.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
```

C.9 AX25.c

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

short LFSR;
int  totalbytes;
char ONEScount = 0;
char lastBit = 0;
char nextBit;
char isFlag;
char data[150] = "UL Lafayette C.A.P.E. AX.25 Protocol UI Frame 2006";



//#define PTT   PORTBbits.RB6


 //  OutputUSART();


#define INTSCRAMBLER

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
};

/*const char prologo[PROLOGO_LEN] = {
  'A' << 1, 'L' << 1, 'L' << 1, ' ' << 1, ' ' << 1, ' ' << 1, 0xE0,
  'P' << 1, 'I' << 1, 'C' << 1, 'P' << 1, 'O' << 1, 'T' << 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(char *cp, int len)
{

    while (len--)
        fcs.u = ((unsigned short)fcs.b.crc2) ^ fcstab[(fcs.b.crc1 ^ *cp++)];
```

```
}


void AX25_ComputeFCS(char * packet, int packet_len)
{
        INITFCS();
        pppfcs(packet, packet_len);
        CPLFCS();
        //packet[packet_len] = fcs.b.crc1;
        //packet[packet_len+1] = fcs.b.crc2;
}

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
                /* send it */
                //tx = out ^ 0x1;

                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);

                }

#ifdef CRCINLINE
                if (flag == 0)
                {
                        //fcs ^= tmp<<15; // we don't need masking, 'cause is already masked
                        //fcs = (fcs & 0x80) ? (fcs<<1)^POLYNNOMIUM_INV : (fcs<<1);
                        fcs ^= tmp;     // we don't need masking, 'cause is already masked
                        fcs = (fcs & 0x01) ? (fcs>>1)^POLYNNOMIUM : (fcs>>1);
                } else {
                        fcs = 0xFFFF;
                        ones_cnt = 0;
                }
#else
                if(flag) ones_cnt=0;
#endif
        }
}

void AX25_SendPacket(unsigned char * packet, unsigned int packet_len)
{
  int i;

        scrambled1 = 0;
        scrambled2 = 0;
```

```
#ifndef INTSCRAMBLER
        scrambled3 = 0;
#endif

        out = 0;

        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 */



}
```

```
void AX25_ReceivePacket(unsigned char *data)
 {
     int i;
     char j;
     char byte = 0x00;

     char DIN, A, B, Y;

 di_nuovo:

     ONEScount = 0;
     lastBit = 0;
     totalbytes = 0;

     while (byte != 0x7E)
     {

     while( (P5IN & DCLK) == 0 );         // wait for CC1020 clock to rise and then

     TIMER_Wait_us(500);           // wait for databit to stable 50 us

        DIN = ((P5IN & LOCK_BIT) != 0);   // sample nextBit from CC1020 DIO

        byte = (byte << 1) | DIN;

        while( (P5IN & DCLK) == 1 );     // wait for next clock cycle


     }


     while (byte == 0x7E)
     {
```

```
    for (lastBit = 0; lastBit < 8; lastBit++)
    {

      while( (P5IN & DCLK) == 0 );         // wait for CC1020 clock to rise and then

     TIMER_Wait_us(50);           // wait for databit to stable 50 us

       DIN = ((P5IN & LOCK_BIT) != 0);     // sample nextBit from CC1020 DIO

       byte = (byte << 1) | DIN;

       while( (P5IN & DCLK) == 1 );     // wait for next clock cycle

    }
}

data[totalbytes] = byte;
totalbytes++;

while ((byte != 0x7E) && (totalbytes < 200))
{

    for (lastBit = 0; lastBit < 8; lastBit++)
    {

      while( (P5IN & DCLK) == 0 );    // wait for CC1020 clock to rise and then

      TIMER_Wait_us(50);              // wait for databit to stable 50 us

       DIN = ((P5IN & LOCK_BIT) != 0);// sample nextBit from CC1020 DIO

       byte = (byte << 1) | DIN;
```

```
            while( (P5IN & DCLK) == 1 );   // wait for next clock cycle

        }

        data[totalbytes] = byte;
        totalbytes++;
    }



    return;
```

```
// Check for flag ------------------------------------------------------------
    while ( byte != 0x7e )              // Check for flag
    {
        A = LFSR & 0x01;
        B = (LFSR >> 5) & 0x01;
        Y = A ^ B;

        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

      for (j = 0; j < 0x0A; j++) asm("nop;"); // wait for databit to stable 50 us

       DIN = P5IN & LOCK_BIT;           // sample nextBit from CC1020 DIO
       nextBit = DIN ^ Y;

        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

        LFSR >>=1;

        if (DIN)
           LFSR |= 0x010000;

        while( (P5IN & DCLK) == 1 );     // wait for next clock cycle
    }
     // End flag check ------------------------------------------------------------

     //PORTBbits.RB6 = 1;
    //UART_SendByte(byte);

// Receive bytes until the end flag is reached -------------------------------------
    // Note: This breaks if more than one consecutive flag is being sent.  This can be
    // fixed later if need be.
```

```
     i = 0;
   byte = 0x00;                // reset byte so it is no longer 0x7e
     totalbytes = 0;

//
     byte = RxByte();
     if (byte != 0x7E)   goto di_nuovo;

printUART("000\n\r");

//   while ( byte != 0x7e )      // Start reading bytes until flag is reached
    while (( byte != 0x7e ) && ( i < 10 ))   // Start reading bytes until flag is reached
    {
       byte = RxByte();

        if ( byte != 0x7e)
         {
           data[i] = byte;
       }
        i = i + 1;
        totalbytes = i;

     }
 } // End void RxPacket(void)


char RxByte(void)
{
    int i;
    char byte = 0x00;

    char DIN, A, B, Y;
```

```
for ( i = 0; i < 8; i++ )               // for all 8 bits
 {
     A = LFSR & 0x01;
    B = (LFSR >> 5) & 0x01;
    Y = A ^ B;

     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 & LOCK_BIT;       // sample nextBit from CC1020 DIO
    nextBit = DIN ^ Y;

    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;

    LFSR >>=1;
    if (DIN)
        LFSR |= 0x010000;


     while( (P5IN & DCLK) == 1 );     // 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 )
        {
           A = LFSR & 0x01;
          B = (LFSR >> 5) & 0x01;
            Y = A ^ B;

           // Check the next bit to see if it is a bitstuffed zero; if it is not then it
           // is probably the flag
          while( DCLK == 0 );     // wait for CC1020 clock to rise and then
           DIN = DIO;          // sample nextBit from CC1020 DIO

            nextBit = DIN ^ Y;

           if (nextBit != lastBit)     // then zero needs to be skipped
            {
              lastBit = nextBit;      // fix lastBit for NRZI
               ONEScount = 0;          // reset ONEScount

               LFSR >>=1;
               if (DIN)
                   LFSR |= 0x010000;


                while( DCLK == 1 );  // wait for next clock cycle
            }
           else                      // otherwise, flag byte has been encountered
            {
                ONEScount = 0;       // Reset ones since we're assuming a flag has been received
                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;

}
```

C.10 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



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

C.11 timer.c

```
#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



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

C.12 timer.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



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

Bibliografia

[1] Twiggs B., Puig-Suari J., CUBESAT Design Specification Document, Standford University and Polytechnical Institute. www.cubesat.org.

[2] Reyneri L., PICPOT Satellite Universitario del Politecnico di Torino, documento di specifiche dei sottosistemi elettronici, Versione 6.

[3] S.Speretta, L.Reyneri, C. Sansoè, M.Tranchero, C. Passerone, and Dante Del Corso, 'Modular Architecture for satellites' 58th International Astronautical Congress, September 2007.

[4] Datasheet: CC1020 Single Chip Low Power RF Transceiver for Narrowband System.

[5] DataSheet: MSP430F1xxMixed Signal Microcontroller.

[6] User Manual: CC1020/1070DK Developement kit.

[7] Andrea De Nigris, Tesi di Laurea 'Definizione dell'architettura a radiofrequenza del satellite PiCPoT II', Politecnico di Torino, Settembre 2007.

[8] Giulia Suriani, Tesi di Laurea 'Progetto di un ricetrasmettitore per satellite universitario',Politecnico di Torino, Settembre 2007.

[9] Giuseppe Benenati, Tesi di Laurea 'Progettazione del sottosistema di comunicazione a 2.4 GHz per PiCPoT II',Politecnico di Torino, Gennaio 2007.

[10] Stefano Speretta, Tesi di Laurea 'Collaudo ed Integrazione del satellite universitario PiCPoT', Politecnico di Torino, Novembre 2006.

[11] sito della IARU: http//www.iaru.org.

[12] sito della chipcon: http//www.chipcon.com.