UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA ELETTRONICA

DRIVER KERNEL-MODE LINUX SU RASPBERRY PI 3 PER MODULI RADIO RFM69HCW

KERNEL-MODE LINUX DRIVER ON RASPBERRY PI 3 FOR RFM69HCW RADIO MODULES

TESI DI LAUREA DI:

DANILO PICHILLI

RELATORE:

PROF.GIORGIO BIAGETTI

ANNO ACCADEMICO 2019/2020

INDICE

INDICE 1

1.INTRODUZIONE 3

2.RASPBERRY PI 3 MODEL B 4

2.1 Introduzione 4

2.2 Fondazione 5

2.3 Software 5

2.4 Raspberry Pi 3 Model B 6

3.RFM69HCW 9

3.1 Introduzione 9

3.2 Caratteristiche 10

3.3 Applicazioni 11

4.SPI 12

4.1 Introduzione 12

4.2 I quattro segnali 13

4.3 Collegamento dispositivi slave 13

4.4 Come avviene la comunicazione 16

4.5 SPI su Raspberry 17

5. DEVICE DRIVER 18

5.1 Introduzione 18

5.2 Costruire moduli 21

5.3 Char Drivers 25

6. RFM69HCW DRIVER 34

6.1 Dichiarazioni preliminari 34

6.2 Inserimento e rimozione **42**

6.3 State Machine **53**

6.4 File operations **56**

6.4.1 Open **56**

6.4.2 Scrittura **58**

6.4.3 Lettura **68**

6.5 Installazione **73**

6.6 Funzionamento **76**

7.CONCLUSIONI 78

RIFERIMENTI 80

1.Introduzione

Lo scopo di questo progetto è stato quello di realizzare un driver per il Kernel di un sistema operativo basato su Linux. In sostanza si tratta di far comunicare efficientemente una piattaforma, che in questo caso è la Raspberry Pi 3 Model B, con un modulo digitale a frequenze ISM (Industrial, Scientific and Medical), nello specifico l'RFM69HCW prodotto dalla HopeRF.

Un driver, in informatica, indica l'insieme di procedure, che permette ad un sistema operativo di pilotare un dispositivo hardware. Esso permette al sistema operativo di utilizzare l'hardware senza sapere come esso funzioni, ma dialogandoci attraverso un'interfaccia standard, i registri del controllore della periferica, che astrae dall'implementazione dell'hardware e che ne considera solo il funzionamento logico. In questo modo hardware diverso costruito da produttori diversi può essere utilizzato in modo intercambiabile.

Ne consegue che un driver è specifico sia dal punto di vista dell'hardware che pilota, sia dal punto di vista del sistema operativo per cui è scritto. Non è possibile utilizzare driver scritti per un sistema operativo su uno differente, perché l'interfaccia è generalmente diversa.

Il Kernel sul quale si ha intenzione di andar ad implementare il driver è quello di un sistema operativo che si basa, come già detto precedentemente, su Linux. Quest'ultimo può essere definito in generale come il primo rappresentante del software cosiddetto "libero", ovvero quel software che viene distribuito con una licenza che ne permette non solo l'utilizzo da parte di chiunque ed in qualsiasi circostanza ma anche la modifica, la copia e l'analisi.

Il supporto hardware sul quale è installato il sistema operativo è Raspberry Pi 3 Model B, un mini computer "single board" ovvero un computer implementato su un'unica scheda elettronica che può essere utilizzato per gli scopi più disparati.

Come già detto su questo mini-computer andrà installato il sistema operativo, `Raspberry Pi OS` nello specifico, e tramite dei pin, situati esternamente alla board, andremo collegare il modulo radio. Quest'ultimo e la Raspberry comunicheranno tramite interfaccia SPI e per gestire il tutto da un pc si utilizzeranno i protocolli SSH e

FTP, che servono per visualizzare il terminale di Raspbian sul monitor e trasferire i file da computer a Raspberry.

Lo sviluppo del progetto è stato agevolato dalla presenza di un altro progetto sviluppato precedentemente da un altro studente: si tratta appunto di un driver per Kernel Linux realizzato per il modulo radio RFM23B, appartenente alla stessa famiglia del modulo radio di nostro interesse (RFM69HCW).

Questo è stato un pilastro per una base di partenza solida sulla quale è stato possibile costruire quello che è diventato, passo dopo passo, un progetto funzionante.

Un altro grande aiuto è stato fornito da un libro (Linux Device Drivers, Third Edition, Corbet – Rubini – Kroah-Hartman) che spiega i fondamenti per creare un driver per Linux; senza parlare dei forum su Internet e dei siti vari ai quali si è fatto affidamento per la risoluzione di alcuni problemi e per l'apprendimento di porzioni di codice. Oltre a questi c'è stata la grande mano fornita dal professore, nonché tutor per il tirocinio, Giorgio Biagetti.

2.Raspberry Pi 3 Model B

2.1 Introduzione

Il Raspberry Pi 3 Model B è un single-board computer (un calcolatore implementato su una sola scheda elettronica) sviluppato nel Regno Unito dalla Raspberry Pi Foundation. Il suo lancio al pubblico è avvenuto il 29 febbraio 2016.

Finora sono state prodotte 12 versioni: Pi 1 Model A, A+, Pi 1 Model B, B+, Pi 2 Model B, Pi Zero ,Pi 3 Model B,Pi 3 Model B+, Pi 3 Model A+, Pi 4 Model B, Pi Zero W, Pi Zero WH con prezzi da 5 a 35 dollari statunitensi ad eccezione del modello Pi 4 che ha diversi tagli di memoria ram.

L'hardware si basa su un System-on-a-chip (SoC) Broadcom (BCM2835), che incorpora un processore ARM, una GPU VideoCore IV e 1 Gigabyte di memoria. Il progetto non prevede né hard disk né una unità a stato solido, affidandosi invece a una scheda SD per il boot e per la memoria non volatile.

La scheda è stata progettata per ospitare sistemi operativi basati sul kernel Linux o RISC OS. [1]

2.2 Fondazione

Lo sviluppo del dispositivo è portato avanti dalla Raspberry Pi Foundation fondata nel maggio 2009, organizzazione di beneficenza registrata presso la Charity Commission for England and Wales. Il suo scopo è quello di "promuovere lo studio dell'informatica e di argomenti correlati, soprattutto a livello scolastico, e di riportare uno spirito di divertimento nell'apprendimento del computer".

Le prime concezioni del Raspberry Pi, nel 2006, si basavano sul microcontrollore Atmel ATmega644.

Eben Upton della Broadcom costituì un gruppo di insegnanti, accademici e appassionati di computer per creare un oggetto capace di incoraggiare i ragazzi, fornendo loro conoscenze, abilità operative necessarie e ispirazione.

Il 19 febbraio 2012 la Raspberry Pi Foundation mise a disposizione un proof of concept di un'immagine disco caricabile su SD Card per produrre un sistema operativo preliminare. L'immagine si basava su Debian 6.0 (Squeeze), con LXDE come desktop environment e Midori come browser e conteneva vari strumenti di programmazione.

La fondazione rilasciò nel 2016 un sistema operativo basato su Fedora e una versione di Arch Linux.

Nello stesso anno venne prodotto un sistema operativo basato su Debian che prese il nome in Raspbian.

Nel maggio 2020 la Fondazione decise di cambiare il nome di quest'ultimo in "RaspBerry Pi Os". [1]

2.3 Software

Il Raspberry Pi è diventato in breve tempo molto popolare tra gli hobbisti di informatica ed elettronica, sia per il prezzo contenuto sia per la sua grande versatilità. Il processore ARM supporta molti sistemi operativi (gran parte basati su Linux) e questo permette ad ognuno di lavorare nell'ambiente più consono. Tra questi i più utilizzati sono Raspberry Pi Os (una versione più leggera di Debian), Pidora (figlia della più famosa Fedora), Arch Linux ARM (rivisitazione di Arch Linux). Poi ci sono

anche sistemi operativi che girano nativamente, come Android 4.0, Google Chromium OS, alcune distribuzioni di UNIX come Free BSD e tante altre che sono nominate in una lista del sito RaspberryPi.org. Comunque il sistema operativo più utilizzato è Raspberry Pi Os (Raspbian per versioni precedenti) questo perché Debian (la "madre") è molto famosa e ben documentata, dunque è molto più semplice trovare soluzioni e supporto in caso di problemi durante l'utilizzo. [1] [2]

2.4 Raspberry Pi 3 Model B

Parte superiore Raspberry Pi 3 Model B

La Raspberry Pi 3 Model B presenta alcune novità rispetto alla versione precendente Raspberry Pi2:

-Wireless integrato

-Bluetooth integrato

-Nuovo processore quad-core ARMv8 1.2 Ghz a 64-bit

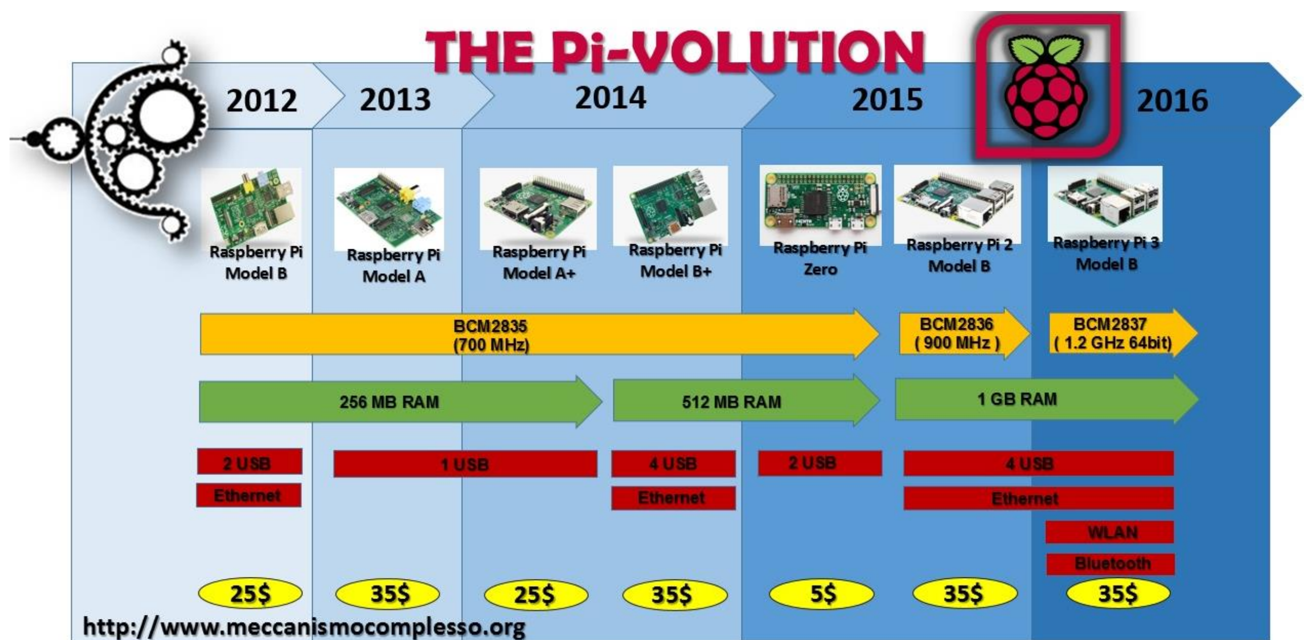
-Alimentazione

Con l'integrazione del wirless LAN nella scheda non sarà più necessario utilizzare un dongle USB WLAB adapter. Nelle versioni precedenti questa ssenza portava ad occupare una porta USB per sempre.

È stata integrata una antenna Bluetooth 4.1. Per usi più comuni, tastiera e mouse bluetooth potranno sostituire quelle a USB, liberando così altre porte USB occupate.

Il processore SoC (system on chip) precedente, BCM2836 è stato sostituito dal BCM2837. Questo chip integra una CPU a 64 bit con frequenza 1.2 GHz, ARM Cortex-A53. Qui c'è stato un ulteriore salto di qualità e di potenza di calcolo, rispetto al modello precedente.

L'alimentazione è stata cambiata. Questa avviene tramite una porta MicroUSB, l'alimentatore da utilizzare deve essere cambiato con un alimentatore 5V/2.5A. [1]



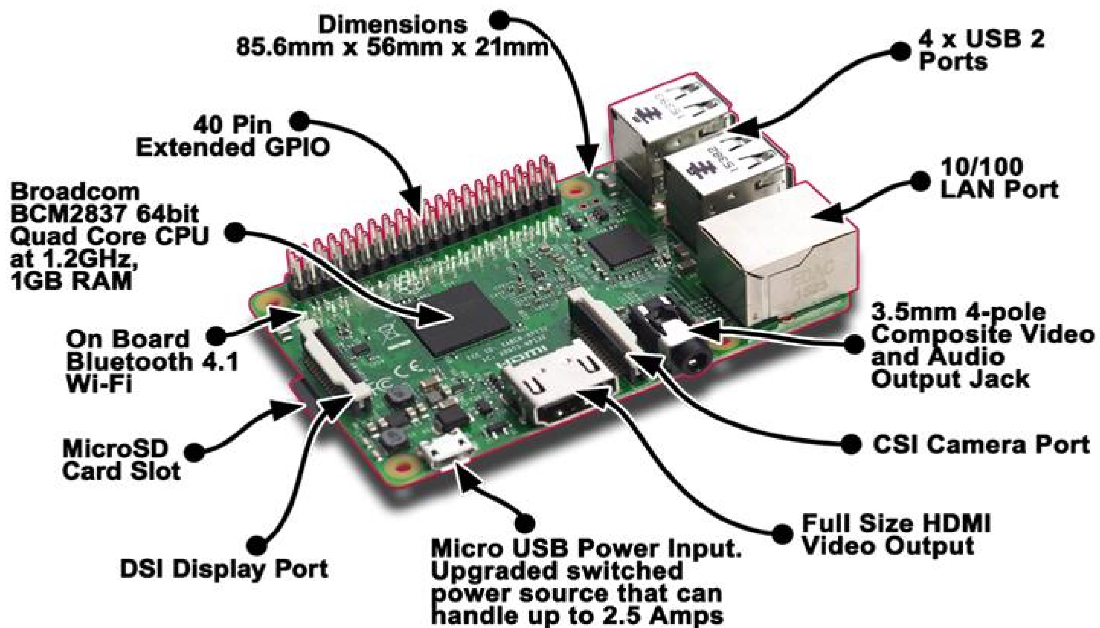
Evoluzione Raspberry Pi. [1]

La scheda grafica è rimasta identica a quella della Raspberry Pi 2 (GPU VideoCore IV 3D). Questa scelta è stata giustificata per motivi di retrocompatibilità con le versioni precedenti di Raspberry. Il poter riutilizzare vecchi codici sviluppati sulle schede precedenti è per la Raspberry Pi Foundation una caratteristica fondamentale.

La memoria è rimasta a 1GB di RAM (LPDDR2) ma la velocità è stata incrementata passando da 450 a 900 MHz

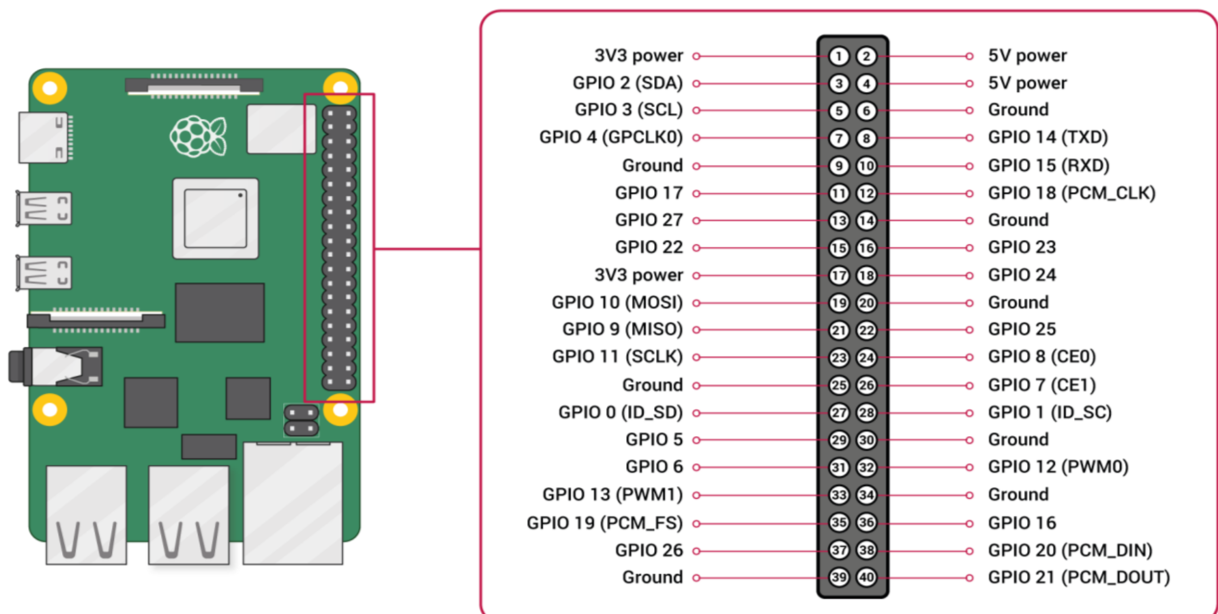
Le porte di I/O della scheda non sono state modificate. Sono le 4 porte USB 2.0, la porta Ethernet, lo slot per la microSD e l'uscita HDMI.

Non hanno subito variazioni neanche le dimensioni e il layout della scheda, questo per permettere la compatibilità di sistemi e accessori sviluppati ed ideati per le versioni precedenti di Raspberry.



Panoramica Raspberry Pi 3 Model B [1].

I 40 GPIO pin sono rimasti gli stessi e non sono stati spostati. Stesso discorso per la Camera Interface CSI e la Display Interface DSI. [3]



Pinout Raspberry Pi 3 Model B

Breve riepilogo delle caratteristiche:

- Broadcom BCM2837 64bit ARM Cortex-A53 Quad Core Processor SoC running @ 1.2GHz

- 1GB RAM
- 4x USB2.0 porte fino a 1.2A output
- Aumento a 40-pin GPIO Header
- Video/Audio Out via 4-pole 3.5mm connector, HDMI, CSI camera, or Raw LCD (DSI)
- Archiviazione: microSD
- 10/100 Ethernet (RJ45)
- BCM43143 WiFi integrato
- Bluetooth Low Energy (BLE) integrato
- Power Requirements: 5V @ 2.4A via microUSB
- Low-Level Peripherals:
  - 27 x GPIO
  - UART
  - I2C bus
  - SPI bus with two chip selects
  - +3.3V
  - +5V
- Ground
- Supports Raspbian, Windows 10 IoT Core, OpenELEC, OSMC, Pidora, Arch Linux, RISC OS.

3.RFM69HCW

3.1 Introduzione

RFM69HCW è un modulo ricetrasmettitore ideale per le odierne applicazioni RF in banda ISM ad alte prestazioni. È progettato per l'uso come ricetrasmettitore RF FSK e OOK ad alte prestazioni e basso costo per collegamenti RF bidirezionali half-duplex

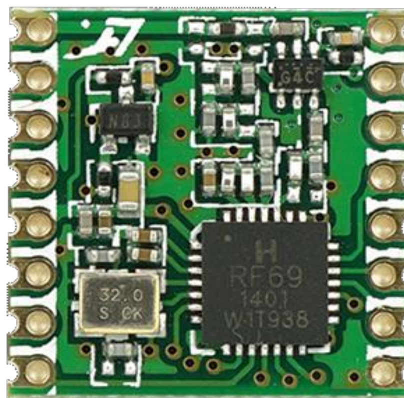
agili in frequenza robusti e dove sono richieste prestazioni RF stabili e costanti sull'intero intervallo operativo del dispositivo fino a 1,8 V.

RFM69HCW è un modulo ricetrasmettitore in grado di funzionare su un'ampia gamma di frequenze, comprese le bande di frequenza ISM (Industry Scientific and Medical) 315.433.868 e 915 MHz senza licenza.

Tutti i principali parametri di comunicazione RF sono programmabili e la maggior parte di essi può essere impostata dinamicamente. RFM69HCW offre il vantaggio esclusivo delle modalità di comunicazione a banda stretta e larga programmabili. RFM69HCW è ottimizzato per un basso consumo energetico offrendo al contempo un'elevata potenza di uscita RF e un funzionamento canalizzato.

Le funzionalità di sistema avanzate dell'RFM69HCW includono un FIFO TX / RX a 66 byte, un gestore di pacchetti automatico configurabile, modalità di ascolto (Listen Mode), sensore di temperatura e DIO configurabili che migliorano notevolmente la flessibilità del sistema allo stesso tempo riducendo significativamente i requisiti MCU.

RFM69HCW è conforme ai requisiti normativi ETSI e FCC.

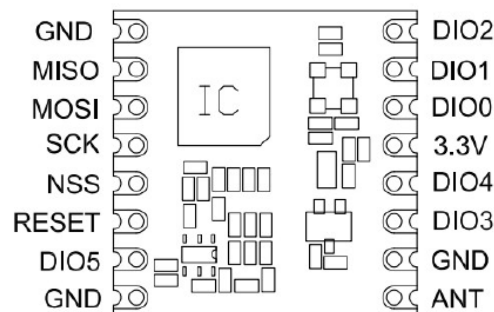


RFM69HCW

3.2 Caratteristiche

- Potenza di uscita: +20dBm - 100 mW
- Alta sensibilità: -120 dBm a 1.2 kbps
- Alta selettività: 16-tap FIR Channel Filter
- Basso consumo di corrente: Rx = 16 mA, 100nA ritenzione registri

- Pout programmabile: -18 a +20 dBm in 1dB steps
- Prestazioni RF costanti nel range di tensione del modulo
- FSK Bit rates fino a 300 kb/s
- Sintetizzatore completamente integrato con una risoluzione di 61 Hz
- Modulazioni FSK, GFSK, MSK, GMSK e OOK
- Bit Synchronizer integrato che esegue il Clock Recovery
- Riconoscimento della Sync Word in entrata
- 115 dB+ Dynamic Range RSSI
- Rilevamento RF automatico con AFC ultraveloce
- Packet engine con CRC-16, AES-128, 66-byte FIFO
- Sensore di temperatura integrato
- Dimensioni del modulo : 16x16mm



Piedinatura RFM69HCW

3.3 Applicazioni

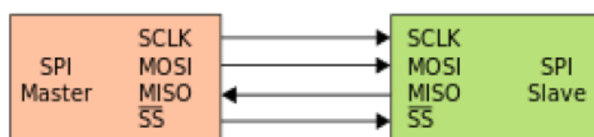
- Lettura automatica dei contatori
- Reti di sensori wireless
- Domotica
- Sistemi di allarme e sicurezza wireless
- Monitoraggio e controllo industriale

- Wireless M-BUS

4.SPI

4.1 Introduzione

Il Serial Peripheral Interface o SPI è un sistema di comunicazione tra un microcontrollore e altri circuiti integrati o tra più microcontrollori. È un bus standard di comunicazione ideato dalla Motorola.



Collegamento Master – Slave

La trasmissione avviene tra un dispositivo detto master e uno o più slave (dall'inglese padrone e schiavo). Il Master è il dispositivo che comanda il sistema; in generale si tratta di un microcontrollore. Esso ha la possibilità di inviare e ricevere dati e comandi e di iniziare una sessione di trasmissione. Dal master viene fornito anche il clock di sincronismo dello scambio di dati.

Lo Slave è un dispositivo periferico che può ricevere e inviare dati, ma non può inviare comandi (risponde solo a quelli dettati dal master), né iniziare una sessione di trasmissione. Il clock per la comunicazione è fornito dal Master e lo Slave non ha alcun controllo su questa linea. È possibile mettere in collegamento più Slave ad uno stesso Master, ma non sono previsti più Master sullo stesso circuito dati.

Il bus SPI si definisce:

- di tipo seriale
- sincrono per la presenza di un clock che coordina la trasmissione e ricezione dei singoli bit e determina la velocità di trasmissione
- full-duplex in quanto il "colloquio" può avvenire contemporaneamente in trasmissione e ricezione.

Per quanto riguarda la velocità di scambio dei dati (la frequenza del clock) non vi è un limite minimo ma vi è un limite massimo che va determinato dai datasheet dei singoli

dispositivi che si stanno utilizzando e dal loro numero in quanto ogni dispositivo collegato al bus introduce sulle linee di comunicazione una capacità parassita.

Il sistema di comunicazione di solito serve per lo scambio di dati tra dispositivi montati "sulla stessa scheda elettronica" (o comunque tra schede elettroniche vicine tra di loro) in quanto non prevede particolari accorgimenti hardware per trasferire informazioni tra dispositivi lontani connessi con cavi soggetti a disturbi.

Il sistema viene comunemente definito a quattro fili. Nonostante però i fili sono effettivamente cinque in quanto deve essere presente anche una tensione di riferimento (0 Vdc comunemente indicata con GND). [4] [5]

4.2 I quattro segnali

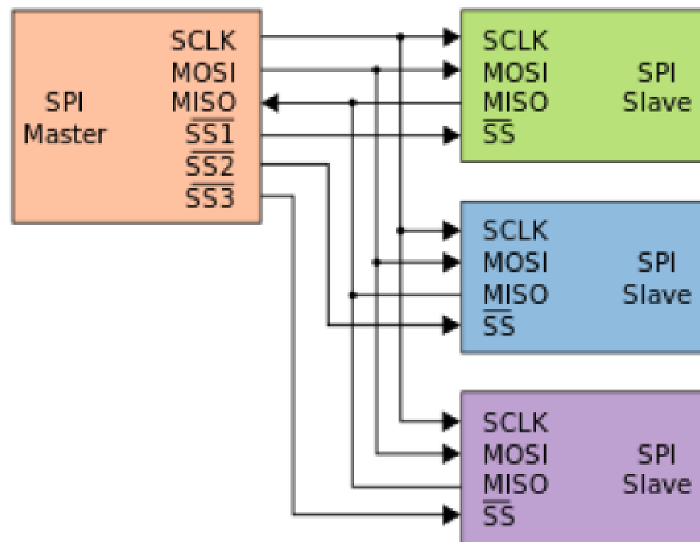
Dunque la trasmissione SPI si basa su 4 segnali (si riportano diversi nomi dei segnali in quanto variare a seconda del costruttore):

- SCLK - SCK: Serial CLocK (emesso dal master);
- SDI – MISO – SOMI – DI - SO: Serial Data Input, Master Input Slave Output (ingresso per il master ed uscita per lo slave);
- SDO – MOSI – SIMO – DO – SI: Serial Data Output, Master Output Slave Input (uscita dal master ed ingresso per lo slave);
- CS – SS – nCS – nSS – STE: Chip Select, Slave Select emesso dal master per scegliere con quale dispositivo slave si vuole comunicare (per comunicare con lo slave deve venire messo a livello logico basso questo bit). [4]

4.3 Collegamento dispositivi slave

Nel caso ci fossero più slave è possibile collegarli al master in 2 modi differenti:

1-Dispositivi slave controllati singolarmente



Slave collegati al master in parallelo

Il primo modo, di uso generale, è quello di fornire una distribuzione in parallelo dei segnali SDI e SDO. Il Master comanda tante linee di chip select quante sono le periferiche. In sostanza, si tratta di utilizzare pin di I/O del microcontrollore, comandandoli in modo da abilitare una sola periferica per volta.

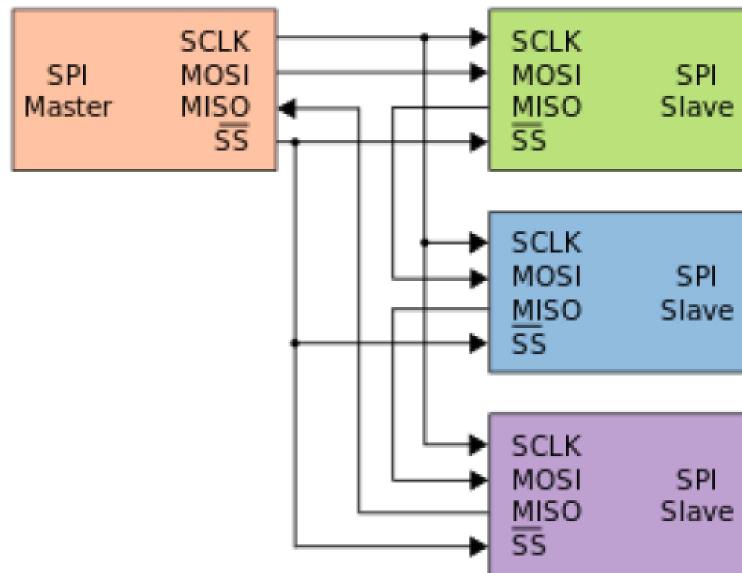
Questo metodo consente di collegare tante periferiche quanti sono i chip select disponibili, il che può risultare uno svantaggio qualora si abbia a che fare con un microcontrollore con un basso numero di pin. In genere, però, non si tratta di un grosso problema dato che è raro che un solo Master comandi un gran numero di Slave, vista anche la necessità di limitare a lunghezze ragionevoli i conduttori del bus per consentire i transfer rate elevati che offre la connessione SPI.

Il vantaggio principale di questa configurazione è che ogni periferica, essendo abilitata separatamente, potrà disporre di una gestione propria. Le periferiche non selezionate saranno disabilitate e indifferenti ai segnali sui pin, che si troveranno in 3-state, quindi con carichi ad alta impedenza che non influenzano il bus, permettendo al Master di configurarsi diversamente per ognuna di esse.

Problemi, però, possono insorgere in quanto il protocollo originale SPI di Motorola si aspetta che tutti i dispositivi Slave utilizzino uscite MISO/DOUT 3-state, ma alcune periferiche non hanno uscite 3-state, dato che prevedono una connessione a catena (vedi sotto). Poiché DOUT di ogni dispositivo è destinato a guidare il pin DIN di un altro, deve rimanere attivo. Se l'output DOUT è andato ad alta impedenza, l'input DIN

di un dispositivo di daisy chain successivo non riceverebbe il dato circolante, bloccando il loop. [4] [5]

2-Dispositivi slave connessi in catena (daisy chain):



Collegamento Master – Slave a catena

Con questa configurazione quelli che erano i vantaggi corrispondono agli svantaggi della connessione singola e viceversa. Nella connessione a catena il master ha un solo pin per chip select ma la velocità di aggiornamento dei singoli slave è minore. In questo caso la linea SS serve per indicare agli slave quando campionare il dato presente nel registro (il master manda i bit sulla linea MOSI, partendo dal bit più significativo da inviare all'ultimo slave. Una volta trasmessi tutti i bit destinati a tutti gli slave in questa sequenza, il master segnala agli slave che i dati in loro possesso nel registro di comunicazione sono effettivamente quelli destinati a loro).

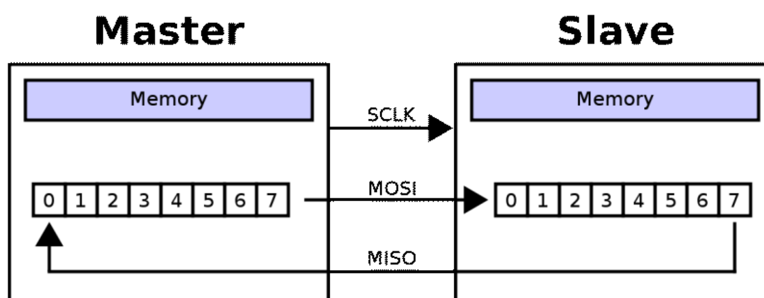
In questo tipo di connessione viene aggirata l'eventualità di un avere conflitto sulla linea MISO del Master, cosa che potrebbe invece accadere nella connessione di dispositivi Slave controllati singolarmente. Infatti qui una eventuale abilitazione di più slave creerebbe un conflitto sulla linea MISO che verrebbe utilizzata da più periferiche creando problemi nella comunicazione. [4] [5]

4.4 Come avviene la comunicazione

La trasmissione dei bit sul bus SPI si basa sul funzionamento dei registri a scorrimento (shift register). Ogni dispositivo usato possiede uno shift register interno i cui bit vengono emessi e, contemporaneamente, immessi, rispettivamente, tramite l'uscita MOSI e l'ingresso MISO. Spesso questo registro ha una dimensione di 8 bit.

Le operazioni di lettura e scrittura sullo shift register vengono effettuate in modo parallelo, durante la trasmissione invece i bit vengono trasmessi e ricevuti in modo seriale.

Ad ogni impulso di clock i dispositivi che stanno comunicando sulle linee del bus emettono un bit dal loro registro interno rimpiazzandolo con un bit emesso dall'altro dispositvo.



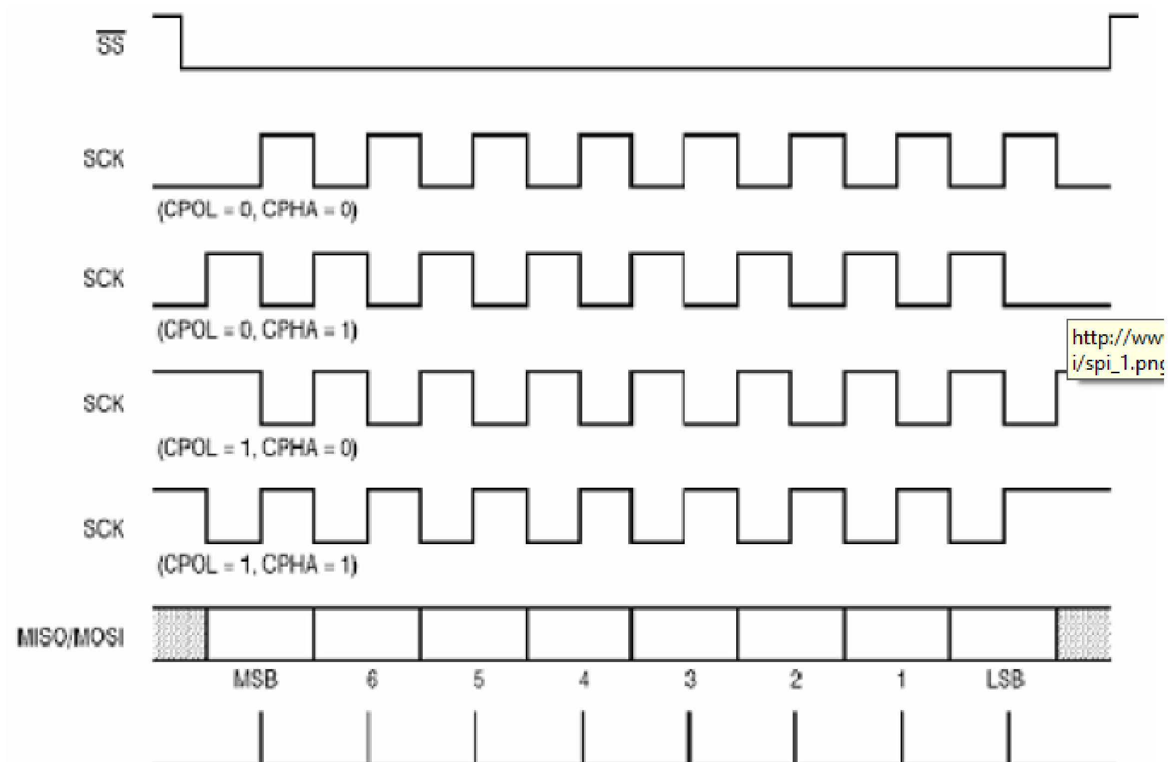
Comportamento Shift Register

La sincronizzazione è fatta sui fronti di clock di salita o di discesa dipendentemente dal valore di 2 parametri impostabili: CPOL e CPHA.

-CPOL indica lo stato di inattività del clock quando non avviene la trasmissione. Se CPOL=0, il clock nel suo stato di riposo è a livello logico basso; viceversase CPOL=1.

-CPHA regola la fase del clock, ovvero il fronte di clock in cui il ricevente campiona il segnale che ha in ingresso. Se CPOL=0 allora possiamo scegliere di campionare il dato sul fronte di salita del segnale di clock con CPHA=0, oppure sul fronte di discesa

con CPHA=1. L'inverso accade se CPOL è settato ad 1.



Questi parametri sono impostabili nel dispositivo master e vanno settati in base a come lavorano gli slave con cui vogliamo comunicare che invece supportano uno solo tra i quattro modi di comunicazione.

Spesso le modalità più sono: CPHA=CPOL=0 (caso del nostro modulo RFM69HCW) e CPHA=CPOL=1.

La comunicazione viene inizializzata sempre dal master che abilita lo slave settando a 0 il CS e successivamente impone il clock sulla linea dedicata. Con questa procedura ha inizio lo scambio dei bit tra i due registri. Alla fine di ogni parola trasmessa il contenuto del registro dello slave sarà passato al master e viceversa. [4] [5]

4.5 SPI su Raspberry

La famiglia di dispositivi Raspberry Pi è dotata di numerosi bus SPI.

Il core BCM2835 comune a tutti i dispositivi Raspberry Pi ha 3 controller SPI:

-SPI0, con due chip select, è disponibile in tutti i Pis (sebbene esista una mappatura alternativa utilizzabile solo su un Compute Module)

-SPI1, con tre chip select, è disponibile nelle versioni a 40 pin di Pis.

-SPI2, anch'esso con tre chip select, è utilizzabile solo su un Compute Module perché i pin non vengono portati sull'intestazione a 40 pin.

Pin/GPIO mappings:

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	19	GPIO10	SPI0_MOSI
MISO	21	GPIO09	SPI0_MISO
SCLK	23	GPIO11	SPI0_SCLK
CE0	24	GPIO08	SPI0_CE0_N
CE1	26	GPIO07	SPI0_CE1_N

Per abilitare l'SPI0 bisogna usare il comando raspi-config, andare su “Advanced options”, selezionare l'opzione “SPI” e premere “yes”; oppure assicurarsi che dtparam=spi=on sia presente nel file di configurazione /boot/config.txt e riavviare.

Se il driver SPI è stato caricato si può notare i device /spidev0.0 e /spidev0.1 nella directory /dev . [6]

5. Device driver

5.1 Introduzione

Uno dei tanti vantaggi dei sistemi operativi gratuiti, come Linux, è che il loro contenuto interno è aperto a tutti per la modifica e la visualizzazione. Il sistema operativo, una volta una zona oscura e misteriosa, il cui codice era limitato a un piccolo numero di programmatori, ora può essere facilmente esaminato, compreso e modificato da chiunque abbia le competenze necessarie. Il kernel di Linux rimane un corpo di codice grande e complesso, tuttavia, i programmatori avrebbero bisogno di un punto di ingresso dove possono avvicinarsi al codice senza essere sopraffatti dalla complessità. Spesso i device driver forniscono quel gateway.

I device driver assumono un ruolo speciale nel kernel di Linux. Sono distinte "scatole nere" che consentono ad un particolare pezzo di hardware di rispondere ad un'interfaccia di programmazione interna ben definita. Le attività degli utenti vengono eseguite tramite una serie di chiamate standardizzate indipendenti dal driver specifico. Mappare tali chiamate alle operazioni specifiche del dispositivo che agiscono

sull'hardware reale è quindi il ruolo del device driver. Questa interfaccia di programmazione fa in modo che i driver possono essere costruiti separatamente dal resto del kernel e "collegati" in fase di runtime quando necessario.

Come detto sopra una delle buone funzionalità di Linux è la capacità di estendere in runtime il set di funzioni offerte dal kernel. Ciò significa che è possibile aggiungere (allo stesso modo rimuovere) funzionalità al kernel mentre il sistema è in esecuzione.

Ogni pezzo di codice che può essere aggiunto al kernel in fase di runtime è chiamato modulo.

Il kernel di Linux offre supporto per diversi tipi (o classi) di moduli, inclusi i device driver.

Ogni modulo è costituito da un codice oggetto (non collegato ad un eseguibile completo) che può essere collegato dinamicamente al kernel in esecuzione dal comando *insmod* e può essere disconnesso dal comando *rmmod*.

Il modo con cui Linux guarda i device si distingue in tre tipi di dispositivi fondamentali. Ogni modulo di solito implementa uno di questi tipi, e quindi è classificabile come un char module, un block module o un network module. Questa divisione di moduli in diversi tipi o classi non è rigida; il programmatore può scegliere di costruire moduli enormi che implementano driver diversi in un solo pezzo di codice.

I buoni programmatori, tuttavia, solitamente creano un modulo diverso per ogni nuova funzionalità che implementano, perché la decomposizione è un elemento chiave della scalabilità e dell'estensione.

Le tre classi sono:

- Character devices (char device)

  Un character (char) device può essere aggiunto come un flusso di byte (come un file); un driver char è responsabile dell'attuazione di questo comportamento. Tale driver effettua solitamente almeno le chiamate di apertura (open), chiusura (close), lettura (read) e scrittura (write). La console di testo (/dev/console) e le porte seriali (/dev/ttyS0 e simili) sono esempi di dispositivi char. I dispositivi Char sono accessibili tramite i nodi dei filesystem, come

/dev/tty1 e /dev/lp0. L'unica differenza rilevante tra un char device e un file regolare è che puoi sempre spostarti avanti e indietro nel file normale, mentre la maggior parte dei dispositivi char sono solo canali di dati cui si può accedere solo in sequenza. Esistono comunque dispositivi char che sembrano aree di dati in cui ci si può muovere avanti e indietro in essi; per esempio, questo si applica solitamente ai frame grabber, in cui le applicazioni possono accedere all'intera immagine acquisita usando mmap o lseek.

- Block devices

  Analogamente ai dispositivi char, i block devices sono accessibili dai nodi dei filesystem nella directory */dev*. Un block device è un dispositivo (ad esempio un disco) che può ospitare un filesystem. Nella maggior parte dei sistemi Unix, un block device può gestire solo le operazioni di I/O che trasferiscono uno o più blocchi interi, di solito 512 (o una potenza di 2 più grande) byte di lunghezza. Linux, invece, consente all'applicazione di leggere e scrivere un block device come un dispositivo char: permette di trasferire qualsiasi numero di byte alla volta. Di conseguenza, i block device variano rispetto ai char decvice solo nel modo in cui i dati vengono gestiti internamente dal kernel e quindi nell'interfaccia del software del kernel */driver*. Come un char device, ogni block device è accessibile attraverso un nodo di filesystem e la differenza tra di essi è trasparente all'utente.

- Network interfaces

  Qualsiasi transazione di rete viene effettuata tramite un'interfaccia, ovvero un dispositivo in grado di scambiare dati con altri host. Di solito, un'interfaccia è un dispositivo hardware, ma potrebbe anche essere un dispositivo software puro. Un'interfaccia di rete è responsabile dell'invio e della ricezione di pacchetti di dati, guidati dal sottosistema di rete del kernel, senza sapere come vengono mappate le transazioni individuali ai pacchetti effettivi che vengono trasmessi. Un driver di rete non sa niente di singoli collegamenti; gestisce solo i pacchetti. Non essendo un dispositivo orientato al flusso, un'interfaccia di rete

non è facilmente mappata su un nodo nel filesystem. La comunicazione tra il kernel e un driver di periferica di rete è completamente diversa da quella utilizzata con i driver char e blocco. Invece di leggere e scrivere, il kernel chiama le funzioni relative alla trasmissione di pacchetti.

Esistono altri modi di classificare moduli di driver ortogonali ai tipi di periferica sopra descritti. In generale, alcuni tipi di driver funzionano con livelli aggiuntivi di funzioni di supporto del kernel per un determinato tipo di dispositivo. Ad esempio, si può parlare di moduli seriali universali (USB), moduli seriali, moduli SCSI e così via. Ogni dispositivo USB è guidato da un modulo USB che funziona con il sottosistema USB, ma il dispositivo stesso appare nel sistema come un dispositivo char (una porta seriale USB, ad esempio), un dispositivo di blocco (un lettore di schede di memoria USB) o un dispositivo di rete (un'interfaccia Ethernet USB). [7]

5.2 Costruire moduli

L'esempio più semplice di modulo e quello più usato dai vari libri e siti internet è "Hello World".

```
#include <linux/init.h>
#include <linux/module.h>
static int hello_init(void)
{
printk(KERN_ALERT "Hello, world\n");
return 0;
}
static void hello_exit(void)
{
printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL v2");
```

Questo modulo definisce due funzioni:

- hello_init(void) da invocare quando il modulo viene caricato nel kernel;
- hello_exit(void) da invocare quando il modulo viene rimosso.

Le funzioni module_init(hello_init) e module_exit(hello_exit) utilizzano macro speciali del kernel per indicare il ruolo delle funzioni prese come argomento.

Un'altra macro speciale (MODULE_LICENSE) viene utilizzata per informare il kernel che questo modulo ha una licenza gratuita. Altre definizioni descrittive che possono essere contenute all'interno di un modulo includono MODULE_AUTHOR (indica chi ha scritto il modulo), MODULE_DESCRIPTION (una descrizione generale del modulo), MODULE_VERSION (numero che identifica la versione del modulo), MODULE_ALIAS (un altro nome per il quale questo modulo può essere conosciuto) e MODULE_DEVICE_TABLE (per indicare all'utente quali dispositivi il modulo supporta). Le diverse dichiarazioni MODULE_ possono essere visualizzate ovunque all'interno del file di origine al di fuori di una funzione.

Tuttavia, una convenzione relativamente recente nel codice del kernel è quella di mettere queste dichiarazioni alla fine del file.

La funzione printk è definita nel kernel Linux e resa disponibile ai moduli; si comporta allo stesso modo della funzione di libreria standard C printf.

Il kernel ha bisogno di una propria funzione di stampa perché funziona da solo, senza l'aiuto della libreria C. Il modulo può chiamare printk perché, dopo che è stato caricato da insmod, è collegato al kernel e può accedere ai simboli pubblici del kernel (funzioni e variabili). La stringa KERN_ALERT indica la priorità del messaggio.

Si è specificato un'alta priorità in questo modulo perché un messaggio con la priorità predefinita potrebbe non apparire in nessuna parte utile, a seconda della versione e della configurazione del kernel in esecuzione. In totale ci sono otto livelli di priorità:

```
#define KERN_EMERG   "<0>" /* system is unusable               */
#define KERN_ALERT   "<1>" /* action must be taken immediately */
#define KERN_CRIT    "<2>" /* critical conditions              */
#define KERN_ERR     "<3>" /* error conditions                 */
#define KERN_WARNING "<4>" /* warning conditions               */
#define KERN_NOTICE  "<5>" /* normal but significant condition */
#define KERN_INFO    "<6>" /* informational                    */
#define KERN_DEBUG   "<7>" /* debug-level messages             */
```

Per visualizzare tutti i messaggi provenienti dal kernel si utilizza il comando *dmesg*.

A questo punto ci sono alcuni prerequisiti che bisogna controllare prima di poter costruire i moduli del kernel. Uno di questi, per esempio, è assicurarsi di avere versioni sufficientemente aggiornate del compilatore, delle utility del modulo e di altri strumenti necessari.

Una volta impostato tutto si ha la necessità di creare un Makefile per il modulo: come tutti i programmatori sanno, creare un programma eseguibile da più file sorgenti consiste nel compilare separatamente i file sorgenti uno ad uno e poi unirli nel file eseguibile finale. Ma quando i file sorgente sono molti si procede creando un file contente il compilatore da usare, in che modo usarlo, quali file compilare, come creare il file eseguibile (programma o libreria che esso sia) e altre cose sempre utili, come la directory dove trovare le librerie necessarie ed i file di include. Questo è il contenuto del file Makefile.

Una volta creato il Makefile basterà utilizzare il comando *make* per avviare la compilazione che genererà alcuni file tra cui uno con estensione *.ko* , e questo sarà il modulo che verrà inserito nel kernel.

Quindi il passo successivo è l'aggiunta del modulo; come già sottolineato, *insmod* fa tutto il lavoro. A differenza del linker, tuttavia, il kernel non modifica il file del disco del modulo, ma piuttosto una copia in memoria. insmod accetta una serie di opzioni di riga di comando (per dettagli, vedere il manuale) e può assegnare i valori ai parametri nel modulo prima di collegarlo al kernel corrente. Quindi, se un modulo è stato progettato correttamente, può essere configurato in fase di inserimento. Esiste un altro programma per caricare moduli per cui vale una nota rapida: *modprobe*. Differisce da *insmod* in quanto esaminerà il modulo da caricare per vedere se contiene i riferimenti a tutti i simboli che non sono attualmente definiti nel kernel. Se si trovano tali riferimenti, *modprobe* cerca altri moduli nel percorso di ricerca del modulo corrente che definiscono i relativi simboli. Quando *modprobe* trova i moduli (che sono necessari per il modulo da caricare), carica anch'essi nel kernel. Come detto prima, i moduli possono essere rimossi dal kernel con il comando *rmmod*. Si noti che la rimozione del modulo non riesce se il kernel ritiene che il modulo sia ancora in uso (ad esempio, un programma dispone ancora di un file aperto per un dispositivo esportato dai moduli) oppure se il kernel è stato configurato per impedire la rimozione del modulo.

Tuttavia è possibile configurare il kernel per consentire la rimozione "forzata" dei moduli, anche quando sembrano occupati.

Bisogna esaminare alcune altre cose che devono essere visualizzate nel file sorgente del modulo. Il kernel è un ambiente unico e impone le proprie esigenze sul codice che si interfaccia con esso. La maggior parte del codice del kernel inizia con un numero abbastanza grande di file di intestazione per ottenere definizioni di funzioni, tipi di dati e variabili. Ci sono alcuni di essi che sono specifici per i moduli e devono essere visualizzati in ogni modulo caricabile.

Quasi tutti i codici dei moduli hanno il seguente:

```
#include <linux/init.h>
#include <linux/module.h>
```

Come già accennato, la funzione di inizializzazione del modulo registra qualsiasi facility offerta dal modulo. Per facility, intendiamo una nuova funzionalità, sia esso un driver intero o una nuova astrazione software, accessibile da un'applicazione. La definizione effettiva della funzione di inizializzazione è sempre:

```
static int __init initialization_function(void)
{
/* Codice di inizializzazione */
}
module_init(initialization_function);
```

Le funzioni di inizializzazione devono essere dichiarate statiche, in quanto non devono essere visibili al di fuori del file specifico; tuttavia non esiste una regola duro in quanto non viene esportata alcuna funzione al resto del kernel a meno che non sia richiesta esplicitamente.

Il token `__init` nella definizione è un suggerimento al kernel che la funzione viene utilizzata solo al momento dell'inizializzazione. Il loader del modulo cede la funzione di inizializzazione dopo il caricamento del modulo, rendendo la sua memoria disponibile per altri usi. Esiste un tag simile (`__initdata`) per i dati utilizzati solo durante l'inizializzazione. L'uso di `__init` e `__initdata` è opzionale, ma ne vale la pena.

Basta essere sicuri di non utilizzarli per qualsiasi funzione (o struttura dati) che verrà utilizzata dopo l'esecuzione dell'inizializzazione. Si potrebbe anche incontrare `__devinit` e `__devinitdata` nel kernel; questi si traducono in `__init` e `__initdata` solo se il kernel non è stato configurato per i dispositivi hotpluggable.

L'utilizzo di `module_init` è obbligatorio. Questa macro aggiunge una sezione speciale al codice oggetto del modulo che indica dove è possibile trovare la funzione di inizializzazione del modulo. Senza questa definizione, la funzione di inizializzazione non viene mai chiamata.

Ogni modulo completo richiede anche una funzione di pulizia, che annulla le registrazioni e restituisce tutte le risorse al sistema prima di rimuovere il modulo. Questa funzione è definita come:

```
static void __exit cleanup_function(void)
{
/* Codice di pulizia */
}
module_exit(cleanup_function);
```

La funzione di cleanup non ha valore da restituire, quindi viene dichiarato `void`. Il modificatore `__exit` contrassegna il codice come quello per il solo scaricamento del modulo (dicendo al compilatore di collocarlo in una sezione ELF speciale). Se il modulo è stato costruito direttamente nel kernel o se il tuo kernel è configurato per impedire lo scarico dei moduli, le funzioni contrassegnate con `__exit` vengono semplicemente scartate.

Per questa ragione, una funzione contrassegnata con `__exit` può essere chiamata solo quando il modulo viene scaricato o al momento di arresto del sistema; qualsiasi altro utilizzo è un errore.

La dichiarazione `module_exit` è necessaria per consentire al kernel di trovare la funzione di pulizia. Se il modulo non definisce una funzione di pulizia, il kernel non permette di essere scaricato. [7]

5.3 Char Drivers

I char devices sono accessibili attraverso dei nomi nel filesystem. Questi nomi sono chiamati file speciali o device file o semplicemente nodi dell'albero del filesystem; sono localizzati convenzionalmente nella directory */dev*. I file speciali per i driver char sono identificati da una "c" nella prima colonna dell'output di *ls -l*. I block devices appaiono in */dev*, ma sono identificati da una "b". Se si utilizza il comando *ls -l*, verranno visualizzati due numeri (separati da una virgola) nelle voci del device file prima della data dell'ultima modifica.

Questi numeri sono: il major e minor number del device per il particolare dispositivo.

Tradizionalmente, il major number identifica il driver associato al dispositivo mentre il minor number viene utilizzato dal kernel per determinare esattamente quale dispositivo viene indicato.

All'interno del kernel, il tipo dev_t (definito in <linux/types.h>) viene utilizzato per contenere i numeri dei dispositivi. Per rivelare i major e minor numbers di un dispositivo si fa uso delle macro:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Se, invece, si hanno a disposizione i due numeri e si vuole trasformarli in un dispositivo dev_t, basta utilizzare:

```
MKDEV(int major, int minor);
```

Una delle prime cose che il driver deve fare quando si crea un dispositivo char è quello di ottenere uno o più device numbers da utilizzare. La funzione necessaria per questa attività è register_chrdev_region, dichiarata in <linux/fs.h> :

```
int register_chrdev_region(dev_t first, unsigned int count,  char *name);
```

Qui first è il device number iniziale dell'intervallo che si desidera assegnare. Il campo riservato al minor number è spesso 0.

count è il numero totale di device numbers contigui che si sta richiedendo. Si noti che, se il count è grande, l'intervallo da richiedere potrebbe passare al major number successivo; ma tutto funzionerà correttamente finché la gamma di numeri richiesta è disponibile.

Infine, name è il nome del dispositivo che dovrebbe essere associato a questo intervallo numerico; apparirà in */proc/devices* e *sysfs*.

Come per la maggior parte delle funzioni del kernel, il valore di ritorno da register_chrdev_region sarà 0 se l'allocazione è stata eseguita correttamente. In caso di errore verrà restituito un codice di errore negativo e non sarà possibile accedere alla regione richiesta.

`register_chrdev_region` funziona bene se si conosce in anticipo esattamente quali numeri di dispositivo si desidera. Spesso, tuttavia, non si sa a priori quali numeri andrà ad utilizzare il dispositivo; esiste un'altra funzione per fornire device numbers allocati dinamicamente:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,  unsigned int count, char *name);
```

Con questa funzione, `dev` è un parametro solo di uscita che, al termine del completamento, tiene il primo numero nell'intervallo assegnato.

`firstminor` dovrebbe essere il primo numero minore richiesto da utilizzare; è di solito 0. I parametri del conteggio e del nome funzionano come quelli passati a `register_chrdev_region`.

Indipendentemente dal modo in cui si assegnano i device numbers, dovrebbero essere liberati quando non sono più in uso. I numeri dei dispositivi vengono liberati con:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Il luogo (nel codice) abituale dove chiamare `unregister_chrdev_region` sarebbe nella funzione di pulizia del modulo.

Come si può immaginare, la registrazione del numero di dispositivo è solo il primo di molte attività che il codice driver deve eseguire.

La maggior parte delle operazioni di driver fondamentali coinvolge tre importanti strutture di dati del kernel, chiamate *file_operations*, *file* e *inode*.

1-File operations:

finora si è riservato alcuni device numbers per il dispositivo, ma quei numeri non sono ancora connessi alle operazioni del driver.

La struttura `file_operations` corrisponde a come un char driver imposta la connessione. La struttura, definita in `<linux/fs.h>`, è una raccolta di puntatori di funzione.

```
struct file_operations {
```

```
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t,
            loff_t *);
ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                unsigned long, loff_t);
ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                unsigned long, loff_t);
ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
int (*iterate) (struct file *, struct dir_context *);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*mremap)(struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, loff_t, loff_t, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
…
};
```

Ogni file aperto (rappresentato internamente da una struttura file) è associato al proprio set di funzioni (includendo un campo chiamato `f_op` che punta a una struttura `file_operations`). Le operazioni sono prevalentemente incaricate di implementare le chiamate di sistema e sono quindi chiamate `open`, `read` e così via. Possiamo considerare il file come un "oggetto" e le funzioni che operano su di esso i suoi "metodi", utilizzando la terminologia di programmazione object-oriented per indicare le azioni dichiarate da un oggetto per agire su sé stesso. Questo è il primo segno della programmazione orientata agli oggetti che è presente nel kernel Linux.

Convenzionale, una struttura `file_operations` o un puntatore ad una di esse viene chiamata `fops`.

Ogni campo della struttura deve indicare la funzione nel driver che implementa un'operazione specifica o essere lasciata `NULL` per operazioni non supportate.

Alcuni parametri includono la stringa `__user`. Questa annotazione è una forma di documentazione, rilevando che un puntatore è un indirizzo utente-spazio che non può essere direttamente dereferenziato. Per la compilazione normale, `__user` non ha alcun

effetto, ma può essere utilizzato da un software di controllo esterno per trovare un uso improprio degli indirizzi dello spazio utente.

Le file operations che si utilizzano più frequentemente sono quelle di apertura e chiusura del file, lettura dal file e scrittura sul file:

```
int (*open) (struct inode *, struct file *);
```

Anche se questa è sempre la prima operazione eseguita sul device file, il driver non è tenuto a dichiarare un metodo corrispondente. Se questa voce è `NULL`, l'apertura del dispositivo riesce sempre, ma il driver non viene notificato.

Nella `open` è previsto per un driver eseguire qualsiasi inizializzazione in preparazione per operazioni successive. Nella maggior parte dei driver, aprire dovrebbe eseguire le seguenti attività: verifica degli errori specifici del dispositivo (ad esempio problemi di hardware non pronto o simili); inizializzare l'apparecchio se viene aperto per la prima volta; aggiornare il puntatore `f_op`, se necessario; assegnare e riempire qualsiasi struttura dati da mettere in `filp->private_data`.

```
int (*release) (struct inode *, struct file *);
```

Questa operazione viene richiamata quando la struttura file viene rilasciata. Come l'apertura, il rilascio può essere `NULL`.

Il ruolo della release è il contrario di quello dell'apertura.

Talvolta si trova che l'implementazione della funzione viene chiamata `device_close` invece di `device_release`. In entrambi i casi, il metodo del dispositivo dovrebbe eseguire le seguenti attività: deallocare tutto ciò che si ha in `filp->private_data` e spegnere il dispositivo.

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Utilizzata per recuperare i dati dal dispositivo. Un puntatore nullo in questa posizione provoca la mancata chiamata di lettura del sistema con `-EINVAL` ("Argomento non valido").

Un valore di ritorno non negativo rappresenta il numero di byte letto correttamente (il valore restituito è un tipo "signed size", di solito il tipo intero nativo per la piattaforma di destinazione).

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

Invia i dati al dispositivo. Se il puntatore è nullo, `-EINVAL` viene restituito al programma che provoca la chiamata di scrittura.

Il valore restituito, se non negativo, rappresenta il numero di byte scritti correttamente.

In un modulo per effettuare la dichiarazione si utilizza una sintassi standardizzata di inizializzazione.

Questa sintassi è preferita perché rende il codice più compatto e leggibile.

Quindi una dichiarazione con delle file operations di base è a seguente:

```
struct file_operations example = {
	.owner = THIS_MODULE,
	.read = ex_read,
	.write = ex_write,
	.open = ex_open,
	.release = ex_release,
};
```

2- File structure:

definita in `<linux/fs.h>`, è la seconda struttura dati più importante utilizzata nei device driver. Si noti che un file non ha nulla a che fare con i puntatori `FILE` di programmi spazio utente. Un FILE è definito nella libreria C e non viene mai visualizzato nel codice del kernel. Una file structure, d'altra parte, è una struttura del kernel che non appare mai nei programmi utente.

La file structure rappresenta un file aperto. (Non è specifico per i device driver, ogni file aperto del sistema ha una struttura file associata nello spazio del kernel.)

Viene creata dal kernel all'apertura e viene passata a qualsiasi funzione che opera sul file fino all'ultima chiusura. Dopo che tutte le istanze del file sono chiuse, il kernel

rilascia la struttura. Nelle origini del kernel, un puntatore alla file structure viene chiamato in genere `file` o `filp`.

Quindi si conviene che `file` si riferisce alla struttura e `filp` ad un puntatore alla struttura.

Come anticipato in precedenza ogni file aperto è associato al proprio set di funzioni tramite la dichiarazione:

```
struct file_operations *f_op;
```

Altre due strutture di tipo file molto importanti sono:

```
void *private_data;
```

La chiamata di sistema `open` imposta questo puntatore su `NULL` prima di chiamare il metodo di apertura del driver. Si è liberi di fare uso del campo o di ignorarlo; è possibile utilizzarlo come puntatore a dei dati allocati, ma è necessario ricordarsi di liberare quella memoria nel metodo di rilascio prima che la `struct file` venga distrutta dal kernel.

`private_data` è una risorsa utile per preservare le informazioni di stato nelle chiamate di sistema e viene utilizzata dalla maggior parte dei moduli di esempio.

```
loff_t f_pos;
```

La posizione attuale di lettura o scrittura.

`loff_t` è un valore a 64 bit su tutte le piattaforme (`long long` nella terminologia gcc). Il driver può leggere questo valore se deve conoscere la posizione corrente nel file, ma non dovrebbe normalmente cambiarla; leggere e scrivere dovrebbero aggiornare una posizione utilizzando il puntatore che ricevono come ultimo argomento invece di agire direttamente su `filp-> f_pos`.

L'unica eccezione a questa regola è nel metodo `lseek`, il cui scopo è cambiare la posizione del file.

3-Inode structure:

la struttura inode viene utilizzata dal kernel internamente per rappresentare i file. È diversa dalla `file structure` che rappresenta la descrizione di un file aperto.

Ci possono essere numerose file structure che rappresentano più descrizioni di un singolo file, ma tutte si riferiscono ad una singola struttura inode.

La struttura inode contiene una grande quantità di informazioni sul file. Come regola generale, solo due campi di questa struttura sono di interesse per la scrittura del codice del driver:

```
dev_t i_rdev;
```

Per le strutture inode che rappresentano i device file, questo campo contiene il device number effettivo.

```
struct cdev *i_cdev;
```

`struct cdev` è la struttura interna del kernel che rappresenta i dispositivi char; questo campo contiene un puntatore a quella struttura quando l'inode si riferisce a un device file char.

Come modo per incoraggiare una programmazione più versatile, gli sviluppatori del kernel hanno aggiunto due macro che possono essere utilizzate per ottenere il major number e il minor number da una struttura inode:

```
unsigned int imajor(struct inode *inode);
unsigned int iminor(struct inode *inode);
```

Come già annunciato, il kernel utilizza strutture di tipo `struct cdev` per rappresentare internamente i char device.

Prima che il kernel invochi le operazioni del dispositivo, bisogna assegnare e registrare una o più di queste strutture.

A tal fine, il codice dovrebbe includere `<linux/cdev.h>`, in cui sono definite la struttura e le relative funzioni. Ci sono due modi per assegnare e inizializzare una di queste strutture. Se si desidera ottenere una `struct cdev` autonoma in fase di esecuzione, è possibile eseguirla con:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Le probabilità sono, tuttavia, che si desidera incorporare la `struct cdev` all'interno di una struttura specifica del proprio dispositivo. In questo caso, è necessario inizializzare la struttura già allocata con:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

In entrambi i casi, c'è un campo di `struct cdev` che è necessario inizializzare. Come la struttura `file_operations`, ha un campo proprietario che dovrebbe essere impostato su `THIS_MODULE`.

Una volta che la `struct cdev` viene impostata, il passo finale è quello di informare il kernel con una chiamata a:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Qui `dev` è la `struct cdev`, `num` è il primo device number a cui questo dispositivo risponde e `count` è il numero di periferiche che dovrebbero essere associati al dispositivo.

Ci sono un paio di cose importanti da tenere a mente quando si utilizza `cdev_add`.

La prima è che questa chiamata può fallire. Se restituisce un codice di errore negativo, il dispositivo non è stato aggiunto al sistema.

Quasi sempre riesce, e questo porta l'altro punto: non appena il `cdev_add` ritorna, il tuo dispositivo è "vivo" e le sue operazioni possono essere chiamate dal kernel. Non si deve chiamare `cdev_add` finché il driver non è completamente pronto per gestire le operazioni sul dispositivo. [7]

Per rimuovere un dispositivo char dal sistema, chiamare:

```
void cdev_del(struct cdev *dev);
```

6. RFM69HCW Driver

6.1 Dichiarazioni preliminari

Come generalmente si trova nei codici, all'inizio si vanno a dichiarare costanti e variabili globali e le funzioni che saranno poi definite in seguito. Oltre a questo si va ad indicare al linker e al compilatore quali sono i file e le librerie che contengono macro, funzioni e costanti da includere per l'esecuzione del programma.

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/cdev.h>
#include <linux/interrupt.h>
#include <linux/gpio.h>
#include <linux/spi/spi.h>
```

Come preannunciato per includere i file si utilizza la direttiva `#include` e il nome del file da associare al codice principale va indicato come percorso (es. il file `module.h` si trova nella directory linux).

Si procede con la definizione di alcune costanti:

```
#define RFM_CLS_NAME      "radio69-ism"
#define RFM_DRV_NAME      "RFM69HCW"
#define RFM_DEV_NAME      "RFM69"
#define RFM_MAX_MINORS     32
```

Grazie alla direttiva `#define` associamo un valore o una stringa ad un nome. Questo valore non potrà essere modificato nel resto del programma (es. a `RFM_MAX_MINORS` viene assegnato il valore `32`).

Successivamente vengono dichiarate ed inizializzate tre variabili `rfmdev_minor`, `rfmdev_major` e `rfm_minors`.

```
static int rfmdev_minor = 0;
static int rfmdev_major = 0;
static unsigned long rfm_minors;
```

Le prime due andranno a contenere, per la registrazione del driver, rispettivamente il minor number e il major number mentre la terza conterrà il minor number per la registrazione dei dispositivi RFM che si vogliono connettere.

I char devices sono accessibili attraverso dei nomi nel filesystem. Questi nomi sono chiamati file speciali o device file o semplicemente nodi dell'albero del filesystem; sono localizzati convenzionalmente nella directory /dev. I file speciali per i driver char sono identificati da una "c" nella prima colonna dell'output di ls -l. I block devices appaiono in /dev, ma sono identificati da un "b". Se si utilizza il comando ls -l, verranno visualizzati due numeri (separati da una virgola) nelle voci del device file prima della data dell'ultima modifica. Questi numeri sono il major e minor number del device per il particolare dispositivo. Tradizionalmente, il major number identifica il driver associato al dispositivo mentre il minor number viene utilizzato dal kernel per determinare esattamente quale dispositivo viene indicato.

Contestualmente a quanto detto si invoca la macro `module_param` che consente di passare come parametro `rfmdev_major` all'inserimento del modulo. [7] [8]

```
module_param(rfmdev_major, int, S_IRUGO);
```

La macro `module_param` () accetta 3 argomenti: il nome della variabile, il suo tipo e le autorizzazioni per il file corrispondente in sysfs.

Si va a dichiarare poi la classe RFM. Una device class descrive un tipo di dispositivo, ad esempio un dispositivo audio o di rete. Ogni device class definisce una semantica e un'interfaccia di programmazione a cui i dispositivi di quella classe aderiscono.

I device driver sono l'implementazione di quella interfaccia di programmazione per un particolare dispositivo su un particolare bus.

```
static struct class *rfm_class;
```

Viene poi dichiarata una variabile d'appoggio utilizzata in seguito per eseguire la lettura.

```
static int misura_disponibile;
```

Passiamo poi alla definizione della struttura `rfm_data`. Solitamente quando si costruisce un modulo è opportuno ed indispensabile salvare e mantenere durante

l'esecuzione del programma determinati dati che servono alle funzioni che manipolano tali dati.

Quindi si crea una struttura che contiene delle informazioni generiche per il driver in questione (es. `irq` corrisponde al numero dell'interrupt che si andrà ad utilizzare).

```
struct rfm_data {
	int irq;
	dev_t devt;
	spinlock_t lock;
	struct spi_device* spi;
	wait_queue_head_t wait_read;
	wait_queue_head_t wait_write;
	int open;
	int counter;
	struct device *dev;
	struct cdev cdev;
};
```

In seguito si trovano definizioni di costanti che indicano, in valori numerici, la causa dell'interrupt. Si andrà ad esaminare questo aspetto più tardi, per ora ci si limita all'esposizione di tali costanti.

```
#define ISR_MODEREADY            0x80
#define ISR_RXREADY              0x4000
#define ISR_TXREADY              0x2000
#define ISR_PLLLOCK              0x1000
#define ISR_RSSI                 0x0800
#define ISR_TIMEOUT              0x0400
#define ISR_AUTOMODE             0x0200
#define ISR_SYNCADDRESSMATCH     0x0100
#define ISR_FIFOFULL             0x0080
#define ISR_FIFONOTEMPTY         0x40
#define ISR_FIFOLEVEL            0x0020
#define ISR_FIFOOVERRUN          0x0010
#define ISR_PACKETSENT           0x08
#define ISR_PAYLOADREADY      0x0004
#define ISR_CRCOK             0x0002
```

Per rendere più leggibile il codice si associano dei valori a delle costanti auto esplicative che appunto vanno a descrivere gli stati di una macchina a stati che verrà utilizzata nel codice.

```
#define RFM69HCW_STATE_UNINITIALIZED     0
#define RFM69HCW_STATE_RESETTING         1
#define RFM69HCW_STATE_INITIALIZING      2
#define RFM69HCW_STATE_DISCONNECTED      3
#define RFM69HCW_STATE_RECEIVE           4
```

```
#define RFM69HCW_STATE_TRANSMIT          5
#define RFM69HCW_STATE_RECEIVING         6
#define RFM69HCW_STATE_ABORTED           7
```

Si definiscono poi alcune costanti che riguardano i pacchetti ed i buffer di ricetrasmissione:

```
#define RFM69HCW_PACKET_MAX_LENGTH    255
#define RFM69HCW_TX_FIFO_AEM    33
#define RFM69HCW_RX_FIFO_AFULL   15
#define RFM69HCW_TX_FIFO_LEN   66
#define RFM69HCW_RX_FIFO_LEN   66
```

`RFM69_PACKET_MAX_LENGTH` corrisponde alla massima lunghezza di un pachetto che il modulo RFM69HCW supporta, `RFM69_TX_FIFO_AEM` e `RFM69_RX_FIFO_AFULL` indicano le soglie dei buffer di trasmissione quasi vuoto e di ricezione quasi pieno. Le ultime due evidenziano la lunghezza dei due buffer della radio.

Si costruiscono successivamente le strutture dei buffer di ricetrasmissione nel programma:

```
typedef struct {
      size_t length;
      size_t offset;
      uint8_t data[255];
} buffer_t;

/* Buffer declaration */
static buffer_t rfm69hcw_rx_buffer;
static buffer_t rfm69hcw_tx_buffer;
```

I campi della struttura sono: `length` indica la lunghezza, `offset` è come un segnalibro che suggerisce il punto dove andare a scrivere o leggere i dati dai rispettivi vettori `data[255]`.

Si passa quindi all'esposizione di un vettore, strutturato visivamente in due colonne.

Contiene l'indirizzo dei registri del modulo radio, nella parte sinistra, e i valori che devono essere inseriti nei registri per configurare lo stesso modulo, nella parte destra.

```
static const uint8_t RFM69HCWP_config[] = {
      0x02 + 0x80, 0x00,
```

```
        0x03 + 0x80, 0x3E,
        0x04 + 0x80, 0x80,

        0x06 + 0x80, 0x52,

        0x07 + 0x80, 0x6C,
        0x08 + 0x80, 0x80,
        0x09 + 0x80, 0x12,

        0x18 + 0x80, 0x88,

        0x19 + 0x80, 0x55,

        0x25 + 0x80, 0x01,
        0x26 + 0x80, 0xB7,

        0x29 + 0x80, 0xE4,

        0x2D + 0x80, 0x08,

        0x2E + 0x80, 0x98,
        0x2F + 0x80, 0x2D,
        0x30 + 0x80, 0xD4,
        0x31 + 0x80, 0xF5,
        0x32 + 0x80, 0xCB,

        0x37 + 0x80, 0x98,

        0x38 + 0x80, 0xFF,

        0x3C + 0x80, 0x8F,

        0x6F + 0x80, 0x30,

        0x01 + 0x80, 0x04,
        0x00, 0x00
};
```

Ovviamente la configurazione è inviata tramite SPI alla radio. Dato che i valori devono essere comunicati all'RFM69HCW si aggiunge un `'+ 0x80'` all'indirizzo del registro per informare il modem che si tratta di un'operazione in scrittura.

- All'indirizzo 0x02 c'è il registro "RegDataModul" scrivendoci 0x00 andiamo a impostare il modulo in packet mode, modulazione FSK senza alcun data shaping;
- Agli indirizzi 0x03 e 0x04 ci sono i registri BitRate(15:8) e BitRate(7:0). Andando a scrivere i valori 0x3E e 0x80 impostiamo il BitRate a 2 kb/s.

- All’indirizzo 0x06 c’è il registro “RegFdevLsb”, registro dedicato al settaggio della deviazione di frequenza del modulo radio. Nella modulazione di frequenza la deviazione di frequenza indica la massima differenza tra la frequenza della portante modulata e la frequenza della portante non modulata. Andando a scrivere in questo registro il valore 0x52 e lasciando il registro 0x05 inalterato (RegFdevMsb 0x00 di defeault), la deviazione di frequenza sarà di 5kHz. Questo valore è stato trovato seguendo la formula: Fdev=Fstep*Fdev(15,0). Fstep=Fxosc / (2^{19}) . (Fxosc= 32MHz)

- Agli indirizzi 0x07 0x08 e 0x09 ci sono i registri Frf(23:16) Frf(15:8) e Frf(7:0). Questi registri sono rispettivamente: Msb, middle byte e LSB della frequenza portante. Con i valori scritti dal vettore si imposta la frequenza a 434 MHz grazie alla formula: $Frf = Fstep * Frf(23:0)$.

- All’indirizzo 0x18 c’è il registro “RegLna” dedicato al settaggio del blocco LNA (Low Noise Amplifier) facente parte del ricevitore. Un amplificatore a basso rumore (LNA, low-noise amplifier) è un amplificatore elettronico utilizzato per amplificare dei segnali molto deboli (per esempio, catturati da un'antenna). Di solito è situato molto vicino al dispositivo di rilevamento, così da ridurre al massimo le perdite di precisione. Un LNA è un componente chiave, inserito nell'estremità anteriore del circuito radio-ricevitore. Scrivendo in questo registro il byte 0x08 andiamo a settare la sua impedenza di ingresso a 50 ohms e attrverso lasciando a 0 i bits 2-0 lasceremo impostare il guadagno al blocco AGC (Automatic Gain Control). Questa impostazione è utile per ottenere un compromesso ottimale tra sensibilità e linearità. Infatti così facendo quando il ricevitore è abilitato resta in modalità WAIT, finchè RssiValue non supera RssiThreshold per due campioni consecutivi, una volta verificatasi questa condizione il ricevitore seleziona autoamticamente il guadagno LNA più adatto ottimizzando il compromesso sensibilità/linearità.

- All’indirizzo 0x18 c’è il registro “RegRxBw” utile per settare alcuni parametri del Channel Filter. Il ruolo di questo blocco è quello di filtrare il rumore e gli interferenti all’esterno del canale. Il filtraggio del canale sull’ RFM69HCW è

implementato con un 16-tap Finite Impulse Response (FIR) filter (filto FIR a 16 “tappi”). Scrivendo in questo registro il byte 0x55 andiamo ad impostare la larghezza di banda (RxBw) a 10.4 kHz. Per rispettare le regole di sovracampionamento nella catena di decimazione del ricevitore, il bit-rate non può essere superiore a due volte la larghezza di banda del ricevitore.

- Agli indirizzi 0x25 e 0x26 ci sono i registri RegDioMapping1 e RegDioMapping2. Il modulo radio RFM69HCW dispone di 6 pin gpio e la loro configurazione è controllata da questi due registri. Le cose più importanti da settare in questo registro sono:
  - DIO1=00 questo pin ci segnalerà l’interrupt di FifoLevel (quando i byte nel FIFO supereranno una certa soglia).
  - ClkOut=111 cioè OFF.
  - Le altre impostazioni sono ininfluenti, il settaggio che è stato messo risultava comodo nell’utilizzo del software Piscope.
- All’indirizzo 0x29 c’è il registro “RegRssiThresh” utile come ci suggerisce il nome per impostare il valore di soglia dell’Rssi per il ricevitore o anche per generare l’interrupt qualora lo si voglia. Con il byte 0xE4 questa soglia viene impostata a $-\frac{RssiThreshold}{2} = -114\ dBm$
- All’indirizzo 0x2D abbiamo il registro “RegPreambleLsb” ovvero gli LSB byte della grandezza del preambolo da inviare dal soddsfacimento della TxStartCondition. Scrivendo il byte 0x08 la grandezza di questo preambolo è di 8 byte ovvero 64 bits di preambolo.
- All’indirizzo 0x2E “RegSyncConfig” ci si occupa della configurazione riguardo la Sync Word. Il riconoscimento della Sync Word è uno dei meccaniscmi che questo modulo offre all’utente per il filtraggio dei pacchetti, assicurando che solo i pacchetti utili siano disponibili al µC. Ogni pacchetto ricevuto che non inizia con la Sync Word configurata localmente viene automaticamente scartato. Quando viene rilevata la Sync Word, viene avviata automaticamente la ricezione del payload e viene impostato il bit SyncAddressMatch. Scrivendo il byte 0x98 imposteremo: SyncOn=1,

FifoFillCondition=0 cioè che il FIFO iniziera a riempirsi solo dopo che il bit SyncAddressMatch si sia settato a 1, SyncSize=011 ovvero lunghezza della SyncWord=011 + 1= 4bytes, SyncTol=000 cioè 0 bit di errore tollerati.

- Dal registro 0x2F al registro 0x32 viene configurata la SyncWord partendo dal MSB(0x2F) all' LSB (0x32 in questo caso).
- All'indirizzo 0x37 c'è il registro RegPacketConfig serve per impostare alcune impostazioni per la gestione dei pacchetti. Scrivedoci il byte 0x88 andremo ad impostare il formato dei pacchetti a lunghezza variabile, nessuna codifica, nessun controllo CRC e nessun AddressFiltering.
- All'indirizzo 0x38 abbiamo il registro PayloadLength, in questo registro ci abbiamo scritto il byte 0xFF in quanto stiamo lavorando con pacchetti a lunghezza variabile e come conseguenza di questa scelta in questo registro va scritta la massima lunghezza di un pacchetto accettata in Rx. Scrivendo 0xFF equivale a dire che in ricezione ci aspettiamo pacchetti lunghi massimo 255 bytes, il massimo possibile.
- All'inidirizzo 0x3C abbiamo RegFifoThresh, in questo registro abbiamo scritto il byte 0x8F. Abbiamo settato TxStartCondition=1, in questo modo il modulo una volta entrato in modalità Tx iniziera a trasmettere non appena è disponibile un byte nel FIFO (FifoNotEmpty). Per i bits FifoThreshold vengono scritto i bit:0001111 quindi questa soglia viene impostata a 15 bytes. Sarà questa soglia a far settare l'interrupt FifoLevel.
- All'inidirizzo 0x6F abbaimo il registro RegTestDagc. Oltre al controllo automatico del guadagno (AGC), l'RFM69HCW è in grado di regolare continuamente il proprio guadagno nel dominio digitale, dopo che è avvenuta la conversione da analogico a digitale. Questa funzionalità, denominata DAGC, è completamente trasparente per l'utente finale. È sempre consigliato di abilitare il DAGC dal costruttore.
- Alla penultima riga si va scrivere 0x04 all'indirizzo 0x04, così viene messo il modulo in STDBY mode.

- All’ultima riga abbiamo scritto 0x00,0x00 nella parte del codice di iniazilizzazione il programma riconoscerà questa riga come termine del vettore di inizializzazione e terminerà la configyrazione del modulo.
-

Rimangono due cose da fare, dichiarare tutte le funzioni presenti nel codice e designare una variabile, inizializzandola, che tiene memoria dello stato della radio:

```
static uint8_t rfm69hcw_state = RFM69HCW_STATE_UNINITIALIZED;

static irqreturn_t rfm69hcw_process_event (int irq, void* dev_id);
static void goIdle (struct rfm_data *data);
static void goReceive (struct rfm_data *data);
static void goTransmit (struct rfm_data *data);
static void goAbort (struct rfm_data *data);
static void feedTxFifo (struct rfm_data *data, size_t len);
static void emptyRxFifo (struct rfm_data *data, size_t len);
```

Le funzioni goIdle e goAbort serviranno per mandare il modulo in STDBY mode e in particolare in goIdle ci sarà anche il reset dei buffer di ricetrasmissione, goTransmit servirà per mandare il modulo in TX mode, goReceive in Rx mode, feedTxFifo servirà per scrivere i byte da trasmettere nel FIFO mentre emptyRxFifo per leggere i bytes ricevuti.

Come di consueto in fondo al codice si aggiungono delle informazioni generali del device driver:

```
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Danilo Pichilli");
MODULE_DESCRIPTION("kernel driver for RFM69HCW radio modules");
MODULE_VERSION("0.0.1");
```

6.2 Inserimento e rimozione

All’inserimento ed alla rimozione del modulo vengono chiamate, rispettivamente, le funzioni designate da `module_init` e `module_exit`:

```
module_init(rfm_module_init);
module_exit(rfm_module_exit);
```

Si andranno a descrivere quindi queste due funzioni:

```
static int __init rfm_module_init (void)
{
	int ret;
```

```
	ret = rfm_register_chrdevices();
	if (ret) return ret;

	rfm_class = class_create(THIS_MODULE, RFM_CLS_NAME);
	if (IS_ERR(rfm_class)) {
		rfm_unregister_chrdevices();
		ret = PTR_ERR(rfm_class);
		printk(KERN_ALERT RFM_DRV_NAME ": failed to create class
			(%d)\n", ret);
		return ret;
	}

	ret = spi_register_driver(&rfm_driver);
	if (ret) {
		class_destroy(rfm_class);
		rfm_unregister_chrdevices();
		printk(KERN_ALERT RFM_DRV_NAME ": failed to register spi
			driver (%d)\n", ret);
		return ret;
	}

	printk(KERN_INFO RFM_DRV_NAME ": driver loaded.\n");

	return ret;
}
```

Come già precedentemente annunciato la funzione `rfm_module_init` viene chiamata quando il modulo viene inserito. La variabile `ret` viene utilizzata per controllare l'eventuale presenza di errori andando ad assegnare a tale variabile il valore di ritorno di alcune funzioni.

Come prima cosa viene invocata la funzione `rfm_register_chrdevices :`

```
static int rfm_register_chrdevices (void)
{
	int ret;
	dev_t dev;

	if (rfmdev_major) {
		dev = MKDEV(rfmdev_major, rfmdev_minor);
		ret = register_chrdev_region(dev, RFM_MAX_MINORS,
						RFM_DRV_NAME);
		if (ret)
			printk(KERN_ALERT RFM_DRV_NAME ": can't register major
						%d (%d)\n", rfmdev_major, ret);
	} else {
		ret = alloc_chrdev_region(&dev, rfmdev_minor, RFM_MAX_MINORS,
						RFM_DRV_NAME);
		if (ret)
			printk(KERN_ALERT RFM_DRV_NAME ": can't get a major
							number (%d)\n", ret);
```

```
		else
			rfmdev_major = MAJOR(dev);
	}
	return ret;
}
```

Questa si occupa semplicemente di creare il char device (`MKDEV(rfmdev_major, rfmdev_minor)`) ed assegnare i device number tramite la funzione `register_chrdev_region` se già si è a conoscenza del major number (`if(rfmdev_major)`). [7]

In effetti una delle prime cose che il driver deve fare quando si crea un dispositivo char è quello di ottenere uno o più device numbers da utilizzare. La funzione necessaria per questa attività è `register_chrdev_region`.

Qui `dev` è il device number iniziale dell'intervallo che si desidera assegnare. `RFM_MAX_MINORS` è il numero totale di device numbers contigui che si sta richiedendo.

Infine, `RFM_DRV_NAME` è il nome del dispositivo che dovrebbe essere associato a questo intervallo numerico (determinato da RFM_MAX_MINORS); apparirà in */proc/devices* e *sysfs*. Come per la maggior parte delle funzioni del kernel, il valore di ritorno da `register_chrdev_region` sarà 0 se l'allocazione è stata eseguita correttamente. In caso di errore verrà restituito un codice di errore negativo e non sarà possibile accedere alla regione richiesta.

Se invece non si conosce a priori il major number viene chiamata la funzione `alloc_chrdev_region`. Questa funzione è usata quando non si sa a priori quali numeri andrà ad utilizzare il dispositivo e fornisce device numbers allocati dinamicamente.

Con questa funzione, `dev` è un parametro solo di uscita che, al termine del completamento, tiene il primo numero nell'intervallo assegnato, `rfmdev_minor` dovrebbe essere il primo minor number richiesto da utilizzare; è di solito 0. I parametri `RFM_MAX_MINORS` e `RFM_DRV_NAME` funzionano come quelli passati a `register_chrdev_region`.

Tornando alla funzione di partenza si passa alla chiamata di altre due funzioni.

La prima è `class_create(THIS_MODULE, RFM_CLS_NAME)` che come suggerisce il nome si occupa di creare la classe relativa al modulo rfm (`rfm_class`).

Infatti: `rfm_class=class_create(THIS_MODULE, RFM_CLS_NAME)`.

La seconda, cioè `spi_register_driver(&rfm_driver)`, è utilizzata per registrare il driver SPI denominato `rfm_driver` e definito come segue:

```
static struct spi_driver rfm_driver = {
	.driver = {
		.name = "RFM69 driver",
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(rfm_of_ids),
	},
	.id_table = rfm_ids,
	.probe    = rfm_dev_probe,
	.remove   = rfm_dev_remove,
};

MODULE_ALIAS("spi:RFM69HCW");
```

Come si nota nella struttura viene dapprima indicato il device driver di pertinenza e questo è rappresentato dalla struttura "`.driver`". All'interno di quest'ultima si trovano il nome (`RFM69 driver`), il proprietario (`THIS_MODULE`) e i device compatibili.

```
static const struct of_device_id rfm_of_ids[] = {
	{
		.compatible = "RFM69HCW",
		.data = (void *) 69,
	},
	{},
};

MODULE_DEVICE_TABLE(of, rfm_of_ids);
```

Successivamente c'è il campo `.id_table` dove si comunicano i dispositivi compatibili `rfm_ids`.

```
static const struct spi_device_id rfm_ids[] = {
	{"RFM69HCW", 0},
	{},
};

MODULE_DEVICE_TABLE(spi, rfm_ids);
```

In ultimo c'è l'assegnazione delle due funzioni `rfm_dev_probe` e `rfm_dev_remove` che corrispondono rispettivamente alla registrazione e alla deregistrazione hardware dei dispositivi. Queste due funzioni verranno chiamate all'inserimento e alla rimozione del device tree (si approfondirà questo aspetto più avanti). Per ora ci si limita alla spiegazione del contenuto.

```
static int rfm_dev_probe (struct spi_device *spi)
{
	struct rfm_data *data;
	int ret;
	unsigned long minor;

	data = devm_kzalloc(&spi->dev, sizeof *data, GFP_KERNEL);
	if (!data) {
	  printk(KERN_ALERT RFM_DRV_NAME ": unable to allocate memory
	failed\n");
		return -ENOMEM;
	}

	data->spi = spi;
	data->open = 0;
	spin_lock_init(&data->lock);
	init_waitqueue_head(&data->wait_read);
	init_waitqueue_head(&data->wait_write);

	rfm69hcw_state = RFM69HCW_STATE_RESETTING;
	printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);
	//RESET DEL MODULO
	gpio_direction_output(23,1);
	msleep(1);
	gpio_direction_input(23);
	msleep(6);

	int ready=0;
	u8 reg_ready[2];
	u8 rx_ready[2];
	int err_ready;
	struct spi_transfer tr_ready = {
				.tx_buf			= reg_ready,
				.rx_buf			= rx_ready,
				.len			= 2,
				.cs_change		= 0,
				.bits_per_word	= 0,
				.delay_usecs		= 0,
				.speed_hz		= 25000
	  };
	while(ready == 0 ){
		struct spi_message msg_ready = {{0}};
		reg_ready[0]= 0x27;
		reg_ready[1]=0x00;
		spi_message_init(&msg_ready);
		spi_message_add_tail(&tr_ready, &msg_ready);
```

```
			err_ready = spi_sync(data->spi, &msg_ready);
			udelay(100);
			if (err_ready) printk(KERN_INFO RFM_DRV_NAME": READY
POLLING error\n");
			ready= rx_ready[1] & ISR_MODEREADY;
	}
	if (ready){
		rfm69hcw_state=RFM69HCW_STATE_INITIALIZING;
		printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);
		const uint8_t *p_conf = RFM69HCWP_config;
		/* Send the configuration table to the RFM69HCW: */
		     while (0 != *p_conf){
				int err_conf;
				struct spi_message msg_conf;
				struct spi_transfer tr_conf = {
					.tx_buf		= p_conf,
					.rx_buf		= NULL,
					.len		= 2,
					.cs_change	= 0,
					.bits_per_word	= 0,
					.delay_usecs	= 0,
					.speed_hz	= 25000
			};
			spi_message_init(&msg_conf);
			spi_message_add_tail(&tr_conf, &msg_conf);
			err_conf = spi_sync(data->spi, &msg_conf);
			udelay(100);
			if (err_conf) printk(KERN_INFO RFM_DRV_NAME": STATE
INIZIALIZING error\n");
			p_conf += 2;
			 }
			 printk(KERN_INFO RFM_DRV_NAME": RFM69HCW
Inizialized\n");
			 rfm69hcw_state = RFM69HCW_STATE_ABORTED;
			 printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);
	}
	ret = devm_request_threaded_irq(&spi->dev, spi->irq, NULL,
rfm69hcw_process_event,IRQF_TRIGGER_HIGH | IRQF_ONESHOT, RFM_DRV_NAME,
data);
	if (ret) {
		printk(KERN_ALERT RFM_DRV_NAME ": unable to get interrupt
(%d)\" , nret);
		return ret;
	}

	minor = rfm_alloc_minor();
	if (minor < RFM_MAX_MINORS) {
		/* Minor found, create device: */
		cdev_init(&data->cdev, &rfm_fops);
		data->cdev.owner = THIS_MODULE;
		ret = cdev_add(&data->cdev, MKDEV(rfmdev_major, minor), 1);
		if (ret) {
			printk(KERN_ALERT RFM_DRV_NAME ": cdev_add failed
(%d)\n", ret);
```

```
                return ret;
            }
            data->dev = device_create(rfm_class, &spi->dev, data-
>cdev.dev, data, RFM_DEV_NAME "-%d.%d",spi->master->bus_num, spi-
>chip_select);
            ret = IS_ERR(data->dev) ? PTR_ERR(data->dev) : 0;
            if (ret) {
                printk(KERN_ALERT RFM_DRV_NAME ": device_create failed
(%d)\n " , ret);
                return ret;
            }

        } else {
            dev_dbg(&spi->dev, "no minor number available!\n");
            ret = -ENODEV;
        }

        if (!ret) {
            spi_set_drvdata(spi, data);
            printk(KERN_INFO RFM_DRV_NAME ": added transceiver %s-
%d.%d\n", RFM_DEV_NAME, spi->master->bus_num, spi->chip_select);
        }

        return ret;
}
```

Come prima cosa viene allocata una porzione di memoria per la struttura `struct rfm_data *data` con la macro `devm_kzalloc` (`devm` indica che la memoria è associata al modulo e che quindi essa verrà liberata automaticamente alla rimozione del modulo stesso).

Se ci sono errori vengono comunicati all'utente (`if(!data)`).

Successivamente vengono inizializzate le due code `wait_read` e `wait_write` tramite la funzione `init_waitqueue_head`. Le code sono utilizzate per "addormentare" un processo in attesa che qualcosa lo svegli. Il codice che si occupa del risveglio deve essere in grado di trovare il processo per essere in grado di fare il suo lavoro. E questo è proprio il compito delle code.

Viene anche inizializzato uno spinlock tramite `spin_lock_init(&data->lock)`. Gli spinlocks sono semplici da capire concettualmente. Uno spinlock è un dispositivo di esclusione reciproca che può avere solo due valori: "bloccato" e "sbloccato".

Normalmente viene implementato come singolo bit in un valore intero. Il codice che desidera prelevare un blocco particolare verifica il bit relativo. Se il blocco è disponibile, il bit "bloccato" viene impostato e il codice continua nella sezione critica.

Se invece la chiusura è stata presa da qualcun altro, il codice entra in un ciclo in cui cerca di prelevare il blocco di interesse finché non diventa disponibile.

Quindi lo spinlock è utilizzato per impedire ad altri processi di usufruire di una risorsa mentre questa è già utilizzata da altri processi. [7]

Viene poi settato lo stato `rfm69hcw_state = RFM69HCW_STATE_RESETTING.` A questo punto avviene il reset del modulo portando alto il valore del GPIO collegato al pin del reset sulla radio, aspettando 1ms e poi rilasciandolo.

A questo punto viene utilizzato un ciclo `while` per attendere che si verifichi l'interrupt ModeReady. Di defeault,dopo il reset,il modulo andrà in modalità STDBY. Questo ciclo while consiste nell'andare a leggere ciclicamente il registro che contiene il flag dell'interrupt ModeReady (0x27).

In ambiente Linux la trasmissione è gestita tramite due strutture, `spi_transfer` e `spi_message`. Quest'ultima è usata per eseguire una sequenza "atomica" di trasferimento di dati, ognuno di essi rappresentati dalla struttura `spi_transfer`. La sequenza è "atomica" nel senso che nessun altro `spi_message` può usufruire del bus SPI in questione finché la sequenza non è completa.

Grazie alla funzione `spi_message_add_tail` è possibile accodare più `spi_transfer` che poi andranno tutti spediti in modalità sincrona tramite la funzione `spi_sync`. Prima di queste ultime il messaggio va inizializzato con la funzione `spi_message_init`.

Come si può notare la struttura spi_message `msg_ready` è inizializzata a zero come si usa fare se non si ha particolari specifiche. Essendo una trasmissione sincrona ogni byte che viene scritto corrisponde ad uno che viene letto. Questa è la motivazione della presenza sia di un buffer di trasmissione (`reg_ready`) sia di un buffer di ricezione (`rx_ready`) di lunghezza identica. Questi due buffer possono anche essere posti al valore `NULL` dipendentemente se si vuole tener conto solo della lettura o della scrittura.

Un altro parametro della struttura spi_transfer è la lunghezza in bytes (`.len`) della porzione di messaggio da trasferire. Gli altri due parametri, `bits_per_word` e `delay_usecs`, sono lasciati a zero, cioè al valore di default.

Come si può intuire il parametro `speed_hz` indica la frequenza di trasmissione che in questo caso è settata ad un valore prescelto. L'ultimo parametro che resta da analizzare è `cs_change` che se uguale a 0 va a disattivare il chip select una volta finita la trasmissione.

Per andare a scrivere (o a leggere) su un registro bisogna comunicare alla radio come prima cosa l'indirizzo del registro e poi cosa andare a scrivere eventualmente. Quindi per conoscere lo stato del bit ModeReady al primo elemento del vettore `reg_ready` diamo il valore 0x27. Il contenuto di questo registro sarà memorizzato nel secondo elemento del vettore `rx_ready` (`rx_ready[1]`).

Una volta accertati che questo flag sia settato, imposteremo lo stato `rfm69hcw_state= RFM69HCW_STATE_INITIALIZING` .

Quindi possiamo procedere con la configurazione del nostro modulo, attraverso un ciclo `while` che controllerà la fine del vettore di configurazione, qui vengono effettuate le varie trasmissioni SPI con l'incremento di due posizioni del puntatore al vettore di configurazione tra registri successivi. Una volta finita la configurazione verrà comunicato tramite il comando `printk` e verrà aggiornato lo stato con `rfm69hcw_state = RFM69HCW_STATE_ABORTED` in quanto sappiamo dal vettore di configurazione che il modulo viene lasciato in STDBY mode.

Avviene poi la registrazione dell'interrupt con la funzione `devm_request_threaded_irq`. Come precedentemente assegnato il prefisso devm_ serve per indicare che l'interrupt è legato al modulo e che quindi alla rimozione di quest'ultimo viene eliminato automaticamente anche l'interrupt. Viene richiesto un interrupt threaded, cioè una volta ricevuto l'interrupt la sua gestione viene spezzata tra due funzioni, l'handler che in questo caso è `NULL` (terzo parametro passato alla funzione) e il thread (quarto parametro). Non indicando l'handler la gestione passa direttamente nelle mani del thread `rfm69hcw_process_event` che corrisponde alla macchina a stati che verrà esposta successivamente. Nel quinto campo che si passa alla funzione di registrazione dell'interrupt si indica il comportamento di quest'ultimo: l'opzione `IRQF_TRIGGER_HIGH` indica che il riconoscimento dell'interrupt avviene sul livello (alto in questo caso) e non sui fronti di salita e di discesa; `IRQF_ONESHOT` impone il rilascio dell'interrupt una volta finita la sua gestione.

A questo punto avviene la chiamata alla funzione `rfm_alloc_minor` che si occupa di trovare il minor number per lo specifico dispositivo:

```
static int rfm_alloc_minor (void)
{
	int i;

	BUILD_BUG_ON(RFM_MAX_MINORS > BITS_PER_LONG);

	for (i = 0; i < RFM_MAX_MINORS; ++i)
		if (!test_and_set_bit(i, &rfm_minors))
			break;

	return i;
}
```

La funzione `BUILD_BUG_ON` interrompe la compilazione se la condizione passata risulta vera. Sappiamo che `BITS_PER_LONG` è 64. [8]

La funzione `test_and_set_bit` imposta un bit e restituisce il suo vecchio valore prendendo come input il numero di bit da impostare e il puntatore alla memoria. Questo finchè `i<RFM_MAX_MINORS` oppure la funzione `test_and_set_bit` restituisce 0.

A questo punto se il minor number è stato trovato si procede con la creazione della `struct cdev` relativa con la chiamata a `cdev_init` per poi informare il kernel con `cdev_add`.

La funzione `device_create` crea il dispositivo e lo registra in sysfs. [8] A questa funzione notiamo che vengono passati come parametri il puntatore alla classe in cui dovrebbe essere registrato il dispositivo(`rfm_class`), i dati da aggiungere al dispositivo per le richiamate(`data`), la stringa per il nome del dispositivo (`RFM_DEV_NAME`) che in questo caso lo ricordiamo è: RFM69, il bus num, il chip select. Questa funzione restituisce il puntatore del dispositivo alla struttura in caso di successo o `ERR_PTR` in caso di errore.

Come al solito ad ogni chiamata di registrazione c'è un controllo errori che provoca l'uscita dal processo e la conseguente notifica all'utente.

Se tutto procede senza errori viene comunicato con un messaggio di avvenuta aggiunzione del dispositvo con il `bus_num` e il relativo `chip_select`.

Tutto quello che viene effettuato nella rfm_dev_probe viene “distrutto” nella rfm_dev_remove che come detto viene chiamata alla rimozione del dispositivo oppure alla rimozione del driver dal kernel (che provoca la rimozione del dispositivo).

```
static int rfm_dev_remove (struct spi_device *spi)
{
	unsigned long flags;
	struct rfm_data *data = spi_get_drvdata(spi);

	printk(KERN_INFO RFM_DRV_NAME ": removed transceiver %s-%d.%d\n",
			RFM_DEV_NAME, spi->master->bus_num, spi->chip_select);

	spin_lock_irqsave(&data->lock, flags);

	data->spi = NULL;
	spi_set_drvdata(spi, NULL);

	spin_unlock_irqrestore(&data->lock, flags);

	device_destroy(rfm_class, data->cdev.dev);
	clear_bit(MINOR(data->cdev.dev), &rfm_minors);

	return 0;
}
```

La funzione spin_lock_irqsave è usata per salvare lo stato degli interrupt. Dopodichè con spin_unlock_irqrestore riabiliteremo gli interrupt e utilizzeremo la funzione device_destroy. Questa chiamata annulla la registrazione e ripulisce un dispositivo creato con una chiamata a device_create () prendendo come argomenti il puntatore alla classe struct con cui è stato registrato questo dispositivo e il dev_t del dispositivo precedentemente registrato. [8]

Per quanto riguarda l’inserimento e la rimozione rimane soltanto da descrivere la controparte della rfm_module_init cioè rfm_module_exit:

```
static void __exit rfm_module_exit (void)
{
	spi_unregister_driver(&rfm_driver);
	class_destroy(rfm_class);
	rfm_unregister_chrdevices();

	/* Unregister gpio */
	gpio_free(23);

	printk(KERN_INFO RFM_DRV_NAME ": driver unloaded.\n");
}
```

Questa si occupa di effettuare una pulizia generale di quello che era stato registrato in `rfm_module_init`. Deregistrare il driver SPI con `spi_unregister_driver`, distruggere la classe `rfm_class` con `class_destroy` ed eliminare i char devices con la chiamata a `rfm_unregister_chrdevices`:

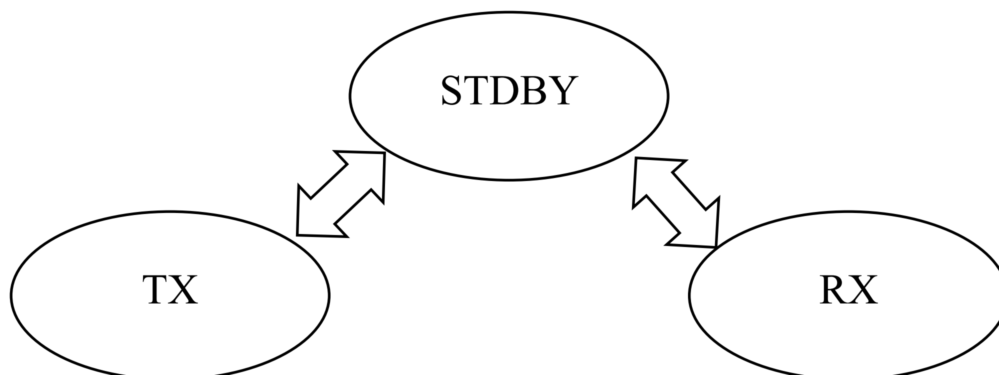
```
static void rfm_unregister_chrdevices (void)
{
	dev_t dev = MKDEV(rfmdev_major, rfmdev_minor);
	unregister_chrdev_region(dev, RFM_MAX_MINORS);
}
```

Indipendentemente dal modo in cui sono stati assegnati i device numbers, devono essere liberati quando non sono più in uso. I numeri dei dispositivi vengono liberati con la funzione: `unregister_chrdev_region (dev_t first, unsigned int count)`.

Infine si libera il GPIO 23 mediante `gpio_free` e si comunica l'avvenuta rimozione del modulo dal kernel.

6.3 State Machine

Il principio di funzionamento del modulo radio si basa su una macchina a stati descritta dal seguente schema:



Come si è già detto la macchina a stati nel codice costituisce il gestore dell'interrupt quindi ogni qual volta il modem invia una segnalazione al master viene effettuata immediatamente la chiamata alla funzione `rfm69hcw_process_event`:

```
static irqreturn_t rfm69hcw_process_event (int irq, void* dev_id)
{
    struct rfm_data *data = dev_id;
	int err;

	/* Read interrupt cause: */
	u16 irq_status;
```

```
        struct spi_message msg = {{0}};
        u8 isr[3];
        u8 txbuff[3];
        struct spi_transfer tr = {
                .tx_buf              = txbuff,
                .rx_buf              = isr,
                .len                 = 3,
                .cs_change           = 0,
                .bits_per_word       = 0,
                .delay_usecs         = 0,
                .speed_hz            = 25000
        };
        txbuff[0] = 0x27;
        txbuff[1] = 0x00;
        txbuff[2] = 0x00;

    spi_message_init(&msg);
    spi_message_add_tail(&tr, &msg);
    err = spi_sync(data->spi, &msg);
    udelay(100);

    irq_status = (isr[1] << 8) + isr[2];

    switch(rfm69hcw_state){
            case RFM69HCW_STATE_ABORTED:
                    goReceive(data);
                    break;
            case RFM69HCW_STATE_RECEIVING:
                    if (irq_status & ISR_FIFOLEVEL){
                            printk(KERN_INFO RFM_DRV_NAME": Fifo level in
                            rfm69hcw_state= receiving.\n");
                            /* RX FIFO almost full */
                            rfm69hcw_state = RFM69HCW_STATE_RECEIVING;
                            printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
                                        %d\n",rfm69hcw_state);
                    if (rfm69hcw_rx_buffer.length >
                    RFM69HCW_PACKET_MAX_LENGTH-
                    RFM69HCW_RX_FIFO_AFULL) {
                                    goAbort(data);
                                    return 0;
                            } else {
                                    emptyRxFifo(data, RFM69HCW_RX_FIFO_AFULL);
                                    misura_disponibile=
rfm69hcw_rx_buffer.length;
                                    goReceive(data);
                                    /* Signal data reception and pass the packet
                                    to the upper layer */
                                    wake_up_interruptible(&data->wait_read);
                                    break;
                            }
                    }

                    if (irq_status & ISR_TIMEOUT) {
                            goAbort(data);
                    }
                    break;
```

```
            case RFM69HCW_STATE_RECEIVE:
                  if (irq_status & ISR_FIFOLEVEL){
                        printk(KERN_INFO RFM_DRV_NAME": Fifo level in
                                    rfm69hcw_state= receive.\n");
                        /* RX FIFO almost full */
                        rfm69hcw_state = RFM69HCW_STATE_RECEIVING;
                        printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
                                    %d\n",rfm69hcw_state);
                  if (rfm69hcw_rx_buffer.length >

                  RFM69HCW_PACKET_MAX_LENGTH - RFM69HCW_RX_FIFO_AFULL) {
                              goAbort(data);
                              return 0;
                        } else {
                              emptyRxFifo(data, RFM69HCW_RX_FIFO_AFULL);
                              misura_disponibile=
rfm69hcw_rx_buffer.length;
                              goReceive(data);
                              /* Signal data reception and pass the packet
to                            the upper layer */
                              wake_up_interruptible(&data->wait_read);
                              break;
                        }
                  }
      }

      return IRQ_HANDLED;
}
```

Ogni volta che il modulo radio invia un interrupt, viene chiamata questa funzione. La prima cosa che viene fatta è la lettura della causa dell'interrupt. Occore andare a leggere i registri 0x27 e 0x28 cioè rispettivamente RegIrqFlags1 e RegIrqFlags2. Per compiere questa lettura sui due registri con indirizzi di memoria contigui si utilizza un accesso BURST. In questo tipo di accesso il byte dell'indirizzo, in questo caso 0x27, è seguito da diversi byte. L'indirizzo viene incrementato automaticamente internamente tra ogni byte di dati. Il pin CS diventa basso all'inizio del frame e rimane basso tra ogni byte, tornerà alto solo dopo l'ultimo trrasmerimento di byte.

Una volta effettuata la transizione si potrà conoscere la causa dell'interrupt dato che isr[1] e isr[2] conterranno i valori dei registri 0x03 e 0x04. Si va a costruire quindi la variabile `irq_status` a cui si vanno ad assegnare i 16 bit complessivi corrispondenti alla causa dell'interrupt.

Arrivati a questo punto il programma procede in base allo stato `rfm69hcw_state` e va ad effettuare le operazioni a seconda della causa dell'interrupt andando a confrontare `irq_status` con le costanti definite ad inizio codice.

Se il modulo si trova in STDBY mode (`rfm69hcw_state=RFM69HCW_STATE_ABORTED`), il modulo verrà mandato in ricezione tramite la funzione `goReceive` che verrà analizzata in seguito.

Se invece il modulo si trova in ricezione (`rfm69hcw_state=RFM69HCW_STATE_RECEIVE),` si va a verificare l'interrupt FifoLevel per capire se effettivamente il modulo abbia ricevuto qualcosa. Una volta che questa verifica da un riscontro positivo viene aggiornato lo stato con `rfm69hcw_state=RFM69HCW_STATE_RECEIVING.` Successivamente viene fatta una verifica: se il buffer di ricezione è ancora libero, si procede con la lettura del FIFO altrimenti si manda il modulo in STDBY. Una volta completata la ricezione si aggiorna il valore di `misura_disponibile`, si rimanda in ricezione il modulo e si sveglia il processo con il comando `wake_up_interruptible(&data->wait_read)` qualora sia stato addormentato tramite la funzione di lettura `rfm_filop_read.`

6.4 File operations

6.4.1 Open

Le file operations utilizzate in questo driver sono solamente tre e sono definite nella struttura:

```
static const struct file_operations rfm_fops = {
   .owner                = THIS_MODULE,
   .write                = rfm_filop_write,
   .read                 = rfm_filop_read,
   .open                 = rfm_filop_open,
};
```

Questa struttura è definita in <linux/fs.h> ed è una raccolta di punatori a funzione.

Ogni file aperto (rappresentato internamente da una struttura file) è associato al proprio set di funzioni (includendo un campo chiamato `f_op` che punta a una struttura `file_operations`). Le operazioni sono prevalentemente incaricate di implementare le chiamate di sistema e sono quindi chiamate open, read e così via. Possiamo considerare il file come un "oggetto" e le funzioni che operano su di esso i suoi "metodi", utilizzando la terminologia di programmazione a oggetti.

Convenzionalmente, una struttura `file_operations` o un puntatore ad una di esse viene chiamata `fops`.

La prima file operation che si va a descrivere è la `rfm_filop_open` che viene chiamata quando si tenta di aprire il file nella directory */dev* corrispondente al dispositivo. Il nome del file per questo specifico driver è *RFM69-0.1* (come abbiamo visto in fase di inizializzazione).

Questa funzione prende come ingressi un puntatore alla struttura `inode` e uno alla struttura `file`.

La file structure rappresenta un file aperto. (Non è specifico per i device driver, ogni file aperto del sistema ha una struttura file associata nello spazio del kernel.) Viene creata dal kernel all'apertura e viene passata a qualsiasi funzione che opera sul file fino all'ultima chiusura. Dopo che tutte le istanze del file sono chiuse, il kernel rilascia la struttura. Nelle origini del kernel, un puntatore alla file structure viene chiamato in genere `file` o `filp`. Quindi si conviene che file si riferisce alla struttura e `filp` ad un puntatore alla struttura. Come anticipato in precedenza ogni file aperto è associato al proprio set di funzioni tramite la dichiarazione: `struct file_operations *f_op;`

La struttura inode invece viene utilizzata dal kernel internamente per rappresentare i file. È diversa dalla di file structure che rappresenta la descrizione di un file aperto. Ci possono essere numerose file structure che rappresentano più descrizioni di un singolo file, ma tutte si riferiscono ad una singola struttura inode. La struttura inode contiene una grande quantità di informazioni sul file. Come regola generale, un campo di questa struttura è di interesse per la scrittura del codice del char driver:

```
struct cdev *i_cdev;
```

`struct cdev` è la struttura interna del kernel che rappresenta i dispositivi char; questo campo contiene un puntatore a quella struttura quando l'inode si riferisce a un device file char.

```
static int rfm_filop_open (struct inode *inode, struct file *filp)
{
	int err=0;
	struct rfm_data *data = container_of(inode->i_cdev, struct rfm_data,
	                                     cdev);
	unsigned long flags;
	if(!err){
	  spin_lock_irqsave(&data->lock, flags);
```

```
        if(!err){
            data->open++;
            filp->private_data=data;
            nonseekable_open(inode, filp);
            }
            spin_unlock_irqrestore(&data->lock, flags);
    }
    else pr_debug("rfm69hcw: nothing for minor %d\n", iminor(inode));
    return err;
}
```

Come prima cosa nella funzione viene assegnato un valore al puntatore data tramite la macro `container_of`.

`container_of(ptr, type, member)` trasmette un membro di una struttura alla struttura contenitore : @ptr: il puntatore al membro ,@type: il tipo di struttura del contenitore in cui è incorporato , @member: il nome del membro all'interno della struttura. [8]

Semplicemente questa è una macro che può essere usata per ottenere un puntatore a una struttura da un puntatore ad un'altra struttura contenuta in esso.

Viene poi utilizzato uno spinlock con la funzione `spin_lock_irqsave` e in assenza di errori viene incrementato il contatore `data->open` ed assegnato a `filp->private_data` il puntatore data. La macro `nonseekable_open` serve per informare il kernel che il driver non supporta il metodo *lseek* (metodo di ricerca nel file che solitamente il kernel implementa modificando `filp->f_pos`, cioè la posizione attuale di scrittura/lettura all'interno file). Viene infine rilasciato lo spinlock con la funzione `spin_unlock_irqrestore`. [7] [8]

6.4.2 Scrittura

L'operazione di scrittura nel file viene utilizzata per far trasmettere al modulo radio proprio ciò che si è scritto nel file. Per andare a scrivere si fa uso del comando *echo*:

echo "ciao" > /dev/RFM69-0.1

Nell'esempio si va a scrivere "ciao" nel file */RFM69-0.1* nella directory */dev*.

La funzione che gestisce la scrittura è `rfm_filop_write`:

```
static ssize_t rfm_filop_write (struct file *filp, const char __user *buf,
size_t count, loff_t *f_pos)
```

```
{
	struct rfm_data *data = (struct rfm_data *) filp->private_data;
	int err;
	if(count<RFM69HCW_TX_FIFO_LEN){
	/* Reset state machine */
	goIdle(data);

	rfm69hcw_tx_buffer.length = count;
	err = copy_from_user(rfm69hcw_tx_buffer.data +
					rfm69hcw_tx_buffer.offset,buf,count);
	if(err) printk(KERN_INFO RFM_DRV_NAME": filop_write->copy_from_user
error 			%d\n", err);

	//impostare DIO1 su FIFOfull per evitare l'interrupt
	u8 tx_diomapping[2];
	u8 rx_diomapping[2];
	struct spi_message msg_diomapping = {{0}};
	int err_diomapping;
	struct spi_transfer tr_diomapping = {
				.tx_buf			= tx_diomapping,
				.rx_buf			= rx_diomapping,
				.len			= 2,
				.cs_change		= 0,
				.bits_per_word		= 0,
				.delay_usecs		= 0,
				.speed_hz		= 25000
	  };

	tx_diomapping[0]= 0x25 + 0x80;
	tx_diomapping[1]= 0x11;
	spi_message_init(&msg_diomapping);
	spi_message_add_tail(&tr_diomapping, &msg_diomapping);
	err= spi_sync(data->spi, &msg_diomapping);
	if (err) printk(KERN_INFO RFM_DRV_NAME": Error dio mapping
filop_write 				%d\n", err);
	udelay(100);

	feedTxFifo(data, RFM69HCW_TX_FIFO_LEN);

	goTransmit(data);
	int interrupt=0;
	u8 reg_interrupt[2];
	u8 rx_interrupt[2];
	int err_interrupt;
	struct spi_transfer tr_interrupt = {
				.tx_buf			= reg_interrupt,
				.rx_buf			= rx_interrupt,
				.len			= 2,
				.cs_change		= 0,
				.bits_per_word		= 0,
				.delay_usecs		= 0,
				.speed_hz		= 25000
	  };
	while(interrupt == 0){
		struct spi_message msg_interrupt = {{0}};
		reg_interrupt[0]= 0x28;
```

```
            reg_interrupt[1]=0x00;
            spi_message_init(&msg_interrupt);
            spi_message_add_tail(&tr_interrupt, &msg_interrupt);
            err_interrupt = spi_sync(data->spi, &msg_interrupt);
            udelay(100);
        if (err_interrupt) printk(KERN_INFO RFM_DRV_NAME": INTERRUPT
POLLING                                              TRANSMIT error\n");
        interrupt= rx_interrupt[1] & ISR_PACKETSENT;
        if(interrupt){
            /* Packet sent */
                        printk(KERN_INFO RFM_DRV_NAME": Packet sent\n");
                        goReceive(data);
            }
      }
      return count;
}
      if(count>RFM69HCW_TX_FIFO_LEN){
            if(count<RFM69HCW_PACKET_MAX_LENGTH){

      /* Reset state machine */
      goIdle(data);
      rfm69hcw_state = RFM69HCW_STATE_TRANSMIT;
      rfm69hcw_tx_buffer.length = count; //La variabile count corrisponde
al numero di byte scritti con il comando echo (pag.50 tesi)

      err = copy_from_user(rfm69hcw_tx_buffer.data +
rfm69hcw_tx_buffer.offset,buf,count);
      if(err) printk(KERN_INFO RFM_DRV_NAME": filop_write->copy_from_user
error %d\n", err);
      //feed tx fifo
      u8 txbuff[2];
      struct spi_message msg = {{0}};
      struct spi_transfer tr = {
                        .tx_buf           = txbuff,
                  .rx_buf           = NULL,
                  .len              = 2,
                    .cs_change        = 0,
                  .bits_per_word    = 0,
                  .delay_usecs      = 0,
                  .speed_hz         = 25000
      };

      txbuff[0] = 0x00 + 0x80;
      txbuff[1]= count; //il primo byte scritto nel FIFO è il LengthByte
      spi_message_init(&msg);
      spi_message_add_tail(&tr, &msg);
      err = spi_sync(data->spi, &msg);
      udelay(100);
      int fifofull;
      int a=count;
      int i=0;
      while (a>0){
            struct spi_message msg1 = {{0}};
            u8 txbuff1[2];
            struct spi_transfer tr1 = {
                  .tx_buf           = txbuff1,
```

```
		.rx_buf		= NULL,
		.len		= 2,
		.cs_change	= 0,
		.bits_per_word	= 0,
		.delay_usecs	= 0,
		.speed_hz	= 25000
			};
	struct spi_message msg2 = {{0}};
	u8 txbuff2[2];
	u8 rxbuff2[2];
	struct spi_transfer tr2 = {
		.tx_buf		= txbuff2,
		.rx_buf		= rxbuff2,
		.len		= 2,
		.cs_change	= 0,
		.bits_per_word	= 0,
		.delay_usecs	= 0,
		.speed_hz	= 25000
			};
	fifofull=0;
	while(fifofull==0){
		if(a>0){
		txbuff1[0]= 0x00 + 0x80;
		txbuff1[1]=*(rfm69hcw_tx_buffer.data +
				rfm69hcw_tx_buffer.offset + i);
		i++;
		a--;
		spi_message_init(&msg1);
		spi_message_add_tail(&tr1, &msg1);
		err = spi_sync(data->spi, &msg1);
		udelay(100);
		txbuff2[0]= 0x28;
		spi_message_init(&msg2);
		spi_message_add_tail(&tr2, &msg2);
		err = spi_sync(data->spi, &msg2);
		fifofull= rxbuff2[1] & 0x80;
		}
		else fifofull=1;
	}
	printk(KERN_INFO RFM_DRV_NAME": Byte scritti nel fifo: %d\n",
i);
	goTransmit(data);
	u8 interrupt=0x40;
	u8 reg_interrupt[2];
	u8 rx_interrupt[2];
	int err_interrupt;
	struct spi_transfer tr_interrupt = {
			.tx_buf		= reg_interrupt,
			.rx_buf		= rx_interrupt,
			.len		= 2,
			.cs_change	= 0,
			.bits_per_word	= 0,
			.delay_usecs	= 0,
			.speed_hz	= 25000
	};
    while(interrupt != 0){
```

```
            struct spi_message msg_interrupt = {{0}};
            reg_interrupt[0]= 0x28;
            reg_interrupt[1]=0x00;
            spi_message_init(&msg_interrupt);
            spi_message_add_tail(&tr_interrupt, &msg_interrupt);
            err_interrupt = spi_sync(data->spi, &msg_interrupt);
            udelay(100);
            interrupt= rx_interrupt[1] & 0x40; //     }
            printk(KERN_INFO RFM_DRV_NAME": Fifo empty. \n");
      }
      rfm69hcw_tx_buffer.offset = rfm69hcw_tx_buffer.offset + i;
      printk(KERN_INFO RFM_DRV_NAME": Count of bytes sent: %d\n", count);
      u8 packet_sent=0x00;
      u8 reg_interrupt_packet_sent[2];
      u8 rx_interrupt_packet_sent[2];
      int err_interrupt_packet_sent;
      struct spi_transfer tr_interrupt_packet_sent = {
                  .tx_buf            = reg_interrupt_packet_sent,
                  .rx_buf            = rx_interrupt_packet_sent,
                  .len               = 2,
                  .cs_change         = 0,
                  .bits_per_word     = 0,
                  .delay_usecs       = 0,
                  .speed_hz          = 25000
        };
      while(packet_sent == 0){
            struct spi_message msg_interrupt_packet_sent = {{0}};
            reg_interrupt_packet_sent[0]= 0x28;
            reg_interrupt_packet_sent[1]=0x00;
            spi_message_init(&msg_interrupt_packet_sent);
            spi_message_add_tail(&tr_interrupt_packet_sent,
                                          &msg_interrupt_packet_sent);
            err_interrupt_packet_sent = spi_sync(data->spi,

&msg_interrupt_packet_sent);
            udelay(100);
        packet_sent= rx_interrupt_packet_sent[1] & ISR_PACKETSENT;
       }
      printk(KERN_INFO RFM_DRV_NAME": Packet sent\n");
      goReceive(data);
      }
      }
      return count;
}
```

Iniziando ad analizzare questa funzione notiamo che dopo aver dichiarato variabili e strutture già analizzate in precedenza, viene controllata la variabile `count` se è minore della lunghezza del FIFO. La variabile `count` viene passata alla funzione come parametro e corrisponde al numero di byte scritti con il comando *echo*. Questo controllo è necessario in quanto i pacchetti di grandi dimensioni, cioè più lunghi del FIFO (66 byte), richiedono una gestione particolare che verrà spiegata in seguito.

Andremo ora a vedere come si procederà per pacchetti con dimensioni inferiori a 66 byte.

Come prima cosa avviene la chiamata alla funzione `goIdle`:

```
static void goIdle (struct rfm_data *data)
{
	int err;
	/* Abort all operations */
	struct spi_message msg = {{0}};
	u8 txbuff1[2];
	struct spi_transfer tr1 = {
			.tx_buf		= txbuff1,
			.rx_buf		= NULL,
			.len		= 2,
			.cs_change	= 0,
			.bits_per_word	= 0,
			.delay_usecs	= 0,
			.speed_hz	= 25000
	};
	txbuff1[0] = 0x01 + 0x80;
	txbuff1[1] = 0x04;

    spi_message_init(&msg);
    spi_message_add_tail(&tr1, &msg);
	err = spi_sync(data->spi, &msg);
	udelay(100);
	if(err){
		printk(KERN_ALERT RFM_DRV_NAME "error goIdle %d", err);}

	/* Reset buffers */
	rfm69hcw_rx_buffer.offset = 0;
	rfm69hcw_rx_buffer.length = 0;
	rfm69hcw_tx_buffer.offset = 0;
	rfm69hcw_tx_buffer.length = 0;
	rfm69hcw_state = RFM69HCW_STATE_ABORTED;
	printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);

	//impostare DIO1 su FIFOlevel
	u8 tx_diomapping[2];
	u8 rx_diomapping[2];
	struct spi_message msg_diomapping = {{0}};
	int err_diomapping;
	struct spi_transfer tr_diomapping = {
			.tx_buf		= tx_diomapping,
			.rx_buf		= rx_diomapping,
			.len		= 2,
			.cs_change	= 0,
			.bits_per_word	= 0,
			.delay_usecs	= 0,
			.speed_hz	= 25000
	  };

	tx_diomapping[0]= 0x25 + 0x80;
```

```
      tx_diomapping[1]= 0x01;
    spi_message_init(&msg_diomapping);
    spi_message_add_tail(&tr_diomapping, &msg_diomapping);
    err= spi_sync(data->spi, &msg_diomapping);
    if (err) printk(KERN_INFO RFM_DRV_NAME": Error dio mapping filop_write
%d\n", err);
        udelay(100);

}
```

Lo prima cosa che viene fatta è mandare in STDBY il modulo, stoppando tutte le operazioni, scrivendo 0x04 all'indirizzo 0x01. Successivamente vengono resettati i buffer di trasmissione e ricezione, viene aggiornato lo stato con `rfm69hcw_state = RFM69HCW_STATE_ABORTED` e come ultima cosa si va ad impostare il pin DIO1 sull'interrupt FifoLevel scrivendo 0x01 all'indirizzo 0x25. Questa operazione viene fatta in quanto successivamente in fase di trasmissione il pin DIO1 viene impostato sull'interrupt FifoFull per evitare che in caso di scrittura sul Fifo di un numero di byte superiori alla soglia FifoThreshold, si generi un interrupt che sistematicamente attivi la relativa funzione leggendo i registri RegIrqFlags 1 e 2 fino a quando il FIFO non riscenda sotto la soglia.

Tornando alla funzione di scrittura, la lunghezza del buffer di trasmissione viene impostata uguale a `count`, dopodichè si prosegue con la funzione `copy_from_user` : il comando *echo* è utilizzato nello spazio utente quindi bisogna trasferire l'informazione nello spazio kernel tramite la funzione `copy_from_user` (la controparte di questa funzione è la `copy_to_user` per copiare dallo spazio kernel allo spazio utente); per effettuare questa copia si vanno ad indicare la posizione di destinazione (`rfm23_tx_buffer.data + rfm23_tx_buffer.offset`), la posizione dov'è contenuta l'informazione nello spazio utente (`buf`) e la lunghezza in byte da copiare (`count`).

Successivamente viene impostato DIO1 su FifoFull come accennato in precedenza, in quanto subito dopo si passa alla funzione con il quale andremo a scrivere nel FIFO: `feedTxFifo(data, RFM69HCW_TX_FIFO_LEN)`.

```
static void feedTxFifo (struct rfm_data *data, size_t len)
{
      uint8_t i = 0, err;
      struct spi_message msg = {{0}};

      size_t remaining = rfm69hcw_tx_buffer.length -
rfm69hcw_tx_buffer.offset;
      if (len > remaining) len = remaining;
      else len=0;
```

```
        if (!len) return;
        {
        u8 txbuff[len+2];
        struct spi_transfer tr = {
                 .tx_buf          = txbuff,
                 .rx_buf          = NULL,
                 .len             = len + 2,
                 .cs_change       = 0,
                 .bits_per_word   = 0,
                 .delay_usecs     = 0,
                 .speed_hz        = 25000
        };

        txbuff[0] = 0x00 + 0x80;
        txbuff[1]= len;
        for(i = 0; i < len; i++){
              txbuff[i+2] = *(rfm69hcw_tx_buffer.data +
rfm69hcw_tx_buffer.offset                            + i);
        }

        spi_message_init(&msg);
        spi_message_add_tail(&tr, &msg);
        err = spi_sync(data->spi, &msg);
        udelay(100);

        rfm69hcw_tx_buffer.offset = rfm69hcw_tx_buffer.offset + len;
        }
        printk(KERN_INFO RFM_DRV_NAME": Count of bytes sent: %d\n", len);
}
```

Qui inizialmnte vengono fatte ulteriori verifiche sulle dimensioni del pacchetto da trasmettere, poi viene semplicemente costruito il buffer di trasmissione dove al primo elemento si pone il registro sul quale andare a scrivere (0x00), al secondo elemento la lunghezza del pacchetto da trasmettere e poi tutti i restanti byte. Fondamentale è il secondo elemento, infatti l'RFM69HCW in caso di utilizzo di pacchetti a lunghezze variabili richiede che il primo byte scritto nel FIFO dall'utente sia proprio la lunghezza del pacchetto da trasmettere (a differenza del modulo RFM23B che aveva un registro dedicato).

La scrittura di questi dati avviene con un metodo FifoAccess. Quando il primo indirizzo corrisponde a quello del FIFO il modulo radio non varierà l'indirizzo per la durata di tutta la scrittura.

Viene aggiornato l'offset. A questo punto tornando alla funzione di partenza si può mandare in TXmode il modulo chiamando la funzione goTransmit:

```
static void goTransmit (struct rfm_data *data)
{
        int err;
```

```
	struct spi_message msg = {{0}};
	u8 txbuff[2];
	struct spi_transfer tr = {
			.tx_buf 		= txbuff,
			.rx_buf 		= NULL,
			.len 			= 2,
			.cs_change 		= 0,
			.bits_per_word 	= 0,
			.delay_usecs 	= 0,
			.speed_hz 		= 25000
	};

	rfm69hcw_state = RFM69HCW_STATE_TRANSMIT;
	printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);

	txbuff[0] = 0x01 + 0x80;
	txbuff[1] = 0x0C;
	spi_message_init(&msg);
	spi_message_add_tail(&tr, &msg);
	err = spi_sync(data->spi, &msg);
	udelay(100);
		if(err){
		printk(KERN_ALERT RFM_DRV_NAME "error goTransmit %d", err);}

}
```

In questa funzione viene mandato il modulo in trasmissione semplicemente scrivendo 0x0C al registro 0x01.

Ora che il modulo sta trasmettendo i dati, tramite il ciclo while andremo a controllare periodicamente il bit dell'interrupt PacketSent che si setta quando l'intero pacchetto è stato inviato. Una volta verificato ciò si manda il modulo in RxMode con la funzione goReceive che analizzeremo in seguito.

Passiamo ora ad analizzare il comportamento in caso si voglia trasmettere un pacchetto di grandi dimensioni.

Quando la lunghezza del payload supera la dimensione FIFO (66 byte) oltrre all'interrupt PacketSent in TXmode è possibile sfruttare gli interrupt FIFO come FifoFull o FifoLevel.

Vediamo come ci si comporta per la trasmissione. Il Fifo può essere precompilato in modalità STDBY ma deve essere ricaricato "al volo" durante la trasmissione con il resto del payload, quindi:

1. Precompilare FIFO (in modalità STDBY) fino a impostare il bit FifoFull;

2. In Tx, attendere che FifoNotEmpty sia cancellato, cioè che il FIFO sia vuoto;
3. Scrivere di nuovo byte nel FIFO fino a reimpostare il bit FifoFull;
4. Continuare dal punto 2 finchè l'intero messaggio non è stato scritto nel FIFO, dopodichè PacketSent verrà attivato quando è stato inviato l'ultimo bit del pacchetto).

Dopo aver controllato che il numero di byte scritti con il comando echo sia inferiore alla massima lunghezza del pacchetto supportata, si vanno a chiamare le funzioni descritte in precedenza `goIdle` e `copy_to_user`.

Successivamente la prima cosa che andiamo a scrivere nel FIFO come da regola è la lunghezza del pacchetto da inviare.

Inizializziamo poi una variabile di appoggio (`a=count`) che terrà conto di quanti byte dovremmo ancora scrivere, in effetti senza questa variabile usciremo dal ciclo di scrittura, descritto nei punti 1 2 3 e 4, solamente se il numero di byte da inviare è un multiplo di 66 byte. Tutta la scrittura nel FIFO infatti procederà solo finche sarà soddissfatta la condizione `a>0`.

Si procederà andando a scrivere i byte nel FIFO controllando contemporaneamente il bit FifoFull, quando questo bit si setterà usciremo dal ciclo while e manderemo il modulo in TXmode tramite la funzione goTransmit.

Una volta mandato in trasmissione il modulo controlleremo ciclicamente che il bit FifoNotEmpty si riporti a 0, una volta verificata questa condizione se a > 0 (a tiene conto dei byte restanti da scrivere) ricominceremo a scrivere gli altri byte nel FIFO. Se i byte da scrivere sono finiti, andiamo a verificare l'avvenuta trasmissione leggendo il bit PacketSent comunicandolo con un `printk`.

Infine verra messo il modulo in RXmode tramite la funzione `goReceive`: ù

```
static void goReceive (struct rfm_data *data)
{
	int err;

	struct spi_message msg = {{0}};
	u8 txbuff[2];
	struct spi_transfer tr = {
			.tx_buf		= txbuff,
			.rx_buf		= NULL,
			.len		= 2,
```

```
                    .cs_change        = 0,
                    .bits_per_word    = 0,
                    .delay_usecs      = 0,
                         .speed_hz         = 25000
      };

      goIdle (data);

      rfm69hcw_state = RFM69HCW_STATE_RECEIVE;
      printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);

      txbuff[0] = 0x01 + 0x80;
      txbuff[1] = 0x10;
      spi_message_init(&msg);
      spi_message_add_tail(&tr, &msg);
      err = spi_sync(data->spi, &msg);
      udelay(100);
      if(err){
            printk(KERN_ALERT RFM_DRV_NAME "error goReceive %d", err);}
}
```

Qui come prima cosa viene chiamata la funzione goIdle utile per mandare in STDBY il modulo ma soprattutto per resettare i buffer di trasmissione e ricezione. Dopodiche viene aggiornato lo stato `rfm69hcw_state = RFM69HCW_STATE_RECEIVE` e poi viene mandato il modulo in ricezione scrivendo il byte 0x10 al registro 0x01.

6.4.3 Lettura

Per come è strutturata la macchina a stati ed il codice in generale la ricezione può avvenire solo dopo la trasmissione di un pacchetto tramite la chiamata alla funzione che abbiamo gia esaminato `goReceive`.

A questo punto si può andare a leggere il file con il comando *strace cat*:

strace cat /dev/RFM69-0.1 > /dev/null

L’istruzione *strace* serve per tracciare l’esecuzione di qualsiasi eseguibile. Solitamente il comando mostra tutte le chiamate di sistema per il dato eseguibile.

A questo punto interviene la funzione che si occupa di effettuare la lettura chiamata in questo caso `rfm_filop_read`:

```
static ssize_t rfm_filop_read (struct file* filp, char __user *buf, size_t
                                          count, loff_t* f_pos)
{
      struct rfm_data *data = (struct rfm_data *) filp->private_data;
      int ret;
      int length;
```

```
	wait_event_interruptible(data->wait_read, (misura_disponibile !=
0));

	length = misura_disponibile;
	if (count < length) length = count;
	ret = copy_to_user(buf, rfm69hcw_rx_buffer.data, length);

	misura_disponibile = 0;

	return length;
}
```

La prima cosa che accade è l'addormentamento del processo tramite `wait_event_interruptible`. Questa macro manda in sleep il processo avviando la coda `data->wait_read` e lo risveglia quando si avvera la condizione "`misura_disponibile !=0`".

Come si può intuire nella macchina a stati questa condizione diventa vera quando la radio ha ricevuto qualcosa sotto al caso `RFM69HCW_STATE_RECEIVE / RFM69HCW_STATE_RECEIVING`. Attenzione, qui se la lunghezza del pacchetto da ricevere è maggiore della soglia "`RFM69HCW_PACKET_MAX_LENGTH - RFM69HCW_RX_FIFO_AFULL`" viene chiamata la funzione `goAbort`:

```
static void goAbort (struct rfm_data *data)
{
	goIdle(data);
	rfm69hcw_state = RFM69HCW_STATE_ABORTED;
	printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);
}
```

Questa si occupa di abortire tutte le operazioni ed effettuare un reset. Altrimenti si passa il compito di svuotare il rxFIFO alla funzione `emptyRxFifo`:

```
static void emptyRxFifo (struct rfm_data *data, size_t len)
{
	int err;
	if (!len) return;
	u8 interrupt_payloadready = 0x00;

	while(interrupt_payloadready==0){
	int i=0;
	for (i = 0; i < len; i++){
		struct spi_message msg = {{0}};
		u8 txbuff[2];
		u8 rxbuff[2];
		struct spi_transfer tr = {
			.tx_buf		= txbuff,
			.rx_buf		= rxbuff,
```

```
                .len                = 2,
                .cs_change          = 0,
                .bits_per_word      = 0,
                .delay_usecs        = 0,
                .speed_hz           = 25000
        };

        txbuff[0] = 0x00;

        spi_message_init(&msg);
        spi_message_add_tail(&tr, &msg);
        err = spi_sync(data->spi, &msg);
        udelay(100);

        *(rfm69hcw_rx_buffer.data + rfm69hcw_rx_buffer.offset) =

rxbuff[1];
        rfm69hcw_rx_buffer.offset++;
        rfm69hcw_rx_buffer.length++;

        struct spi_message msg_fifo_not_empty = {{0}};
        u8 txbuff_fifo_not_empty[2];
        u8 rxbuff_fifo_not_empty[2];
        struct spi_transfer tr_fifo_not_empty = {
                        .tx_buf             = txbuff_fifo_not_empty,
                        .rx_buf             = rxbuff_fifo_not_empty,
                        .len                = 2,
                        .cs_change          = 0,
                        .bits_per_word      = 0,
                        .delay_usecs        = 0,
                        .speed_hz           = 25000
        };
        txbuff_fifo_not_empty[0] = 0x28;

        spi_message_init(&msg_fifo_not_empty);
        spi_message_add_tail(&tr_fifo_not_empty,
&msg_fifo_not_empty);
        err = spi_sync(data->spi, &msg_fifo_not_empty);
        udelay(100);
        if(!(rxbuff_fifo_not_empty[1] & ISR_FIFONOTEMPTY)) i=len;
    }
    struct spi_message msg_payloadready = {{0}};
        u8 txbuff_payloadready[2];
        u8 rxbuff_payloadready[2];
        struct spi_transfer tr_payloadready = {
                .tx_buf             = txbuff_payloadready,
                .rx_buf             = rxbuff_payloadready,
                .len                = 2,
                .cs_change          = 0,
                .bits_per_word      = 0,
                .delay_usecs        = 0,
                .speed_hz           = 25000
        };
```

```
            txbuff_payloadready[0] = 0x28;
            spi_message_init(&msg_payloadready);
            spi_message_add_tail(&tr_payloadready, &msg_payloadready);
            err = spi_sync(data->spi, &msg_payloadready);
            udelay(100);
            interrupt_payloadready= rxbuff_payloadready[1] & 0x04;
}
      printk(KERN_INFO RFM_DRV_NAME": Payload Ready\n");
      /* Abort all operations */
      struct spi_message msg = {{0}};
      u8 txbuff1[2];
      struct spi_transfer tr1 = {
                 .tx_buf           = txbuff1,
                 .rx_buf           = NULL,
                 .len              = 2,
                 .cs_change        = 0,
                 .bits_per_word    = 0,
                 .delay_usecs      = 0,
                 .speed_hz         = 25000
      };
      txbuff1[0] = 0x01 + 0x80;
      txbuff1[1] = 0x04;


     spi_message_init(&msg);
     spi_message_add_tail(&tr1, &msg);
     err = spi_sync(data->spi, &msg);
     udelay(100);
     if(err){printk(KERN_ALERT RFM_DRV_NAME "error goIdle %d", err);}
     rfm69hcw_state = RFM69HCW_STATE_ABORTED;
     printk(KERN_INFO RFM_DRV_NAME": rfm69hcw_state=
%d\n",rfm69hcw_state);
     u8 fifoempty=0x40;
     while(fifoempty){
            struct spi_message msg = {{0}};
            u8 txbuff[2];
            u8 rxbuff[2];
            struct spi_transfer tr = {
                  .tx_buf           = txbuff,
                  .rx_buf           = rxbuff,
                  .len              = 2,
                  .cs_change        = 0,
                  .bits_per_word    = 0,
                  .delay_usecs      = 0,
                  .speed_hz         = 25000
            };

            txbuff[0] = 0x00;
            spi_message_init(&msg);
            spi_message_add_tail(&tr, &msg);
            err = spi_sync(data->spi, &msg);
            udelay(100);
```

```
		*(rfm69hcw_rx_buffer.data + rfm69hcw_rx_buffer.offset) = 
	rxbuff[1];
		rfm69hcw_rx_buffer.offset++;
		rfm69hcw_rx_buffer.length++;

		struct spi_message msg_fifo_empty = {{0}};
		u8 txbuff_fifo_empty[2];
		u8 rxbuff_fifo_empty[2];
		struct spi_transfer tr_fifo_empty = {
			.tx_buf		= txbuff_fifo_empty,
			.rx_buf		= rxbuff_fifo_empty,
			.len		= 2,
			.cs_change	= 0,
			.bits_per_word	= 0,
			.delay_usecs	= 0,
			.speed_hz	= 25000
		};
		txbuff_fifo_empty[0] = 0x28;

		spi_message_init(&msg_fifo_empty);
		spi_message_add_tail(&tr_fifo_empty, &msg_fifo_empty);
		err = spi_sync(data->spi, &msg_fifo_empty);
		udelay(100);
		fifoempty= fifoempty & rxbuff_fifo_empty[1];
		}
	printk(KERN_INFO RFM_DRV_NAME": Count of bytes received: %d\n",
			rfm69hcw_rx_buffer.length);
}
```

Dunque abbiamo visto che questa funzione viene chiamata quando l'interrupt FifoLevel viene settato, cioè non appena viene ricevuto un numero di byte superiore FifoThreshold (15 bytes).

Bisogna far notare che in ricezione, lavorando con pacchetti a lunghezza variabile, il primo byte che deve essere ricevuto dopo la SyncWord deve essere quello che indica la lunghezza del pacchetto che si sta ricevendo, cosicchè il modulo possa settare l'interrupt PayloadReady non appena sia stato ricevuto l'ultimo byte.

Qui non abbiamo potuto fare una distinzione tra pacchetti di grandi dimensioni e piccole dimensioni non sapendo a priori questa informazione.

Come prima cosa dobbiamo far notare che il modulo scarterà automaticamente tutti i pacchetti con dimensione maggiore a quella scritta nel registro "RegPayloadLength" 0x38.

In ricezione ci seguiremo questo schema:

1. Iniziare a leggere i byte non appena FifoLevel viene impostato;
2. Sospensione della lettura del FIFO se FifoNotEmpty si cancella prima che tutti i byte del messaggio siano letti;
3. Continuare al passaggio 1 fino a quando PayloadReady non si attiva;
4. Leggere tutti i byte rimanenti nel FIFO in RXmode o STDBY.

Questo è proprio ciò che avviene, si comincia leggendo i byte dal FIFO e controllando contemporaneamente che il bit FifoNotEmpty si cancelli. Una volta che abbiamo finito di leggere i 15 bytes oppure si sia cancellato il bit che ci interessa, andremo a controllare il bit dell' interrupt PayloadReady.

Se questo bit è settato vuol dire che tutto il messaggio è stato ricevuto, altrimenti si torna indietro a "risvuotare" il FIFO.

Una volta ricevuto tutto si mette il modulo in STDBYmode scrivendo 0x04 al registro 0x01. Si concluderà leggendo i restanti byte presenti nel FIFO fino a quando il bit FifoNotEmpty non si cancella, ciò vorra dire che l'intero pacchetto è stato letto.

A questo punto tornando alla funzione `rfm69hcw_process_event` si aggiorna il valore di misura_disponibile e si sveglia il processo di lettura tramite `wake_up_interruptible`.

6.5 Installazione

Per iniziare si è provveduto a scaricare il sistema operativo Raspberry Pi OS dall'apposito sito poi lo si è ovviamente installato su una MicroSD che funge da memoria di massa per la Raspberry.

Per collegare la board ad un PC si utilizza un cavo Ethernet classico per effettuare una comunicazione tramite protocollo SSH.

Si prosegue avviando il sistema operativo: dopo aver fatto un update generale e aver abilitato la comunicazione SSH, è necessario installare il pacchetto *rpi-source* che consente di costruire moduli caricabili per il Kernel. Si consiglia di effettuare tutte le operazioni come superuser: basta eseguire il comando *sudo bash*.

Spostandosi nella directory dove si trova il codice con il comando *cd* non resta che compilare il codice con *make* che svolge quello che è indicato nel *Makefile*:

```
obj-m += rfm69.o

KVERSION := $(shell uname -r)

ifneq ("$(wildcard
$(/lib/modules/$(KVERSION)/build/include/generated/uapi/linux/version.h))"
,"")
	EXTRA_CFLAGS := -
I/lib/modules/$(KVERSION)/build/include/generated/uapi/
endif

all:
	make -C /lib/modules/$(KVERSION)/build $(INCLUDE) M=$(PWD) modules
clean:
	make -C /lib/modules/$(KVERSION)/build $(INCLUDE) M=$(PWD) clean
```

Compilato il codice verranno a crearsi alcuni file tra cui *rfm69.ko* che corrisponde al modulo da caricare con il comando *insmod*, quindi si procede con:

insmod rfm69.ko

A questo punto il driver è inserito ma manca l'inserimento del dispositivo che avviene quando il kernel viene provvisto del device tree relativo. Il device tree è una struttura di dati per la descrizione dell'hardware. La struttura può contenere qualsiasi tipo di dati in quanto è internamente rappresentata come un albero di nodi e proprietà. I nodi contengono le proprietà e nodi "bambini", mentre le proprietà sono coppie costituite da nome-valore. I device tree vengono utilizzati dal software del sistema operativo, in particolare dal kernel.

I device tree possono essere incorporati in un software come il kernel Linux per gestire i casi in cui il loader di avvio non è consapevole del device tree del dispositivo di interesse. Con quest'ultima modalità si va a comunicare al kernel il driver per RFM69. Il contenuto del device tre descritto nel file RFM69.dts è:

```
/dts-v1/;
/plugin/;

/ {
	compatible = "brcm,bcm2835", "brcm,bcm2708", "brcm,bcm2709";

	fragment@0 {
		target = <&spidev1>;
		__overlay__ {
			status = "disabled";
		};
	};
	fragment@1 {
		target = <&spi0>;
```

```
		__overlay__ {
			status = "okay";
			#address-cells = <1>;
			#size-cells = <0>;

			radio69: RFM69HCW@1 {
				compatible = "RFM69HCW";
				reg = <1>;
				spi-max-frequency = <1000000>;
				interrupt-parent = <&gpio>;
				#interrupt-cells=<1>;
				interrupts = <27 4>;
				/*
					1 = low-to-high edge triggered
					2 = high-to-low edge triggered
					4 = active high level-sensitive
					8 = active low level-sensitive
				*/
			};
		};
	};

	fragment@2 {
		target = <&gpio>;
		__overlay__ {
			radio69_pins: radio69_pins {
				brcm,pins = <27>;      /* gpio no. */
				brcm,function = <0>;   /* 0:in, 1:out */
				brcm,pull = <1>;       /* 2:up 1:down 0:none */
			};
		};
	};


};
```

Quindi questo file va a descrivere i collegamenti tra Raspberry e RFM69HCW.

Anch’esso ha bisogno di essere compilato ed aggiunto al kernel. Per la compilazione si procede con l’istruzione:

dtc -@ -I dts -O dtb -W no-unit_address_vs_reg -o RFM69.dtbo RFM69.dts

Questa produce un file *RFM69.dtbo* che va inserito con il comando:

dtoverlay RFM69.dtbo

A questo punto il dispositivo è registrato con la funzione `rfm_dev_probe` presente nel codice sorgente e descritta in precedenza.

Per la rimozione di tutto il modulo basta digitare:

rmmod rfm69.ko

Questa istruzione fa sì che venga chiamata la funzione `rfm_module_exit` che provvede alla deregistrazione e alla liberazione della memoria.

6.6 Funzionamento

In questo paragrafo dimostreremo brevemente l'effettivo funzionamento del modulo radio.

Dopo aver dato i comandi *dtoverlay RFM69.dtbo* e *insmod rfm69.ko*, il modulo dovrebbe essere caricato correttamente. Con il comando *dmesg | grep "RFM"* dovremmo visualizzare i seguenti messaggi:

```
root@raspberrypi:/home/pi/Desktop/ultimo/rfm23# dmesg | grep "RFM"
[   74.675098] RFM69HCW: rfm69hcw_state= 1
[   74.727879] RFM69HCW: rfm69hcw_state= 2
[   74.756712] RFM69HCW: RFM69HCW Inizialized
[   74.756724] RFM69HCW: rfm69hcw_state= 7
[   74.757319] RFM69HCW: added transceiver RFM69-0.1
[   74.757531] RFM69HCW: driver loaded.
```

Caricamento del driver.

Come abbiamo visto il modulo dopo essere stato caricato resta in STDBY mode, bastera fargli trasmettere un messaggio per farlo uscire da questa modalità. Questo messaggio d'altronde può essere un segnale per l'interlocutore per far capire che si è pronti alla comunicazione.

Dunque con il comando *echo "testo del messaggio" > /dev/RFM69-0.1* invieremo il messaggio e con il comando *dmesg* avremo un quadro analogo al seguente:

```
[  129.702630] RFM69HCW: rfm69hcw_state= 7
[  129.717554] RFM69HCW: Count of bytes sent: 19
[  129.717569] RFM69HCW: rfm69hcw_state= 5
[  129.848200] RFM69HCW: Packet sent
[  129.849411] RFM69HCW: rfm69hcw_state= 7
[  129.850621] RFM69HCW: rfm69hcw_state= 4
[  129.852271] RFM23B: Packet Valid in RFM23_STATE_RECEIVE
[  129.855693] RFM23B: Read the remaining bytes from RX FIFO in RFM23_STATE_RECEIVE
[  129.879613] RFM23B: Count of bytes received: 20
[  129.885101] RFM23B: rfm state= 4
```

Invio di un messaggio

Dall'ultimo screenshot possiamo notare anche che il modulo radio RFM23B ha effettivamente ricevuto il messaggio in maniera corretta.

Per quanto riguarda la ricezione facendo trasmettere dal modulo RFM23B il seguente messaggio: "Hello RFM69!" visualizzeremo:

```
[   117.071940] RFM23B: Count of bytes sent: 20
[   117.071958] RFM23B: rfm state= 7
[   117.073276] RFM23B: rfm state= 5
[   117.174303] RFM69HCW: Fifo level in rfm69hcw_state= receive.
[   117.174319] RFM69HCW: rfm69hcw_state= 6
[   117.195789] RFM23B: Packet sent
[   117.206325] RFM23B: rfm state= 4
[   117.221189] RFM69HCW: Payload Ready
[   117.222508] RFM69HCW: rfm69hcw_state= 7
[   117.238210] RFM69HCW: Count of bytes received: 21
[   117.239529] RFM69HCW: rfm69hcw_state= 7
[   117.240846] RFM69HCW: rfm69hcw_state= 4
```

Ricezione messaggio

Vediamo che l'interrupt Payload Ready è verificato quindi il messaggio è stato ricevuto interamente e correttamente.

Infatti per verificare la correttezza dell'operazione possiamo effetturare la lettura conj il comando: *strace cat /dev/RFM69-0.1 > /dev/null* .

```
close(3)                                = 0
fstat64(1, {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...}) = 0
openat(AT_FDCWD, "/dev/RFM69-0.1", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0600, st_rdev=makedev(0xef, 0), ...}) = 0
fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap2(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x768de000
read(3, "\24\342\200\234Hello RFM69! \342\200\235\n", 131072) = 21
write(1, "\24\342\200\234Hello RFM69! \342\200\235\n", 21) = 21
read(3,
```

Lettura RFM69

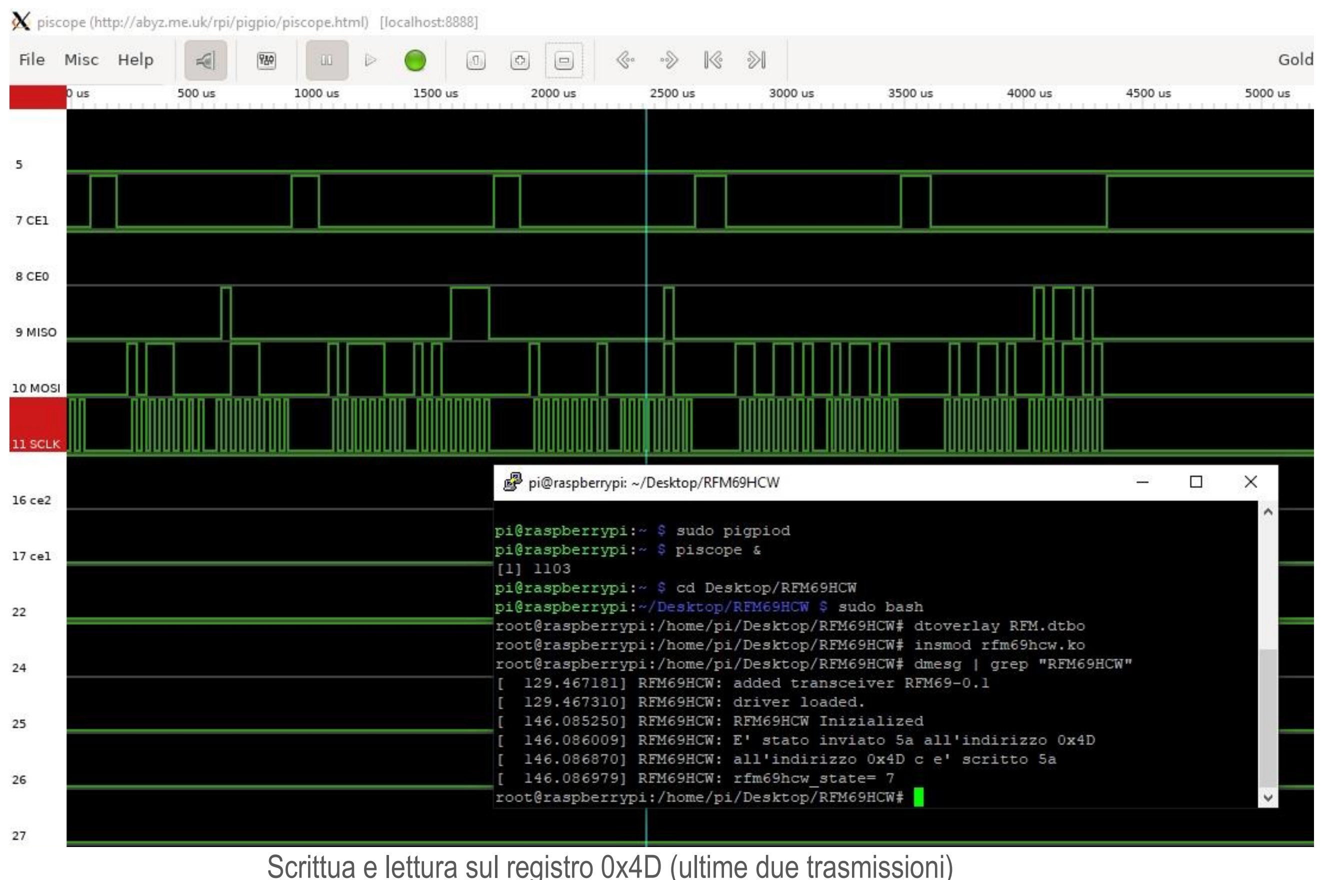
Un software molto utile per verificare la correttezza di tutti i settaggi avvenuti tramite la trasmissione SPI è Piscope.

Piscope mostra lo stato (alto o basso) dei GPIO selezionati in tempo reale.

Per poter visualizzare un'applicazione grafica X11 su Windows, abbiamo bisogno di installare un apposito server grafico sul sistema operativo di Microsoft. Il server grafico consigliato è Xming.

C'è solo un accorgimento di cui tener conto se vogliamo visualizzare correttamente le trasmissioni e cioè quello di diminuire la velocità delle trasmissioni. Basta andare a cambiare il paramentro relativo `spi_transfer.speed_hz` in tutte le trasmissioni che vogliamo visualizzare.

Nell'esempio che stiamo per mostrare siamo andati a inserire in fase di inizializzazione una parte di codice dove inizialmente abbiamo scritto un byte in un registro e subito dopo siamo andati a leggerlo proprio per verificare la corretta comunicazione. Il registro in questione si trova all'indirizzo 0x4D ed è un registro non influente ai fini del funzionamento del modulo nel nostro caso, il byte scritto è 0x5A.



Scrittua e lettura sul registro 0x4D (ultime due trasmissioni)

7.Conclusioni

Il funzionamento del driver è stato testato alla fine dello sviluppo del progetto facendo prove di trasmissione e ricezione. Per fare queste verifiche è stato utilizzato il modulo radio RFM23B prodotto dalla stessa azienda, il relativo driver era stato sviluppato da un altro studente. Dopo aver apposratto alcune modifiche per far si che i moduli comunicassero dai test svolti si può dire che il modulo kernel sviluppato funziona perfettamente sia in trasmissione che in ricezione.

Questo progetto tratta il funzionamento base di un modulo kernel e frutta le funzionalità essenziali sia della Raspberry Pi sia del RFM69HCW. È quindi possibile ampliare e rendere migliore il lavoro fin qui svolto rendendo completo il progetto.

Come esempio si può implementare nel codice anche le modalità ListenMode e AutoMode. Ovviamente tutti gli aspetti sono descritti nel data sheet del dispositivo.

Un altro modo per ampliare il funzionamento del driver è l'aggiunta della file operation ioctl. La chiamata di sistema ioctl offre un modo per emettere comandi specifici per i dispositivi (ad esempio definire la frequenza di trasmissione o la velocità di modulazione). Inoltre, alcuni comandi ioctl vengono riconosciuti dal kernel senza riferirsi alla tabella degli indirizzi `fops`. Se il dispositivo non fornisce un metodo ioctl, la chiamata di sistema restituisce un errore per qualsiasi richiesta non predefinita (-ENOTTY, "Nessun ioctl per il dispositivo"). Ovviamente chi utilizza consapevolmente il driver discusso in questa tesi andrà ad utilizzare solamente lettura e scrittura per non incappare nell'errore sopra indicato.

Un'altra questione che si può trattare è che il modulo radio trasmette e riceve qualsiasi cosa ed è trasparente a modifiche di pacchetto quali possono essere dei protocolli di trasmissione. Quindi volendo si possono implementare dei protocolli che aggiungano un'intestazione al pacchetto oppure che vanno a modificarne il contenuto tramite un processo di crittografia.

Riferimenti

[1] https://it.wikipedia.org/wiki/Raspberry_Pi.

[2] https://www.raspberrypi.org/.

[3] https://www.meccanismocomplesso.org/raspberry-pi-3-evolution/.

[4] https://it.wikipedia.org/wiki/Serial_Peripheral_Interface.

[5] http://www.microcontroller.it/Tutorials/Elettronica/SPI.htm.

[6] https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md.

[7] Linux Device Drivers, 3rd Edition - Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini.

[8] https://elixir.bootlin.com/.