POLITECNICO DI TORINO

Corso di laurea magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Implementazione di un protocollo di comunicazione safety-critical per una scheda multiprocessore

Relatore
prof. Massimo Violante

Candidato
Erica Magari
matricola: 233199

Supervisore aziendale
Ideas&Motion
ing. Marco Novaro

Anno accademico 2018-2019

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contesto della tesi

Le funzionalità di guida autonoma che si stanno affermando in campo automotive richiedono l'adozione di piattaforme computazionali estremamente performanti ma allo stesso tempo affidabili e "fault tolerant", ovvero in grado di gestire eventuali guasti senza compromettere la sicurezza degli utenti. In questo scenario, la centralina elettronica HDF (Hyper Data Fusion), sviluppata da Ideas&Motion s.r.l. nell'ambito del progetto AutoDrive, cofinanziato attraverso l'Iniziativa Tecnologica Congiunta ECSEL ed il programma quadro della Comunità Europea Horizon 2020, si prefigge di rendere disponibile una piattaforma di sviluppo ad alte prestazioni per applicazioni ADAS (Advanced Driver Assistance Systems), sfruttando la combinazione di diversi processori multi-core, in grado di unire una elevata potenza di calcolo ad una architettura di safety allo stato dell'arte. In particolare, la piattaforma hardware HDF è basata su due diversi processori che operano in stretta relazione.

Il primo processore (NXP i.MX8QuadMax) è dedicato all'acquisizione dei dati dai sensori ed alla "data fusion", ovvero è in grado di combinare le informazioni provenienti dai diversi sensori presenti sull'autoveicolo (telecamere, radar, lidar, moduli inerziali, GPS, mappe satellitari, etc) e ricostruire una mappa dell'ambiente circostante sulla quale collocare gli oggetti rilevati. Tale processore fornisce la maggior parte della potenza computazionale richiesta, grazie a due core ARM Cortex A72, quattro core ARM Cortex A53 ed alla GPU integrata.

Il secondo processore (Infineon AURIX TC297TA) è invece dedicato alla safety ed all'esecuzione delle parti di codice safety critical (riconoscimento

delle situazioni di pericolo, calcolo delle contromisure, gestione degli avvisi e allarmi verso il guidatore, attuazione delle manovre di emergenza, etc). Grazie all'architettura interna in lockstep, ed all'utilizzo di un supervisore esterno integrato in un alimentatore dedicato, è possibile supportare applicazioni fino ad ASIL D. A sua volta, il safety processor supervisiona il fusion processor e ne garantisce l'integrità operativa.

Nella suddetta architettura, è fondamentale un canale di comunicazione affidabile, sicuro e deterministico tra i due processori: una coppia di collegamenti SPI ad alta velocità, nei quali ogni processore funge da master in un collegamento SPI mentre è slave nell'altro, fornisce una adeguata banda di comunicazione a supporto dell'applicazione ed insieme la ridondanza richiesta dai requisiti di safety. Oggetto della presente tesi è lo sviluppo del software necessario, dal lato del fusion processor, ad implementare la comunicazione SPI inter-processore nelle due varianti, master e slave.

1.2 Soluzione proposta

L'architettura scelta per lo sviluppo di questo progetto sfrutta un processore NXP i.MX8QuadMax, comprendente un Cortex A53 quad core e un Cortex A72 dual core. La scheda utilizzata, su cui risiede il suddetto processore, è denominata i.MX8QuadMax lpddr4 arm2, sviluppata sempre da NXP. Per lo sviluppo di un sistema operativo compatibile con il sistema, e per lo sviluppo del protocollo di comunicazione richiesto, è stato scelto di utilizzare gli strumenti forniti dal progetto Yocto. Yocto è un progetto open source che permette di creare distribuzioni Linux emedded personalizzate in modo facile. Con gli strumenti di Yocto è possibile partire da una distribuzione base di partenza (chiamata comunemente Poky), pronta per essere utilizzata nell'architettura scelta, apportare le modifiche opportune ed aggiungere componenti. Nel corso di questo progetto si è scelto di utilizzare la versione di Yocto denominata "Morty", che utilizza al suo interno il kernel Linux 4.9.51, e si è incluso manualmente un supporto per SPI (Serial Peripheral Interface) slave, oltre che i driver implementati per lo sviluppo del protocollo di comunicazione tra i due sistemi.

Per l'implementazione dell'effettivo protocollo di comunicazione sono stati realizzati due driver, uno per la comunicazione in modalità master e uno per la comunicazione in modalità slave. Per entrambi è stata utilizzata una struttura driver a caratteri. Questi permettono di effettuare operazioni di

trasmissione e ricezione dati, trattando il sistema come un semplice file. Infatti quando si inserisce il modulo nel sistema viene creato il cosiddetto device file, un file che fa da interfaccia tra spazio utente e dispositivo reale. Alcune tra le operazioni disponibili sul device file sono apertura e chiusura, lettura e scrittura.

Una volta realizzati, i suddetti driver sono stati testati tramite degli eseguibili cross-compilati con la eSDK generata. É stato quindi provato che tramite il protocollo sviluppato i due processori possono comunicare correttamente, interpretanto chiaramente i dati ricevuti, nel giusto ordine, identificandone il significato.

1.3 Struttura della tesi

Questa tesi è sviluppata nel modo seguente.

Il capitolo 2 approfondisce i vari strumenti utilizzati nel corso del progetto, ovvero Yocto e i suoi componenti, viene spiegato nel dettaglio il protocollo SPI (Serial Peripheral Interface), che rappresenta la base del protocollo di comunicazione sviluppato in questa sede. Vi è inoltre un approfondimento sulle caratteristiche dei device driver, e più precisamente dei character driver (i driver a caratteri). Infine, sono elencate le caratteristiche principale del processore della scheda su cui è installata la distribuzione personalizzata Linux embedded, con un particolare focus sul modulo Low Power SPI (LPSPI).

Nel capitolo 3 sono elencate tutte le operazioni eseguite per la realizzazione del lavoro. Per prima cosa è descritto il procedimento per configurare l'ambiente di Yocto, e quindi realizzare la distribuzione di riferimento, Poky. Successivamente è implementato il supporto per SPI slave, con enfasi su come funzionano i trasferimenti di dati tramite SPI. Segue poi la descrizione precisa del protocollo di comunicazione scelto per il progetto, e la sua implementazione in entrambe le modalità, con la descrizione dei due driver realizzati. Infine, sono descritti i risultati ottenuti dai vari test realizzati, e il procedimento di creazione di una patch che include tutte le modifiche al kernel realizzate.

La tesi si conclude con il capitolo 4, in cui sono descritti vari possibili miglioramenti futuri.

Capitolo 2

Gli strumenti utilizzati

2.1 Il progetto Yocto

2.1.1 Il contesto del progetto

Il termine inglese build ha molteplici significati. Con esso si intende sia il processo di trasformazione di codice sorgente in prodotto eseguibile, sia lo stesso prodotto finale. Quando si tratta di progetti complessi, come ad esempio lo sviluppo di distribuzioni Linux personalizzate, la mole di file da configurare e compilare è elevata, e risulta difficile per uno sviluppatore gestire tutto il sistema in ogni sua parte. Ma esistono degli strumenti che automatizzano le operazioni di build e semplificano il lavoro degli sviluppatori. Questi strumenti sono chiamati build system.

Un build system permette di automatizzare alcune delle operazioni da svolgere nella creazione del sistema Linux Embedded. Esso si occupa ad esempio di costruire il cross-compilatore per la CPU di destinazione, includere automaticamente gli oggetti necessari al sistema finale, preparare il root file system (un file che configura l'applicazione al suo avvio, ed inizializza i vari componenti), generare l'immagine da caricare sul dispositivo, e tanto altro.

Tra i progetti che offrono build systems i più famosi ed utilizzati sono Yocto e Buildroot. Per lo sviluppo di questa tesi è stato utilizzato Yocto.

2.1.2 Le caratteristiche

Yocto è un progetto open source nato con l'obiettivo di aiutare gli sviluppatori a creare distribuzioni personalizzate di Linux per sistemi embedded.

Fornisce diversi strumenti per personalizzare e compilare molteplici piattaforme hardware e software, e supporta diverse architetture, tra cui ARM, MIPS e x86. [1]

L'ambiente di Yocto è composto da un grande numero di strumenti per lavorare con Embedded Linux, che permettono di compilare in modo automatico, testare il prodotto e documentarlo. Il vero e proprio build system di Yocto è chiamato OpenEmbedded. Sono presenti inoltre un Application Development Toolkit ed una distribuzione di riferimento, Poky, che contiene tutto il necessario per permettere al sistema di avviarsi nell'architettura scelta.

Il build system OpenEmbedded è mantenuto separatamente da Yocto, ed è composto da bitbake, uno strumento per generare le build, e dal framework software OpenEmbedded Core (OE-core). [2] Con Bitbake si possono generare le immagini da caricare nel sistema. Questo strumento è in grado di interpretare le ricette e i file di configurazione, e da essi genera un albero di dipendenze che ha come obiettivo schedulare la compilazione. I packages, ovvero i file binari generati a partire dalle ricette di partenza, sono cross-compilati durante questo processo, e da questi si ottiene l'immagine finale.

In generale i file coinvolti ed esaminati da OpenEmbedded in questo procedimento sono genericamente chiamati metadati. I metadati includono ricette, file di configurazione, istruzioni di compilazione, le versioni dei software da utilizzare e i percorsi dove risiedono questi, cambiamenti ai software stessi (ad esempio patch). Le patch sono file che applicano modifiche al codice esistente. Uno sviluppatore può modificare il codice e tramite le patch può rendere disponibile il suo lavoro alla community. [3]

Le ricette (recipe) sono un tipo di metadato contenente le istruzioni per creare i packages finali. Possono indicare dove si trova il codice da usare, le libreria da includere, oppure quali patch applicare. [3] Le ricette sono generalmente raggruppate in collezioni, chiamate layer.

Poky è la distribuzione di riferimento; è il punto di partenza per iniziare a sviluppare un sistema con Yocto. Questa infatti genera un sistema basilare ma funzionante per l'architettura scelta. Fra i vari metadati compresi nell'ambiente di Poky ci sono metadati specifici di OE-Core (meta-openembedded), di Yocto (meta-yocto) e di BSP (meta-yocto-bsp). [4]

L'Application Development Toolkit è uno strumento che permette la scrittura di codice specifico per il sistema creato, senza il bisogno che lo sviluppatore conosca precisamente come esso è strutturato. Contiene la Extensible

Software DevelopmentKit (eSDK), un Software Development Kit personalizzabile, che permette agli sviluppatori di inserire le proprie librerie, e qualora volessero possono renderle disponibili agli altri sviluppatori.

Con Yocto è possibile generare distribuzioni diverse per la stessa architettura di riferimento, ma anche rendere la stessa distribuzione personalizzata compatibile con tante architetture. Questo è reso possibile dalla struttura di Yocto, divisa in contenitori: i layer. I layer sono delle collezioni di ricette connesse tra loro. Sono utilizzati quando si hanno informazioni relative ad esempio ad architetture differenti, o a versioni diverse. Yocto fornisce diversi layer di base, che possono essere inclusi nei nuovi progetti insieme ai layer creati dallo sviluppatore. Gli sviluppatori possono aggiungere layer dove includere le loro modifiche e aggiunte. [5] OpenEmbedded-Core è un layer di metadati utilizzato da tutti i sistemi derivati da OpenEmbedded, quindi non solo da Yocto. Altri layer inclusi nell'ambiente di Yocto sono layer dedicati al Board Support Package, e layer specifici di yocto (meta-yocto). Un Board Support Package (BSP) è una collezione di informazioni che permette il supporto per una specifica piattaforma hardware. Tra le informazioni ci sono le configurazioni di kernel, i software e i driver necessari. [6]

Il progetto Yocto è interamente open source e possiede una vasta community. Viene utilizzato prevalentemente per creare sistemi Linux embedded, molto complessi. Infatti è possibile sfruttare delle distribuzioni esistenti già pronte, sviluppate da membri della community oppure fornite con Yocto stesso (la distribuzione di riferimento Poky), come punto di partenza per comporre il sistema finale. Si possono fare modifiche mirate, modificando solo quello che serve senza intaccare tutto il sistema. Ogni blocco è infatti separato dal resto. Esistono file di configurazione specifici per la macchina scelta, dove viene indicata l'architettura dell'hardware, e file di configurazione che indicano cosa deve essere installato nell'immagine finale.

Il workflow

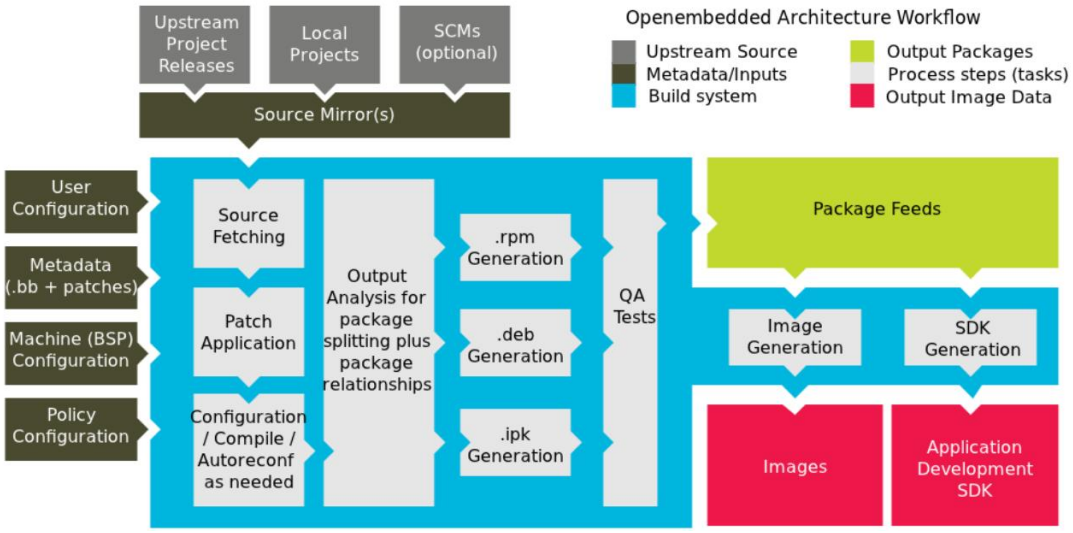

Figura 2.1. Schema generale del workflow di Yocto.

Figura 2.1 dal Yocto Mega Manual versione 2.1[7]

La creazione di una distribuzione di Linux Embedded personalizzata è un processo complesso, che può essere riassunto così di seguito. Inizialmente vengono interpretati i vari file di configurazione. Alcuni di questi specificano l' architettura selezionata, altri identificano la versione di kernel da utilizzare, le ricette da considerare e quali immagini generare. I file di metadati (di estensione .bb) sono le ricette per creare i packages. Esistono inoltre particolari file chiamati policy di distribuzione, e definiscono come ciascuna ricetta deve essere compilata. Queste policy possono essere selezionate tramite la scelta del campo DISTRO. Le opzioni predefinite sono quattro: [8]

- poky: definisce la distribuzione base.
- poky-lsb: utilizza Poky, e attiva il supporto LSB .
- poky-tiny: viene realizzato un sistema più piccolo del normale.
- poky-bleeding: una versione di poky specifica per test.

Uno dei file di configurazione più importanti è il file build/conf/local.conf che specifica come deve essere creato il prodotto finale. Di seguito è riportato un esempio con parti di un file local.conf basilare.

```
MACHINE ??= 'imx8qmlpddr4arm2'
DISTRO ?= 'fsl-imx-x11'

(...)

EXTRA_IMAGE_FEATURES ?= "debug-tweaks tools-debug"

(...)

#PACKAGECONFIG_append_pn-qemu-native = " sdl"
#PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
CONF_VERSION = "1"

(...)

ACCEPT_FSL_EULA = "1"
```

Questo codice rappresenta una parte di un semplice file di configurazione locale. Il campo MACHINE specifica l'architettura di destinazione, in questo caso la scheda NXP i.MX8 QuadMax lpddr4 arm2, utilizzata nello sviluppo di questa tesi. Nel campo DISTRO è definita la distribuzione. Vengono definiti ulteriori strumenti da includere nel campo EXTRA_IMAGE_FEATURES, come ad esempio strumenti per il debug. I simboli # rappresentano commenti, quindi comandi non considerati dal compilatore. L'ultimo campo indica che la licenza è stata accettata.

Il file seguente rappresenta invece un esempio di file di tipo bblayers.conf, in cui vengono indicati tutti i layer utilizzati dal sistema.

```
POKY_BBLAYERS_CONF_VERSION = "2"

(...)

BBFILES ?= ""
BBLAYERS = " \
  ${BSPDIR}/sources/poky/meta \
```

```
8     ${BSPDIR}/sources/poky/meta-poky \
9     \
10    ${BSPDIR}/sources/meta-openembedded/meta-oe \
11    "
12
13  (...)
14
15  # Freescale Yocto Project Release layers
16  BBLAYERS += " ${BSPDIR}/sources/meta-fsl-bsp-release/imx/
        meta-bsp "
17
18  (...)
19
20  BBLAYERS += " ${BSPDIR}/sources/meta-new "
21  BBLAYERS += " ${BSPDIR}/build-imx/workspace "
```

In questo esempio $BSPDIR/sources/meta-new è un layer generato dallo sviluppatore, $BSPDIR/sources/meta-openembedded/meta-oe rappresenta il layer di OpenEmbedded-core. $BSPDIR/build-imx/workspace è il layer dove risiedono le modifiche al kernel.

Una volta configurato Bitbake avviene una fase di fetching. Le ricette indicano il percorso dove trovare tutti i file sorgente e le patch. Una volta prelevato il codice, questo viene estratto, vengono applicate le patches e il tutto viene compilato. Sono poi creati i packages nel formato binario scelto (deb, rpm, ipk) e sono effettuati alcuni controlli per verificare che non ci siano stati errori durante il processo di build e se siano stati rispettati i requisiti. Dai binari creati vengono prodotte le immagini e una eSDK, che comprende il cross-compilatore, gli strumenti per l'analisi e un plugin di Eclipse.

2.2 Il protocollo SPI

Il protocollo SPI (Serial Peripheral Interface) è un sistema di comunicazione ideato da Motorola che permette la comunicazione tra più microcontrollori. Uno di questi dispositivi è chiamato master (padrone) e può collegarsi ad uno o più dispositivi denominati slave (schiavi). Il master è il responsabile del controllo del bus: infatti solo lui può iniziare o terminare una comunicazione. Per farlo, emette un segnale di clock alla frequenza da lui scelta. Gli slave restano in ascolto in attesa di ordini dal master e rispondono di conseguenza, alla stessa velocità. Un dispositivo, se ne ha la possibilità, può essere sia un master che uno slave, ma per esserlo deve instaurare due canali diversi di comunicazione.[9]

Il bus SPI può essere configurato sia in modalità full duplex che half duplex. Si tratta di un protocollo seriale perchè i bit sono trasferiti sequenzialmente e arrivano a destinazione nell'ordine in cui sono stati trasmessi. Per via della presenza di un segnale di clock che comanda la comunicazione, è anche definito sincrono.

Le linee di connessione tra due dispositivi sono in genere quattro. Questi quattro segnali sono:

- SCLK – Serial Clock: Il clock generato dal master, è un segnale in uscita dal master e in ingresso nello slave. Il master dovrà conoscere la velocità massima del dispositivo slave, dato che quest'ultimo dovrà rispondere obbligatoriamente alla velocità scelta dal master.

- SDI / MISO – Serial Data Input / Master In Slave Out : la linea tramite cui il master riceve il dato trasmesso dallo slave, e in cui lo slave trasmette il dato per il master.

- SDO / MOSI – Serial Data Output / Master Out Slave In: la linea tramite cui lo slave riceve il dato trasmesso dal master, e in cui il master trasmette il dato per lo slave.

- CS / SS – Chip Select / Slave Select: questo segnale viene emesso dal master, per indicare con quale slave intende comunicare. In modalità master è quindi un segnale di direzione output, mentre in modalità slave è un segnale di direzione input. Quando è attivo, lo slave capisce che è in corso una comunicazione con il master.

In presenza di più slave, si possono scegliere configurazioni differenti.

- Linee di slave select dedicate.

  Ciascuno slave è controllato singolarmente, e ogni slave deve possedere un proprio SS. Quando uno slave non è selezionato, quindi con il segnale SS disabilitato (normalmente a livello logico alto), la sua uscita risulta in alta impedenza, quindi non intralcia il master. Il numero di slave possibili con cui il master può connettersi è limitato dal numero di linee di SS che può gestire. La velocità che si può raggiungere può arrivare agli ordini dei MHz. [10] [11] Questa configurazione è vantaggiosa per la velocità della comunicazione, ma è dispendiosa in quanto necessita di più linee di Slave Select.

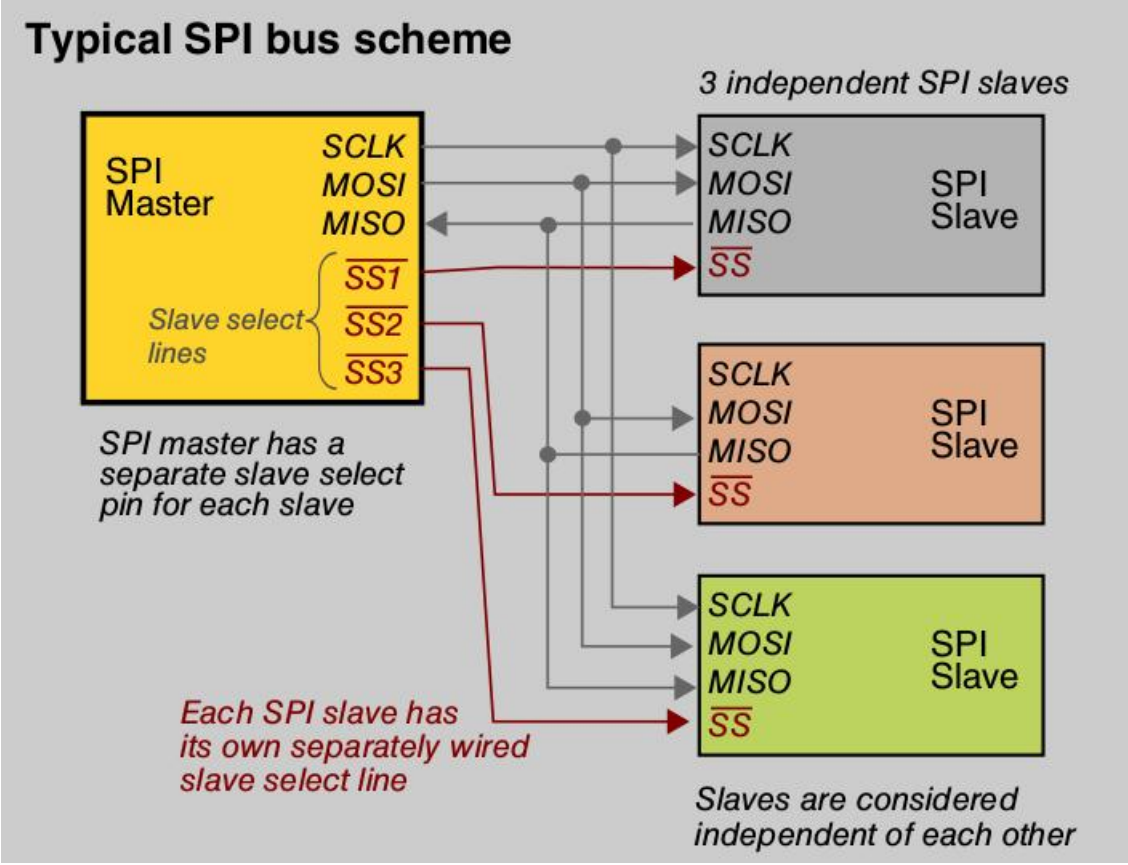

Figura 2.2. Linee di slave select dedicate

Figura 2.2 dal Manuale del processore i.MX8QuadMax, Rev. E[10]

- Daisy chain: linee di slave select condivise.

  In questo caso il master trasmette i dati partendo dal bit più significativo destinato all'ultimo slave, fino al bit meno significativo del primo. La linea MOSI è collegata unicamente al primo slave, la cui linea MISO è collegata al MOSI del secondo slave, fino all'ultimo slave, la cui linea MISO è effettivamente collegata alla linea MISO del master. [10]

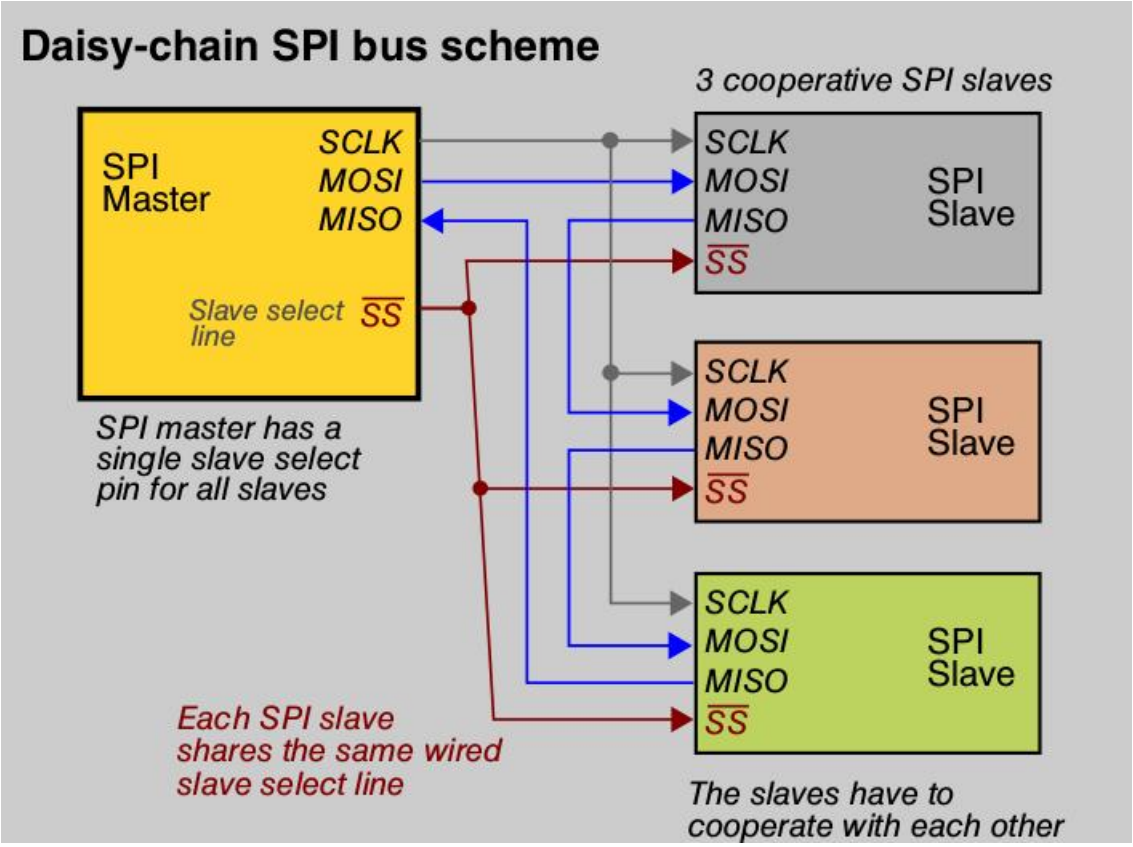

Figura 2.3. Linee di slave select condivise

Figura 2.3 dal Manuale del processore i.MX8QuadMax, Rev. E[10]

Come avviene la comunicazione

Il bus SPI sfrutta la tecnologia dei registri a scorrimento (shift registers). Tramite questi registri i dati sono trasmessi un bit ogni colpo di clock. Infine una volta terminato il trasferimento di una parola, questa è prelevata interamente. La parola può essere grande 8, 16 o 32 bit.

Sono disponibili alcuni parametri che regolano quando deve essere campionato il dato. Si tratta della polarità del clock (CPOL) e della fase (CPHA). La polarità indica il livello logico del clock quando si trova in stato di riposo, mentre la fase regola il fronte in cui viene campionato il segnale. Da questi due parametri e le loro combinazioni ne derivano quattro modalità di utilizzo, elencate di seguito. [9]

1. CPOL=0, CPHA=0
2. CPOL=1, CPHA=0
3. CPOL=1, CPHA=1
4. CPOL=0, CPHA=1

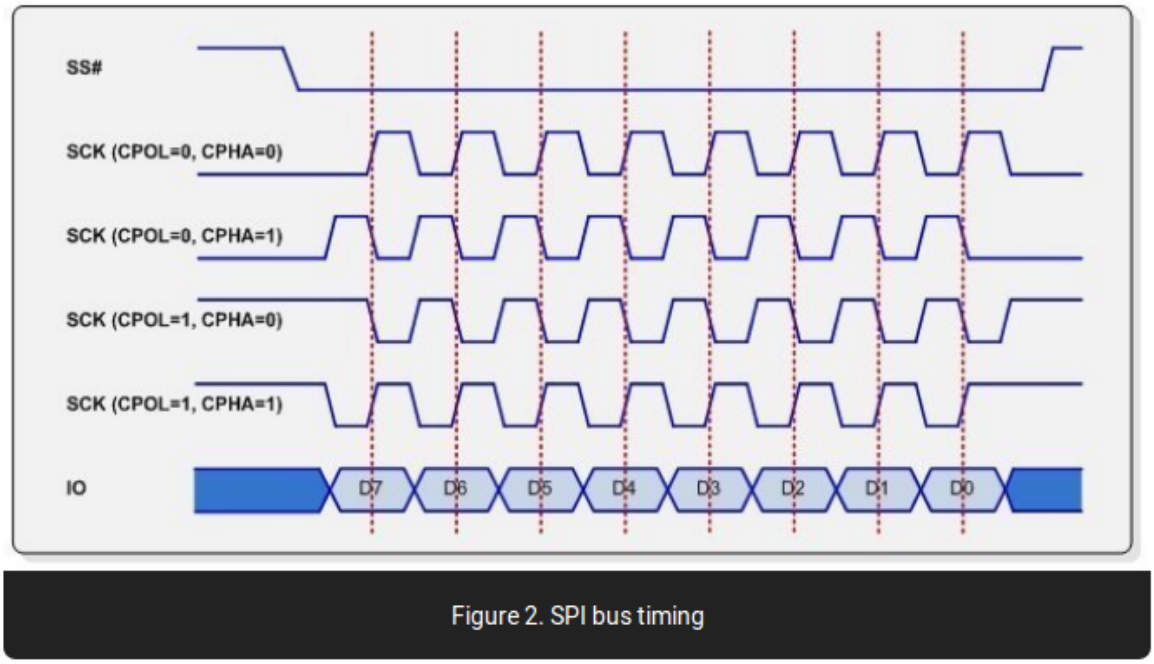

Figura 2.4. Polarità e fase del clock

Figura 2.4 da https://www.corelis.com/education/tutorials/spi-tutorial/[12]

Il disegno proposto mostra le differenze tra le varie combinazioni di questi parametri, e in particola mostra in quale istante il dato viene campionato.

In generale il master è progettato in modo da poter lavorare in uno qualsiasi di queste configurazioni, mentre lo slave è compatibile solo con una. Le configurazioni più usate sono: CPOL=0-CPHA=0, e CPOL=1-CPHA=1.

La comunicazione inizia quando il master attiva la linea CS (chip select - slave select) e il segnale di clock. Il master trasmette parole allo slave tramite la linea MOSI e contemporaneamente lo slave invia altrettante parole al master tramite la linea MISO. Il dispositivo slave riempire il registro con i dati da trasmettere prima che il master trasmetta i dati.

2.3 I device driver

Molti sistemi operativi, tra cui i sistemi Unix-like, sfruttano dei meccanismi di protezione che limitano le operazioni eseguibili dall'utente. Questo per evitare di apportare danni al sistema operativo e all'hardware sottostante, e per evitare che parti di codice vengano corrotte. Vengono distinte due modalità di permesso: la modalità kernel e la modalità user. Utilizzando il sistema in modalità kernel non si ha nessun tipo di restrizione, si può accedere a tutto il set di istruzioni del processore, tutte le aree di memoria e tutto lo spazio di I/O. La modalità utente non gode degli stessi privilegi: i processi utente utilizzano un limitato spazio di memoria riservato, non hanno accesso alle istruzioni del processore e non possono comunicare direttamente con i dispositivi. Per poter usufruire di qualcosa di cui non hanno accesso, i processi utente devono utilizzare le system calls e i device drivers. [11] Le prime sono un meccanismo con cui vengono richiesti dei servizi a livello kernel, tramite la chiamata di una funzione, in modo sicuro. Si tratta di un'interfaccia attraverso la quale si richiede un servizio offerto dal kernel, senza che l'applicazione utente conosca lo strato sottostante, e quindi senza che questo possa essere intaccato. Queste operazioni vengono svolte in modo sicuro, perchè le system call sono realizzate includendo dei meccanismi di protezione.

I secondi invece sono delle interfacce tramite cui è possibile accedere ai dispositivi, e di conseguenza interagire con l'hardware. I dispositivi disponibili, con cui le applicazioni utente possono interagire, sono elencati all'interno della cartella /dev, mentre le loro informazioni sono dei nodi nella cartella /sys.

Un device driver può essere parte del kernel Linux ed eseguito all'avvio del sistema operativo, oppure può essere un modulo kernel. I moduli kernel sono caricati successivamente, una volta che il sistema operativo è già stato avviato.

Gestione dell'interrupt

Gli interrupt sono segnali inviati dalle periferiche al processore per richiedere un determinato servizio. Ogni dispositivo ha un proprio numero identificativo di interrupt, chiamato interrupt request (IRQ). Quando il processore riceve un segnale di interrupt, esso controlla il numero IRQ in modo da identificare il dispositivo che ha scatenato la richiesta, ed esegue la corrispondente interrupt service routine (ISR, gestore di interrupt), ovvero una funzione atta

a soddisfare le richieste del dispositivo. Questa funzione viene attivata ogni volta che un dispositivo scatena un interrupt. [11]

Le ISR vengono eseguite nel contesto di interrupt, ovvero un ambiente in cui vengono eseguiti i gestori di interrupt. Le ISR, dal momento che devono essere eseguite all'istante e bloccando i processi attivi, devono essere molto veloci e non devono intaccare l'ambiente circostante. Per questo sono eseguite in un contesto diverso.

Ad ogni interrupt è associato un livello di priorità: un interrupt con priorità più alta viene eseguito prima e può interrompere un interrupt a priorità più bassa. La lista degli IRQ attivi nel sistema si può trovare in /proc/interrupts. [13]

2.3.1 I driver a caratteri

I driver sviluppati in questa tesi sono della tipologia a caratteri. Un dispositivo a caratteri rappresenta una periferica o un dispositivo virtuale su cui le operazioni di I/O sono effettuate parola per parola (con parole da 1 a 4 byte), in contrapposizione con i dispositivi a blocchi, nei quali i dati vengono trasferiti a blocchi di 64, 128 o più byte. [11]

Questi dispositivi sono identificati da due numeri: il major number e il minor number. Il primo è un numero intero che identifica univocamente il driver del dispositivo all'interno del kernel. Il secondo è un numero usato per identificare la singola istanza, tra tutti i dispositivi gestiti dallo stesso driver. Si può trovare un elenco dei device driver del kernel con i rispettivi major number con il comando cat /proc/devices.[13]

I driver a caratteri sono strutturati nel modo seguente: è presente una funzione di inizializzazione, (init()), dove vengono allocate le strutture globali da utilizzare, come:

- Strutture che identificano il dispositivo, come ad esempio una struct spi_driver che rappresenta un driver spi.
- Struct cdev, che rappresenta il driver a caratteri. Infatti registra il major number del dispositivo.
- Struct file, che contiene informazioni sul nodo associato in /dev.
- Struct file_operations, che contiene gli indirizzi dei metodi richiamabili dai programmi utente. Questi metodi vengono registrati in questa struttura proprio dalla funzione init.

Questa funzione viene chiamata quando il modulo viene inserito nel kernel.

Di seguito è riportato un esempio di funzione di inizializzazione per un modulo fittizio chiamato "modulo1": [13]

```
1  static dev_t dev_modulo1;
2  struct cdev cdev_modulo1;
3
4  struct file_operations file_modulo1 = {
5      .owner = THIS_MODULE,
6      .read = modulo1_read,
7      .ioctl = modulo1_func,
8  };
9
10 static int __init modulo1y_init(void)
11 {
12     alloc_chrdev_region(&dev_modulo1, 0, 1, "modulo1");
13     cdev_init(&cdev_modulo1, &file_modulo1);
14     cdev_modulo1.owner = THIS_MODULE;
15     cdev_add(&cdev_modulo1, dev_modulo1, 1);
16     return 0;
17 }
```

dev_t è una struttura dati che contiene il major number e il minor number del modulo, e quindi lo identifica univocamente. Viene utilizzato quando si crea il device file associato al modulo.

La struct cdev contiene un puntatore alla struct file_operation, contenente l'associazione tra i metodi richiamabili dall'utente e la loro implementazione, e anche il major number del modulo (rappresentato da dev) e il minor number (count).

Con alloc_chrdev_region si registra un range di numeri di device a caratteri. 0 è il primo minor number del modulo, 1 è la quantità di minor number da riservare a questo modulo, e modulo1 è il nome del modulo.

Con cdev_init viene inizializzata la struttura cdev, con la struct file_operation relativa.

Con cdev_add il device a caratteri è aggiunto al kernel Linux, con i relativi major e minor number.

Complementare a questa funzione è il metodo exit(), dove viene fatto l'inverso, quindi queste strutture sono distrutte e la memoria viene liberata. Exit è allo spegnimento del sistema.

```
1 static void __exit modulo1_cleanup(void)
2 {
3     cdev_del(&cdev_modulo1);
4     unregister_chrdev_region(dev_modulo1, 1);
5 }
```

Con cdev_del il device a caratteri è rimosso dal kernel Linux. Il range di major e minor number registrata con init() viene liberato tramite unregister_chrdev_region.

All'interno di un driver sono implementati anche metodi che possono essere chiamati dal programma utente per interagire con l'hardware. Questi metodi sono: open, close, read, write, ioctl, llseek. Possono essere implementati tutti oppure solo una parte. Possono anche essere implementati gestori di interrupt e altre funzioni di aiuto, che risulteranno invisibili alle applicazioni utente.

In particolare la routine ioctl (I/O control) è utilizzata per implementare le operazioni personalizzate, non richiamabili tramite read o write. Ad esempio, all'interno di un driver SPI, una tipica operazione effettuata da ioctl() è la modifica della velocità massima o della polarità del clock.

I dati in gioco in un'operazione di I/O possono essere trasferiti tramite la CPU, oppure tramite il DMA (Direct Memory Access), che effettua i trasferimenti lasciando libera la CPU. Questo sistema permette di velocizzare i trasferimenti di dati di grandi dimensioni.

2.4 Il processore NXP i.MX8 QuadMax

La scheda utilizzata per lo sviluppo della tesi è la i.MX8 QuadMax lpddr4 arm2, prodotta da NXP. I driver generati sono stati creati appositamente per essere utilizzati su questa scheda.

2.4.1 Caratteristiche generali

Il processore interno è il i.MX8QuadMax (i.MX8QM). Questo chip è stato pensato per i settori automobilistico ed industriale, e permette di raggiungere grandi performance e contemporaneamente basso consumo. Il sistema comprende un ARM Cortex A72 dual core, responsabile delle alte performance, e ARM Cortex A53 quad core, per i per i processi che intendono sfruttare una bassa potenza. La grafica è gestita da due Graphic Processing Units (GPUs) che supportano OpenVX per computer vision, e sono presenti due blocchi DRAM che supportano memorie di tipo DDR4 e LPDDR4. É presente un motore video dedicato che supporta fino a quattro display contemporaneamente. [10]

Altre caratteristiche del sistema sono: una System Control Unit (SCU) dedicata e un sottosistema di sicurezza che forniscono avvio sicuro e accelerazione crittografica, un sistema audio, due ARM Cortex M4 general purpose con le proprie periferiche, e molteplici periferiche molto utilizzate in ambito automobilistico. [10]

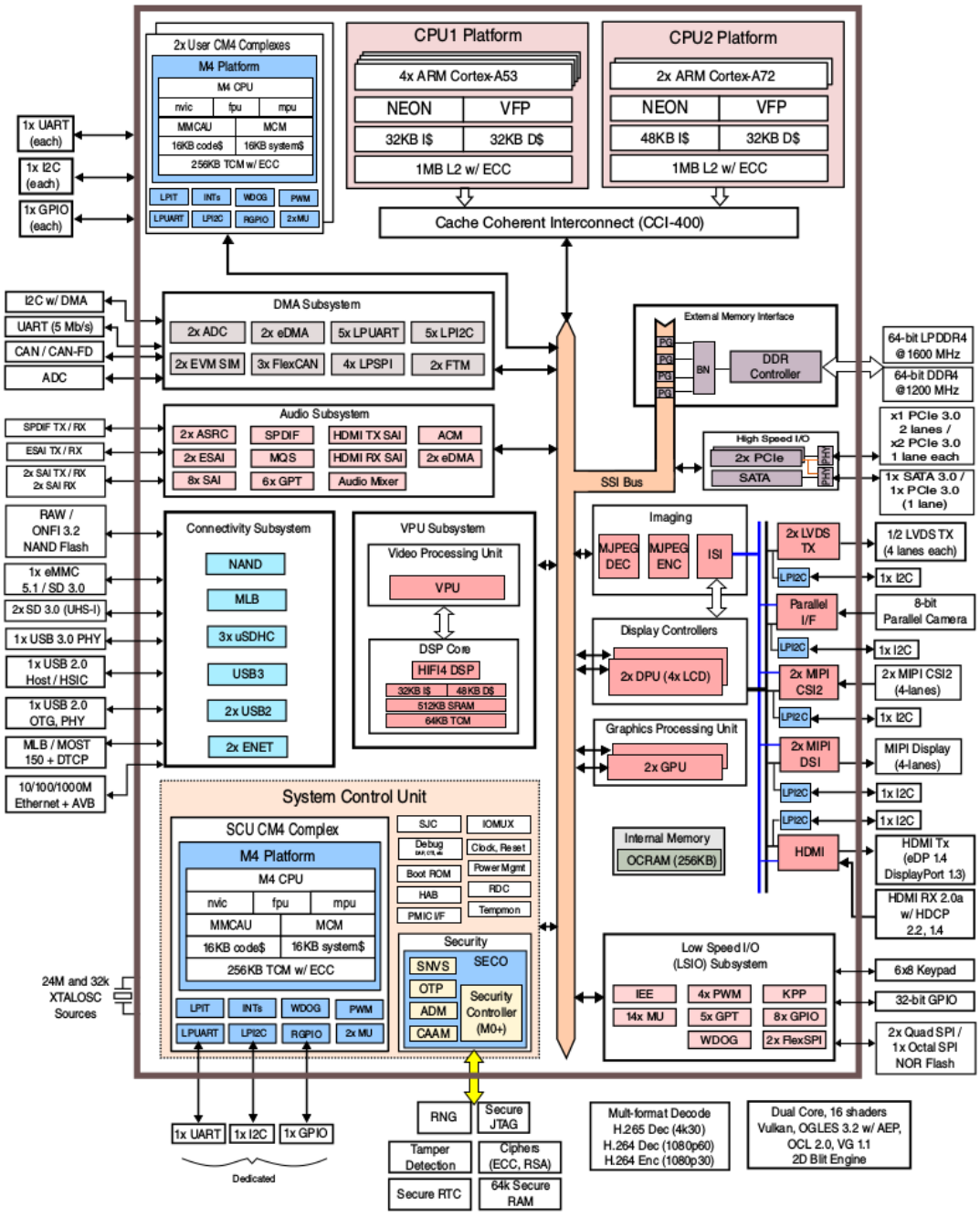

Figura 2.5. Diagramma a blocchi semplificato

Figura 2.5 dal Manuale del processore i.MX8QuadMax, Rev. E[10]

2.4.2 Low Power Serial Peripheral Interface

La scheda utilizza quattro moduli LPSPI (Low Power SPI), che supportano sia la modalità master che la modalità slave. LPSPI è una paricolare tipologia di SPI pensato per l'utilizzo in applicazioni che devono consumare poca potenza.

Ogni bus master può attivare un unico chip select alla volta. Sono possibili entrambe le modalità di collegamento con i dispositivi slave: le linee di slave select dedicate e la configurazione di tipo daisy chain, in cui più slave sono collegati allo stesso chip select. Questi moduli supportano inoltre l'accesso al DMA e possono generare DMA request. [10]

Le caratteristiche principali del modulo LPSPI sono:

- Dimensione massima delle parole = 32 bits
- Polarità e fase del clock configurabili
- Il master può essere collegato fino ad un massimo di quattro slave. Ha quindi a disposizione un massimo di quattro chip select
- É supportata anche la modalità slave
- FIFO di comando e di trasmissione di 64 parole
- FIFO di ricezione di 64 parole
- Frequenza di clock e ritardi tra PCS (Peripheral Chip Select) e fronti di clock flessibili in modalità master
- Supporto per trasferimento sia full duplex che half duplex
- Logica per data match in ricezione (confronto dei dati ricevuti con una parola o una parola mascherata) [10]

La tabella seguente mostra le tre tipologie di clock utilizzate dal sistema LPSPI, e la loro descrizione. [10]

Functional Clock	Il clock funzionale è asincrono rispetto al Bus Clock. Se questo clock è abilitato in modalità low power allora il modulo LPSPI può effettuare trasferimenti e risvegli, sia in modalità master che slave. Questo clock è diviso da un prescaler. La frequenza risultante deve essere almeno due volte più veloce rispetto alla frequenza del clock esterno.
External Clock	Il clock esterno controlla il registro a scorrimento di LPSPI. In modalità master esso è generato internamente, mentre in modalità slave il clock è fornito dall'esterno.
Bus Clock	Questo clock è utilizzato per l'accesso ai registri di controllo e configurazione di LPSPI. La frequenza deve essere compatibile con i requisiti di accesso ai registri e alle FIFO utilizzate.

Tabella 2.1. Le tipologie di clock

La tabella successiva mostra le varie tipologie di reset possibili, e le loro differenze: [10]

Chip reset	Riporta tutti i registri e la logica del modulo LPSPI allo stato predefinito.
Software reset	Riporta la logica e i registri del modulo LPSPI, tranne il registro di controllo, allo stato predefinito.
FIFO reset	Cancella il contenuto delle FIFO di ricezione e di trasmissione e comando.

Tabella 2.2. Le tipologie di reset

La tabella riportata di seguito indica la descrizione dei vari segnali coinvolti nei trasferimenti: [10]

Segnale	Nome	Descrizione
SCK	Serial Clock	Input in slave mode, output in master mode
PCS[0]	Peripheral Chip Select	Input in slave mode, output in master mode. Il master sceglie lo slave con cui comunicare e inizia i trasferimenti.
PCS[1] / HREQ	Peripheral Chip Select / Host Request	Input in slave mode o se il segnale è usato come host request, output altrimenti.
PCS[2] / DATA[2]	Peripheral Chip Select / secondo pin di dato in quad-data transfers	Input in slave mode o se si ricevono dati in quad-data transfers. Output in master mode o se si trasmettono dati in quad-data transfers.
PCS[3] / DATA[3]	Peripheral Chip Select / secondo pin di dato in quad-data transfers	Input in slave mode o se si ricevono dati in quad-data transfers. Output in master mode o se si trasmettono dati in quad-data transfers.
SOUT / DATA[0]	Serial Data Output	Può essere configurato anche come serial data input (ad esempio in slave mode). É usato come data pin 0 in quad-data transfers e in dual-data transfers.
SIN / DATA[1]	Serial Data Output	Può essere configurato anche come serial data output (ad esempio in slave mode). É usato come data pin 1 in quad-data transfers e in dual-data transfers.

Tabella 2.3. Descrizione dei segnali

LPSPI: i registri

I moduli LPSPI sfruttano una serie di registri, ogniuno con una funzione specifica. Questi registri sono utilizzati sia dai moduli master che dagli slave, ad eccezione del registro che configura il clock, che è sfruttato solo dai moduli master.

Un registro svolge una funzione particolare, ovvero il Transmit Command Register (TCR). Questo registro contiene una parola di comando, in cui sono descritti tutti i parametri che configurano un trasferimento. Questo registro deve essere riempito con la parola di comando prima di ogni trasferimento. In modalità slave, è importante che la parola venga scritta quando il modulo

è disabilitato, altrimenti si scatena un errore e il trasferimento non avviene come programmato.

Di seguito è riportata una descrizione dei vari registri che costituiscono la logica dei moduli LPSPI della scheda, e di alcuni tra i loro campi più importanti. [10]

- Version ID Register (VERID): Contiene major number, minor number e numero di identificazione del modulo.
- Parameter Register (PARAM): Identifica la capienza della coda di ricezione e di trasmissione, ovvero il numero massimo di parole che le code possono contenere.
- Control Register (CR): Questo registro permette di controllare come viene utilizzato il modulo lpspi. Infatti contiene un campo denominato "module enable", che abilita o disabilita il dispositivo. Altri campi importanti sono:
  - "software reset": riporta tutti i registri (ad eccezione di questo) allo stato originale. É utile per elimare tutti i dati ricevuti che non sono stati eliminati durante un reset della coda di ricezione.
  - "debug enable": il dispositivo è abilitato in modalità debug.
  - "reset txfifo" e "reset rxfifo": questi bit cancellano il contenuto di uno dei due registri.
- Status Register (SR): descrive i vari flag di stato. Questi flag, se di valore 1, possono scatenare richieste di interrupt e di dma. Tra i vari flag sono menzionati:
  - MBF: Il modulo è occupato (busy).
  - TDF: La coda di trasmissione richiede dati da inserire.
  - RDF: La coda di ricezione è pronta per essere letta.
  - TCF: Il trasferimento è stato completato.
  - TEF: Si è verificato un errore di trasmissione.
- Interrupt Enable Register (IER): abilita o disabilita i vari interrupt che il modulo può scatenare. Questi interrupt dipendono dal contenuto del registro dello stato. Ad esempio il flag RDF del registro dello stato scatena un interrupt se è abilitato il flag RDIE del registro di Interrupt. Tra i vari flag presenti:

  - TDIE: Il flag TDF del registro dello stato ha valore 1.
  - TCF: Il flag TCF del registro dello stato ha valore 1.

- Configuration Register 0 (CFGR0): É il primo dei due registri dedicati alla configurazione dei trasferimenti. Tra i vari campi:
  - "circular fifo": abilita l'utilizzo di una fifo circolare.
  - "HRSEL": Host Request Select
  - "HRPOL": polarità del pin Host Request.
  - "HREN": abilita Host Request.

- Configuration Register 1 (CFGR1): è il secondo registro di configurazione e può modificato solo quando il modulo è disabilitato.
  - "master": identifica il dispositivo come master o come slave.
  - "PCSPOL": la polarità per ciascun chip select.
  - "PINCFG": la configurazione deipin SOUT e SIN:
    * 00b: SIN ha direzione input, SOUT ha direzione output. É la configurazione base per la modalità master.
    * 01b: SIN è usato sia come input che come output.
    * 10b: SOUT è usato sia come input che come output.
    * 11b: SOUT ha direzione input, SIN ha direzione output. É la configurazione base per la modalità slave.
  - "peripheral chip selct configuration": usato in trasferimenti di quattro bit per colpo di clock.

- Data Match Register 0 (DMR0) e Data Match Register 1 (DMR1): questi registri vengono utilizzati se è abilitata la modalità data match. I dati ricevuti vengono confrontati con i valori di questi registri.

- Clock Configuration Register (CCR): è utilizzato solo in modalità master, e non può essere modificato quando il modulo è abilitato. Tra i campi è presente il valore di clock divider, che è utilizzato per dividere il valore del SCK, un campo di ritardo tra i trasferimenti, e altri.

- FIFO Control Register (FCR): il registro di controllo delle fifo di trasmissione e di ricezione. Contiene il volore dei watermark di trasmissione e ricezione, ovvero indica quanti byte sono necessari per far scattare i flag RDF e TDF del registro di stato e scatenare gli interrupt.

- FIFO Status Register (FSR): il registro dello stato delle fifo. Contiene il numero di parole attualmente presenti nella fifo di ricezione e in quella di trasmissione.
- Transmit Command Register (TCR): identifica i parametri con cui configurare un trasferimento. Un trasferimento non può iniziare se non è stata scritta una nuova parola di comando in questo registro. In modalità slave, il registro TCR può essere modificato solo se il modulo è disattivato. Di seguito sono elencati i vari campi che compongono questo registro.
  - "framesz": il numero di bit che compongono un frame.
  - "width": il numero di bit trasferiti per colpo di clock (1, 2 o 4).
  - "CONTC": Continuing Command. In modalità master, permette alla parola di comando di essere cambiata durante un trasferimento continuo.
  - "CONT": trasferimento continuo.
  - "byte swap": scambia il contenuto dei bit [31:24] con i it [7:0] e i bit [23:16] con [15:8] per ogni parola.
  - polarità e fase di clock.
  - "pcs": il chip select scelto per la tramissione.
- Transmit Data Register (TDR): Contiene i dati da che devono essere trasmessi. Se si mandano parole di 8 o 16 bit ciascuno, il resto del registro è imposto a zero.
- Receive Status Register (RSR): il registro dello stato della coda di ricezione. Sono presenti due campi importanti:
  - "start of frame": indica se questa è la prima parola ricevuta dopo l'attivazione del chip select.
  - "rxempty": indica se la fifo di ricezione è vuota.
- Receive Data Register (RDR): Contiene i dati ricevuti.

Capitolo 3

Il lavoro svolto

3.1 La configurazione di Yocto

Per poter configurare Yocto per la macchina di destinazione è stata seguita la guida fornita da Yocto stesso, per la versione Morty L4.9.51_imx8qm-beta1. [8]

Download dei file

Per ottenere tutti i file necessari per lo sviluppo è stato utilizzato lo strumento Repo, che sfrutta Git per raccogliere i dati e scaricarli.

La cartella scaricata contiene varie sottocartelle, tra cui una denominata sources, dove sono presenti le ricette per creare i progetti, ed alcuni script per configurare l'ambiente. Queste ricette sono state realizzate da NXP e dalla community di Yocto. In questa cartella sources vengono scaricati i vari layer. Per poter utilizzare Yocto e lavorare sulla scheda scelta, le istruzioni necessarie sono le seguenti: [8]

```
1 mkdir imx-yocto-bsp
2 cd imx-yocto-bsp
3 repo init -u https://source.codeaurora.org/external/imx/imx-
    manifest -b imx-linux-morty -m imx-4.9.51-8qm_beta2.xml (
4 repo sync
```

Come primo comando viene crea una cartella di partenza, imx-yocto-bsp, che conterrà l' intero progetto. Successivamente si indica il sito da dove recuperare i dati, e la versione di Yocto scelta, Morty, che sfrutta il kernel 4.9.51. Mediante l'ultimo comando il codice è scaricato nella cartella creata.

Creazione dell'immagine

Nella cartella appena scaricata è presente uno script chiamato fsl-setup-release.sh che crea una directory e configura l'ambiente in modo da essere compatibile con la macchina scelta.

Nel layer meta-fsl-bsp-release sono presenti dei file di configurazione specifici per ogni scheda. Lo script fsl-setup-release.sh copia questi file in meta-freescale/conf/machine così da sovrascrivere la configurazione predefinita. Tra le varie configurazioni disponibili, quella scelta nel corso di questo progetto è denominata "imx8qmlpddr4arm2" e coincide con la scheda utilizzata. [8]

La distro scelta è stata quella denominata "fsl-imx-x11". Questa distro sfrutta il gestore grafico X Window System (x11).

Le configurazioni delle distro sono specificate nel file local.conf. Gli sviluppatori possono creare le loro distro, in cui possono personalizzare l'ambiente scegliendo le versioni e i provider.

La sintassi del comando per lanciare lo script è: [8]

```
1 DISTRO=<distro name> MACHINE=<machine name> source fsl-setup
    -release.sh -b <build dir>
```

<build dir> specifica la cartella da creare e dove salvare l'immagine generata. Per questo progetto il nome scelto per questa cartella è "build-imx".

Una volta terminato lo script, all'interno della cartella generata è presente una sottocartella conf, contenente i file local.conf e bblayers.conf, di cui è stato parlando in precedenza, nel capitolo 2.1.2.

La tabella seguente, tratta dalla guida di Yocto Morty [8], elenca alcune tra le possibili immmagini che si possono generare tramite Bitbake.

Immagine	Obiettivo	Fornita dal layer
core-image-minimal	Piccola immagine che permette l'avvio del sistema.	Poky
core-image-base	Immagine che fornisce supporto completo per il dispositivo scelto	Poky
core-image-sato	Un'immagine che sfrutta Sato, un'interfaccia grafica per sistemi mobili, contentente un terminale, un editor e un gestore di file.	Poky
fsl-image-machine-test	Immagine fornita dalla community di i.MX	meta-freescale-distro
fsl-image-validation-imx	Un'immagine con interfaccia grafica, specifica per i.MX.	meta-fsl-bsp-release/imx/meta-sdk
fsl-image-qt5-validation-imx	Un'immagine Qt5 open source, supportata solo da sistemi i.MX provvisti di interfaccia grafica.	meta-fsl-bsp-release/imx/meta-sdk

Tabella 3.1. Descrizione dei segnali

Per eseguire una build si utilizza bitbake, scegliendo l'immagine da generare. Per il progetto è stata utilizzata un'immagine di tipo "core-image-minimal", e il comando relativo è:

```
1 bitbake core-image-minimal
```

con il quale è stata creata la più piccola immagine disponibile.

Spostamento su sd card

Una volta che l'immagine è stata creata, essa si trova in build-imx/tmp/deploy /images. L'immagine è generalmente generata in più formati. Quello utilizzato per questo progetto è il formato .sdcard.

Con questo comando è possibile installare l'immagine in una scheda sd. [8]

```
1 sudo dd if=<image name>.sdcard of=/dev/sd<partition> bs=1M
    && sync
```

Una volta che la scheda è collegata alla corrente ed il pulsante di accensione è premuto, il sistema operativo si avvierà.

La generazione della SDK

Una volta che la distribuzione Linux è pronta è possibile creare la il suo Software Development Kit, costituito da un cross-compilatore per la macchina specificata e le librerie presenti nella distribuzione scelta. Il comando usato per creare la SDK per il progetto è:

```
1 bitbake core-image-minimal -c populate_sdk
```

Questo comando crea un file necessario per installare il cross-compilatore. Ogni volta che si intende utilizzare il cross-compilatore bisogna preparare l'ambiente, richiamando lo script generato dall'installazione.

La personalizzazione dell'immagine

La figura seguente schematizza in modo semplificato il processo di modifica del kernel e il suo aggiornamento.

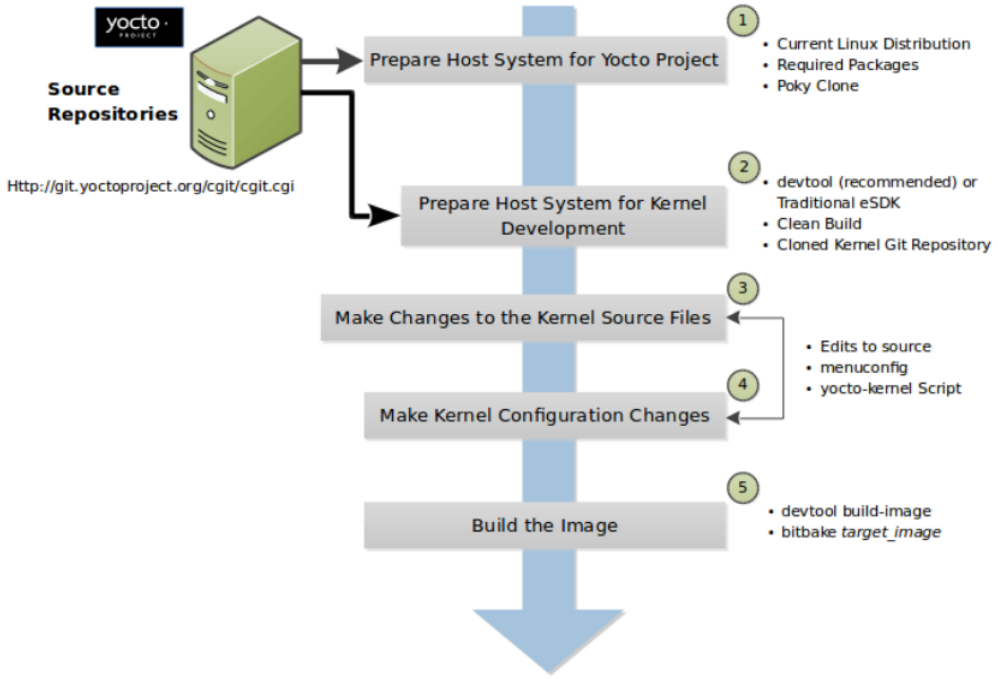

Figura 3.1. Il processo di modifica e applicazione delle modifiche al kernel Linux con Yocto

Figura 3.1 dal Manuale di sviluppo del kernel con Yocto[14]

Per personalizzare la configurazione del kernel, si può utilizzare il comando

```
1 bitbake -c menuconfig virtual/kernel
```

Una volta eseguito, viene aperta un'interfaccia grafica da cui si possono scegliere varie opzioni. Scegliendo il campo "Device Drivers" si ottiene un elenco di device supportabili. Scegliendo tra questi il campo "SPI Support" si abilita il supporto per SPI nella scheda selezionando il campo "Freescale i.MX LPSPI controller".

Il passo successivo per abilitare il supporto SPI è l'inserimento di moduli nel device tree che utilizzano il protocollo. Il device tree è una struttura dati che descrive i componenti hardware di un dispositivo. Ogni nodo dell'albero corrisponde ad un dispositivo. I file che descrivono il device tree hanno estensione .dts (device tree source). Questi file una volta compilati dal device tree compiler, si trasformano in file di estensione .dtb (device tree blob), che saranno poi letti dal boot loader. I file .dts devono essere modificati adeguatamente per abilitare il supporto per il protocollo SPI. Una volta fatte queste modifiche, esse vengono applicate ripetendo il comando bitbake. Il lavoro sul device tree è approfondito nela capitolo 3.2.1

Per includere nel progetto i nuovi driver è stato aggiunto un nuovo layer. Per aggiungere un layer di nome mata-new si utilizza il comando (dalla cartella build-imx): [15]

```
1 yocto-layer create ../sources/meta-new
```

É stato creato un layer la cui cartella è situata in sources/meta-new. Per utilizzare il layer nuovo bisogna includere il suo percorso nel file build-imx/conf/bblayers.conf.

Per aggiungere i nuovi driver SPI, uno per il protocollo master e uno per il protocollo slave, per prima cosa bisogna creare delle nuove cartelle. Avendo creato un nuovo layer di nome meta-new, le cartelle sono state create in sources/meta-new/recipes-kernel/. Viene di seguito riportato il procedimento per l'inserimento del modulo master.

Per il modulo master (di nome spi-master) è stata creata una cartella di nome spi-module in sources/meta-new/recipes-kernel/. All'interno della cartella si aggiungono una seconda cartella, di nome files, ed un file chiamato spi-module.bb, contenente

```
DESCRIPTION = "Linux spi master kernel module"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=(...)"

inherit module

COMPATIBLE_MACHINE = "imx8qmlpddr4arm2"

PV = "1.0"

SRC_URI = "file://Makefile \
           file://spi-module.c \
           file://spi-module.h \
           file://COPYING \
           "
S = "${WORKDIR}"
```

In questo file sono indicati la licenza, la checksum, la macchina di destinazione, la versione del driver e l'elenco dei file presenti. All'interno della cartella files sono creati un file spi-module.c, contenente il codice del driver, un file header, spi-module.h, un Makefile per compilare e rendere eseguibile il codice, e un file COPYING, che identifica la licenza del driver.

Una volta creati questi file, si inserisce il modulo nel sistema inserendo le seguenti righe nel file /build-imx/local.conf:

```
MACHINE_ESSENTIAL_RRECOMMENDS += "kernel-modules spi-module"
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-modules spi-
    module"
KERNEL_MODULE_AUTOLOAD += "spi-module"


MACHINE_ESSENTIAL_RRECOMMENDS += "kernel-modules spi-slave"
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-modules spi-
    slave"
KERNEL_MODULE_AUTOLOAD += "spi-slave"
```

In questo esempio è stato inserito anche il modulo relativo allo slave.

La cartella workspace e Devtool

Per poter effettuare modifiche al kernel esistente è possibile utilizzare uno strumento chiamato Devtool. Esso permette di creare una cartella chiamata Workspace in cui è presente una copia del kernel usato da Yocto. É possibile modificare i file presenti nella cartella e una volta ripetuto il comando

```
1 bitbake core-image-minimal
```

è possibile visualizzare gli effetti delle modifiche. Nel corso di questo progetto questo strumento è stato molto utile, dal momento che la versione del Linux kernel utilizzata (la versione 4.9.51) non include un supporto per SPI slave ed è stato necessario includerlo manualmente modificando alcuni file. Una volta terminate le modifiche, Devtool permette di creare una una patch per applicarle al sistema originale (capitolo 3.6).

Per creare la cartella Workspace è stato utilizzato il comando

```
1 devtool modify linux-imx
```

linux-imx è il nome del kernel in uso. Dopo di che la cartella è stata aggiunta all'elenco dei layer nel file conf/bblayers.conf, per includere la suddetta cartella tra i layer da considerare durante la build.

3.2 Aggiunta del supporto per SPI slave

Il kernel 4.9 non contiene supporto per SPI in modalità slave, ma solo per la modalità master. Il supporto per dispositivi slave verrà introdotto solo a partire dal kernel 4.13. La versione di Yocto utilizzata per questo progetto è Morty, che sfrutta il kernel 4.9.51; è stato quindi necessario apportare delle modifiche per poter includere il supporto per i dispositivi slave, prendendo come riferimento un kernel più recente, ovvero la versione 4.14.62.

Questo processo ha portato alla modifica prima di tutto del driver che gestisce i moduli lpspi (chiamato fsl_lpspi.c), i file spi.c e spi.h, e i file che descrivono il device tree. Le modifiche effettuate consistono nel aggiornare i codici con quelli della versione del kernel 4.14.62. [16] [17] [18] Il file spi-fsl-lpspi.c è stato poi ulteriormente modificato, per renderlo compatibile con l'hardware di destinazione.

3.2.1 Il device tree

In origine nei file di device tree si potevano includere solo dispositivi spi master. Il file fsl-imx8qm.dtsi contiene le definizioni dei moduli presenti nel sistema, con le loro caratteristiche, ad esempio le linee di interrupt utilizzate, i clock, la gestione della potenza.

Di seguito è riportata la descrizione del modulo spi master nel file fsl-imx8qm.dtsi. Il modulo scelto è il modulo lpspi3. [19]

```
lpspi3: lpspi@5a030000 {
        compatible = "fsl,imx7ulp-spi";
        reg = <0x0 0x5a030000 0x0 0x10000>;
        interrupts = <GIC_SPI 219 IRQ_TYPE_LEVEL_HIGH>;
        interrupt-parent = <&gic>;
        clocks = <&clk IMX8QM_SPI3_CLK>,
                 <&clk IMX8QM_SPI3_IPG_CLK>;
        clock-names = "per", "ipg";
        assigned-clocks = <&clk IMX8QM_SPI3_CLK>;
        assigned-clock-rates = <20000000>;
        power-domains = <&pd_dma_lpspi3>;
        status = "disabled";
};
```

5a03000 è l'indirizzo (in esadecimale) a cui è mappato il modulo lpspi3. l'etichetta compatible indica quale driver sarà associato al dispositivo. Lo stato di default è "disabilitato". Nel file fsl-imx8qm-lpddr4-arm2.dts lo stato è sovrascritto con "okay", così da abilitare solo i moduli necessari.

Il file fsl-imx8qm-lpddr4-arm2.dts include il file fsl-imx8qm.dtsi e ne personalizza i componenti. Di norma non contiene definizioni dei componenti LPSPI. Esempi dei inclusione dei suddetti si trovano nel file fsl-imx8qm-lpddr4-arm2-lpspi.dts. Di seguito è riportata la sezione del file relativa al modulo master lpspi3. [20]

```
&lpspi3 {
        #address-cells = <1>;
        #size-cells = <0>;
        fsl,spi-num-chipselects = <1>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpspi3 &pinctrl_lpspi3_cs>;
        cs-gpios = <&gpio2 21 GPIO_ACTIVE_LOW>;
        status = "okay";
```

```
9
10          spi0: spimaster@0 {
11                  spi-max-frequency = <0x7a120>;
12                  reg = <0x0>;
13                  compatible = "linux,spimodule";
14          };
15  };
16
17  pinctrl_lpspi3: lpspi3grp {
18          fsl,pins = <
19                  SC_P_SPI3_SCK_DMA_SPI3_SCK                  0
                        x0600004c
20                  SC_P_SPI3_SDO_DMA_SPI3_SDO                  0
                        x0600004c
21                  SC_P_SPI3_SDI_DMA_SPI3_SDI                  0
                        x0600004c
22          >;
23  };
24
25  pinctrl_lpspi3_cs: lpspi3cs {
26          fsl,pins = <
27                  SC_P_SPI3_CS1_LSIO_GPIO2_IO21               0x21
28          >;
29  };
```

I moduli utilizzati vengono abilitati, sovrascrivendo il loro stato (da "disabled" a "okay"). Il modulo fa riferimento alla definizione dei pin utilizzati. pinctrl_lpspi3 è il gruppo di pin che definiscono il clock, il SDO e l'SDI, mentre pinctrl_lpspi3_cs è il pin corrispondente al chip select.

In modalità master, SDO è utilizzato come output, e SDI come input, mentre in modalità slave i due pin sono utilizzati al contrario. Le definizioni di questi pin sono contenute nel file /imx/build-imx/workspace/sources/linux-imx/include/dt-bindings/pinctrl/pads-imx8qm.h, mentre il numero a destra (0x0600004c nel caso di SDI, SDO e clock, 0x21 nel caso di chip select) corrisponde ai bit da scrivere nel registro corrispondente.

I pin posso essere descritti anche in un altro modo.

```
1  pinctrl_lpspi3: lpspi3grp {
2          fsl,pins = <0x98 0x0 0x600004c 0x99 0x0 0x600004c 0
               x100 0x0 0x600004c>;
3  };
```

```
4
5  pinctrl_lpspi3_cs: lpspi3cs {
6          fsl,pins = <0x102 0x3 0x21>;
7  };
```

Considerando il file pads-imx8qm.h, "SC_P_SPI3_SCK_DMA_SPI3_SCK 0x0600004c" può essere riscritto come 0x98 0x0 0x600004c. 0x98 corrisponde al pad SC_P_SPI3_SCK, 0x0 specifica che il pad è configurato per l'utilizzo di dma (0x3 corrisponde a SC_P_SPI3_SCK_LSIO_GPIO2_IO17, quindi il pad sarebbe usato come gpio), 0x60004c è nuovamente il valore da scrivere nel registro. Seguono sulla stessa riga i valori per SDO e SDI. Infine la definizione del pad del chip select.

Diversa è invece la definizione del dispositivo in modalità slave nel file fsl-imx8qm-lpddr4-arm2.dts. Come slave è stato scelto il modulo lpspi0. Di seguito è riportata la definizione del modulo nel file fsl-imx8qm.dtsi: è equivalente alla definizione del modulo master. [21]

```
1  lpspi0: lpspi@5a000000 {
2          compatible = "fsl,imx7ulp-spi";
3          reg = <0x0 0x5a000000 0x0 0x10000>;
4          interrupts = <GIC_SPI 216 IRQ_TYPE_LEVEL_HIGH>;
5          interrupt-parent = <&gic>;
6          clocks = <&clk IMX8QM_SPI0_CLK>,
7                   <&clk IMX8QM_SPI0_IPG_CLK>;
8          clock-names = "per", "ipg";
9          assigned-clocks = <&clk IMX8QM_SPI0_CLK>;
10         assigned-clock-rates = <20000000>;
11         power-domains = <&pd_dma_lpspi0>;
12         status = "disabled";
13 };
```

Di seguito invece è presente il frammento del file fsl-imx8qm-lpddr4-arm2.dts relativo al dispositivo slave.

```
1  &lpspi0 {
2
3          pinctrl-names = "default";
4          pinctrl-0 = <&pinctrl_lpspi0 &pinctrl_lpspi0_cs>;
5          status = "okay";
6          spi-slave;
```

```
7
8           slave {
9                   compatible = "linux,spislave";
10          };
11  };
12
13  pinctrl_lpspi0: lpspi0grp {
14          fsl,pins = <0x75 0x0 0x600004c 0x76 0x0 0x600004c 0
                x77 0x0 0x600004c>;
15  };
16
17  pinctrl_lpspi0_cs: lpspi0cs {
18          fsl,pins = <0x79 0x3 0x21>;
19  };
```

Il modulo lpspi0 contiene l'etichetta spi-slave, che lo identifica come dispositivo slave. Questa etichetta è stata introdotta a partire dal kernel 4.13, dal momento che precedentemente non vi era un supporto per slave. Le etichette #address-cells, #size-cells, fsl,spi-num-chipselects, cs-gpsios non sono più necessarie, resta solo la definizione dei pin. Il nodo figlio è chiamato slave, ed al suo interno è presente solo l'etichetta compatible, che associa il modulo con il driver spi-slave. Il driver spi-slave creato contiene al suo interno l'etichetta compatible = "linux,spislave", in modo da poterlo legare al dispositivo lpspi0.

3.2.2 Interfaccia con il bus

I file spi.c e spi.h contengono dei metodi utili per la registrazione dei driver di tipo spi, definiscono un bus, e fanno da congiunzione tra i driver a caratteri implementati nel progetto (spi-master e spi-slave) e il driver che comunica con l'hardware (fsl-lpspi). Permettono quindi di richiamare le funzionalità di fsl-lpspi senza doverlo conoscere e senza sapere cosa succede al suo interno.

Per poter includere un supporto per i dispositivi in modalità slave, sono stati introdotti dei nuovi metodi e modificati degli altri. I nuovi metodi introdotti sono:

- spi_alloc_slave: alloca un controllore spi di tipo slave.
- spi_controller_is_slave: identifica se il dispositivo in questione è master o slave.

- spi_slave_abort: richiama la funzione fsl_lpspi_slave_abort del driver fsl-lpspi.

Sono inoltre definite diverse strutture di supporto all'inizializzazione e alla configurazione dei trasferimenti, e alla registrazione dei dispositivi.

Il file spi.c definisce una tipologia di bus chiamata "spi". Questa struttura astrae il concetto di bus SPI. Il bus viene registrato durante la funzione di inizializzazione chiamata all'avvio. In questa funzione sono anche registrate due classi, spi_master_class e spi_slave_class (quest'ultima introdotta dal kernel 4.13). Le classi sono delle astrazioni delle tipologie di dispositivi disponibili. Quando viene allocato un controllore spi, ad esso è associata la relativa classe.

3.2.3 Inizializzazione del modulo LPSPI

fsl-lpspi è un driver di tipo platform, ovvero una tipologia di driver trasparente al software soprastante. Non possono essere richiamati in modalità utente. Questa tipologia di driver viene registrata nel sistema tramite una chiamata al metodo module_platform_driver, che al suo interno chiama il metodo platform_driver_register, che richiama driver_register. Quest'ultimo registra un driver con il bus corrispondente, e chiama una funzione, probe, definita all'interno del driver da registrare.

Questo modulo definisce una struttura contenente tutte le informazioni utili per la configurazione del dispositivo e dei trasferimenti. Questa struttura è chiamata fsl_lpspi_data, e al suo interno è presente anche una seconda struttura che identifica precisamente i parametri del trasferimento. Le due strutture sono così composte: [18]

```
1 struct fsl_lpspi_data {
2         struct device *dev;
3         void __iomem *base;//l'indirizzo di partenza del
              modulo
4         struct clk *clk_ipg;//il clock "ipg"
5         struct clk *clk_per;//il clock "per"
6
7         bool is_slave;//se il modulo    in modalit  master
              o slave
8
9         void *rx_buf;//contiene i dati ricevuti
```

```
10          const void *tx_buf;//contiene i dati da inviare
11          void (*tx)(struct fsl_lpspi_data *);//metodo per
                inviare i dati
12          void (*rx)(struct fsl_lpspi_data *);//metodo per
                ricevere i dati
13
14          u32 remain;//indica quante parole devono ancora
                essere trasferite
15          u8 watermark;//il numero di parole che compongono un
                 trasferimento
16          u8 txfifosize;//dimensione massima della coda di
                trasmissione
17          u8 rxfifosize;//dimensione massima della coda di
                ricezione
18
19          struct lpspi_config config;
20          struct completion xfer_done;//per attendere la fine
                di un trasferimento
21
22          bool slave_aborted//se il modulo    slave, indica se
                      stato disabilitato
23
24          int chipselect[0];
25  };
26
27  struct lpspi_config {
28          u8 bpw;//bits per word
29          u8 chip_select;
30          u8 prescale;//parametro che divide il periodo di
                clock
31          u16 mode;//modo di funzionamento: polarit  e fase
                del clock, chip select attivo alto o basso
32          u32 speed_hz;//velocit  massima del modulo (in
                hertz)
33  };
```

La funzione probe del suddetto driver identifica se il modulo è utilizzato in modalità master o slave, configura eventualmente i chip select, alloca la memoria necessaria e assegna i metodi da invocare nel corso di alcune operazioni, abilita interrupt e clock, e registra il dispositivo tra i moduli del kernel.

Questo metodo è stato aggiornato alla versione 4.14.62 del kernel. Di

seguito sono riportate alcune parti del metodo probe, con breve spiegazione del codice. [18]

Viene fatta una distinzione tra dispositivo master e slave, leggendo la proprietà "spi-slave" nel file device tree.

```
1 if (of_property_read_bool((&pdev->dev)->of_node, "spi-slave"
    ))
2                 controller = spi_alloc_slave(&pdev->dev,
3                                              sizeof(struct
                                                 fsl_lpspi_data));
4         else
5                 controller = spi_alloc_master(&pdev->dev,
6                                              sizeof(struct
                                                 fsl_lpspi_data));
```

Qualora il dispositivo sia di tipo slave, viene invocato il metodo spi_alloc_slave, altrimenti verrà invocato spi_alloc_master. [16]

spi_alloc_slave è definita in spi.h, ed al suo interno richiama ___spi_alloc_master, passandole come terzo parametro un bool di valore true, indicando così che si tratta di un dispositivo slave.

Anche spi_alloc_master richiama ___spi_alloc_master, passando come terzo parametro false. [16]

Viene allocata una nuova struttura di tipo spi_master, e ne si inizializza il dispositivo. Viene assegnata la classe relativa, a seconda che il dispositivo sia slave o master.

Continuando con la funzione probe, se il dispositivo è un master si passa all'abilitazione dei chipselect. Per ogni chipselect, si legge la proprietà "cs-gpios" dal device tree e si controlla se il gpio relativo è valido. Il gpio restituito è assegnato alla struct fsl_lpspi_data. Vengono poi assegnati puntatori a funzione per prepare_message e unprepare_message, metodi che sono richiamati poi in ___spi_pump_messages e spi_finalize_current_message (sezione 3.2.4), ma solo se il dispositivo è di tipo master. Queste operazioni servono per inizializzare i gpio dei chipselect del master. Il primo metodo inizializza il gpio come output e gli assegna valore attivo, mentre il secondo gli assegna un valore disattivato. Il valore che indica se il chipselect è attivo dipende dalla proprietà SPI_CS_HIGH. Se questa proprietà ha valore 1 allora il chip select quando è attivo ha valore alto. Solitamente viene utilizzato il valore basso.

Una volta terminate queste operazioni, si prosegue configurando altri metodi per il controllore.

```
1     controller->transfer_one_message =
          fsl_lpspi_transfer_one_msg;
2         controller->prepare_transfer_hardware =
              fsl_lpspi_prepare_xfer_hardware;
3         controller->unprepare_transfer_hardware =
              fsl_lpspi_unprepare_xfer_hardware;
4     controller->slave_abort = fsl_lpspi_slave_abort;
```

Questi quattro metodi saranno poi utilizzati durante i trasferimenti, ad eccezione di slave_abort, che verrà richiamato qualora si voglia disattivare il modulo slave. Questo metodo setta il flag slave_aborted della struttura fsl_lpspi_data e nel caso in cui il modulo fosse in attesa della fine di un trasferimento, sveglia sveglia la wait_queue. [18]

```
1 static int fsl_lpspi_slave_abort(struct spi_controller *
      controller)
2 {
3         struct fsl_lpspi_data *fsl_lpspi =
4                                     spi_controller_get_devdata(
                                          controller);
5
6         fsl_lpspi->slave_aborted = true;
7         complete(&fsl_lpspi->xfer_done);
8         return 0;
9 }
```

Si configurano anche altri parametri che definiscono i modi di utilizzo del controllore:

```
1         controller->mode_bits = SPI_CPOL | SPI_CPHA |
              SPI_CS_HIGH;
2         controller->flags = SPI_MASTER_MUST_RX |
              SPI_MASTER_MUST_TX;
3         controller->dev.of_node = pdev->dev.of_node;
4         controller->bus_num = pdev->id;
```

In particolare si assegna al controllore il nodo del modulo preso dal device tree, quindi con tutte le proprietà utili che definiscono il componente.

A questo punto viene inizializzata una "completion" (un particolare meccanismo di sincronizzazione, costruito sfruttando delle strutture wait_queue), che servirà poi per attendere il completamento dei trasferimenti.

Successivamente viene richiesta una zona di memoria per il dispositivo e viene mappata.

Viene richiesta una linea di interrupt, e come interrupt service routine viene assegnato un metodo definito in questo file, fsl_lpspi_isr, che verrà approfondito nella sezione 3.2.4. Ogni volta che la linea di interrupt scatenerà un evento, verrà eseguita la isr. [18]

```
1         irq = platform_get_irq(pdev, 0);
2         if (irq < 0) {
3                 ret = irq;
4                 goto out_controller_put;
5         }
6
7         ret = devm_request_irq(&pdev->dev, irq,
              fsl_lpspi_isr, 0,
8                                dev_name(&pdev->dev),
                                     fsl_lpspi);
9         if (ret) {
10                dev_err(&pdev->dev, "can't get irq%d: %d\n",
                      irq, ret);
11                goto out_controller_put;
12        }
```

Si passa ora all'inizializzazione dei clock. Si richiedono le due sorgenti di clock definite nel device tree, una di nome per e l'altra di nome ipg, e successivamente si abilita il clock chiamando il metodo fsl_lpspi_init_rpm, definito di seguito. Questo metodo abilita la gestione della potenza e identifica un timeout per la sospensione del modulo. [18]

```
1         fsl_lpspi->clk_per = devm_clk_get(&pdev->dev, "per")
              ;
2         if (IS_ERR(fsl_lpspi->clk_per)) {
3                 ret = PTR_ERR(fsl_lpspi->clk_per);
4                 goto out_controller_put;
5         }
6
7         fsl_lpspi->clk_ipg = devm_clk_get(&pdev->dev, "ipg")
              ;
8         if (IS_ERR(fsl_lpspi->clk_ipg)) {
9                 ret = PTR_ERR(fsl_lpspi->clk_ipg);
10                goto out_controller_put;
```

```
11          }
12
13          /* enable the clock */
14          ret = fsl_lpspi_init_rpm(fsl_lpspi);
15          if (ret)
16                  goto out_controller_put;
```

Infine viene letto il contenuto del registro Parameter del modulo LPSPI, per conoscere il numero massimo di parole che possono essere contenute nella fifo di ricezione e in quella di trasmissione. Questi parametri vengono immagazzinati nella struct fsl_lpspi_data.

```
1           temp = readl(fsl_lpspi->base + IMX7ULP_PARAM);
2           fsl_lpspi->txfifosize = 1 << (temp & 0x0f);
3           fsl_lpspi->rxfifosize = 1 << ((temp >> 8) & 0x0f);
```

Ora che il controllore del modulo è pronto, si passa alla sua registrazione. devm_spi_register_controller è un metodo definito in spi.c, ed al suo interno richiama spi_register_master, anch'esso definito in spi.c. Questo ulteriore metodo, nel caso di master, configura e registra tutti i gpio del dispositivo. In ogno caso alloca un numero di bus dinamico utilizzando una struttura idr, inizializza tutti i meccanismi di sicronizzazione che verranno utilizzati dal controllore (mutex, spinlock...), registra il nome del dispositivo e infine richiama diversi metodi che registrano il dispositivo nel kernel.

```
1           ret = devm_spi_register_controller(&pdev->dev,
                controller);
2           if (ret < 0) {
3                   dev_err(&pdev->dev, "spi_register_controller
                        error.\n");
4                   goto out_controller_put;
5           }
```

3.2.4 Come funzionano i trasferimenti

Per poter ricevere o trasmettere qualcosa è necessario utilizzare due struct importanti, spi_transfer e spi_message. La struct spi_transfer è l' unità base per effettuare un trasferimento. Essa contiene un buffer per immagazzinare i dati ricevuti e uno per i dati da trasmettere. Contiene anche numerosi parametri per configurare il trasferimento, ovvero il numero di byte da inviare

e trasmettere, la velocità massima espressa in frequenza, il numero di bit che compongono una parola, il numero di bit trasmessi per colpo di clock. Contiene poi dei parametri utili per l'utilizzo del dma. [17]

La struct spi_message contiene una coda di spi_transfer, quindi costituisce una serie di transfer consecutivi. Contiene un puntatore alla struct spi_device che lo utilizza, ed un puntatore ad una funzione da invocare quando i trasferimenti sono completati. [17] Il parametro frame_length rappresenta il numero totale di byte che costitiscono il messaggio, metre actual_length il numero di byte trasferiti dopo un segmento di trasferimento. Lo status è 0 in caso di trasferimento con successo, altrimenti un numero negativo.

Una volta inizializzata una struct spi_message con gli spi_transfer necessari, per invocare il trasferimento si utilizzano tre metodi a scelta: spi_async, spi_async_locked e spi_sync. Sono disponibili anche altri metodi equivalenti, ma al loro interno richiamano uno di questi tre.

1. spi_async

   Questo metodo richiama ___spi_validate, che al suo interno invoca

   ___spi_validate_bits_per_word. Questi metodi servono per verificare il tipo di trasferimento, ovvero se ad ogni colpo di clock viene trasferito un bit, due o quattro, e per definire il numero di bit per ogni parola. Infatti il questo numero deve essere necessariamente 8, 16, o 32. Se viene indicato un numero diverso viene modificato con un uno dei precedenti. Successivamente spi_async invoca ___spi_async, che richiama la funzione transfer del controllore. Questa funzione è spi_queued_transfer. Essa al suo interno richiama ___spi_pump_messages. É qui che viene effettuato il vero trasferimento, e sono richiamate diverse funzioni definite in spi_fsl_lpspi.c, tra cui il metodo fsl_lpspi_transfer_one_msg, che è approfondito più avanti.

2. spi_async_locked

   É una versione di spi_async con la garanzia che il bus è usato esclusivamente dal dispositivo.

3. spi_sync

   Questo metodo richiama al suo interno ___spi_sync, che come spi_async

richiama spi_validate. Se come metodo da utilizzare per il trasferimento è indicato

___spi_queued_transfer, allora questo viene chiamato, altrimenti viene utilizzata spi_async_locked. Nel progetto implementato nel corso di questa tesi viene utilizzato ___spi_queued_transfer. Differentemente da quanto avviene nei due metodi precedenti, viene aspettata una wait_queue, che viene svegliata quando i trasferimenti sono completati.

Nel corso di questa tesi il metodo fsl_lpspi_transfer_one_msg è stato opportunamente modificato per adattarlo alle esigenze della scheda i.MX8QM lpddr4 arm2. Di seguito è riportato il codice.

```
1  static int fsl_lpspi_transfer_one_msg(struct spi_controller
       *controller,
2                                         struct spi_message *
                                             msg)
3  {
4          struct fsl_lpspi_data *fsl_lpspi =
5                                      spi_controller_get_devdata(
                                           controller);
6          struct spi_device *spi = msg->spi;
7          struct spi_transfer *xfer;
8          bool is_first_xfer = true;
9          int ret = 0, i, j;
10         msg->status = 0;
11         msg->actual_length = 0;//the number of bytes
               transmitted
12         list_for_each_entry(xfer, &msg->transfers,
               transfer_list) {//for each spi_transfer
13                 if((xfer->len == 0)){
14                         //software reset, to clear all
                               registers except the control
                               register.
15                         //this prepares the slave module for
                                the next communication cycle.
16                         u32 temp = readl(fsl_lpspi->base +
                               IMX7ULP_CR);
17                         temp |= CR_RST;
18                         writel(temp, fsl_lpspi->base +
                               IMX7ULP_CR);
```

```
19                          temp &= ~CR_RST ;
20                          writel(temp, fsl_lpspi->base +
                                IMX7ULP_CR);
21                          timesUsed=0;//do this because in the
                                first transfer the slave module
                                will be enabled after the
                                transmit fifo is filled.
22                          //see the function
                                fsl_lpspi_write_tx_fifo
23                  }
24                  else {//normal transfer
25                              fsl_lpspi_setup_transfer(spi
                                    , xfer);
26                              fsl_lpspi_set_cmd(fsl_lpspi,
                                    is_first_xfer);
27                              is_first_xfer = false;
28                              ret = fsl_lpspi_transfer_one
                                    (controller, spi, xfer);
29                              if (ret < 0)
30                                      goto complete;
31                  }
32          }
33
34
35  complete:
36
37          //reset the counter
38          timesUsed = 0;
39          msg->status = ret;
40          spi_finalize_current_message(controller);
41
42          return ret;
43  }
```

Per ogni spi_transfer del messaggio viene effettuato un controllo in base alla lunghezza indicata. Se questa lunghezza è un numero maggiore di zero allora si procede alla configurazione del trasferimento. Viene invocato il metodo fsl_lpspi_setup_transfer, che configura i vari parametri con cui effettuare il trasferimento attuale (numero di bit per parola, velocità, polarità e fase di clock, watermark ...), e in più invoca fsl_lpspi_config. Quest'ultima scrive nel registro CFGR1 il bit che indica se il modulo è di tipo master o di tipo slave. Nel caso fosse slave inverte le direzioni dei segnali SOUT e SIN:

SIN è usato come output e SOUT è usato come input. Se poi il modulo è un master, oppure uno slave che ha già effettuato trasferimenti, il modulo viene abilitato. Se invece si è nel caso di uno slave al suo primo utilizzo, l'abilitazione avverrà durante il primo trasferimento. Questo perchè in caso di slave il TCR (Transmit Command Register) deve essere riempito prima di abilitare il modulo.

Successivamente si continua con la configurazione e si invoca

fsl_lpspi_set_cmd, che scrive la parola di comando nel Transmit Command Register. Infine si passa al trasferimento vero e proprio invocando fsl_lpspi_transfer_one.

Se la lunghezza è invece zero, allora viene effettuato un software reset, in cui tutti i registri e la logica (eccetto il registro di controllo) vengono riportati allo stato originale.

Questi metodi sono stati adattati alla versione del driver per il kernel 4.14.62 [18], usato nella versione Sumo di Yocto. Inoltre il metodo fsl_lpspi_config è stato ulteriormente modificato, per permettere al modulo in modalità slave di essere abilitato solo dopo la scrittura del registro di trasmissione ed evitare così errori di trasmissione, dovuti alla particolarità dell'hardware di destinazione.

```
1  static int fsl_lpspi_config(struct fsl_lpspi_data *fsl_lpspi
       )
2  {
3          u32 temp;
4          int ret;
5
6          if (!fsl_lpspi->is_slave) {
7                  ret = fsl_lpspi_set_bitrate(fsl_lpspi);
8                  if (ret)
9                          return ret;
10         }
11
12         fsl_lpspi_set_watermark(fsl_lpspi);
13
14
15         if (!fsl_lpspi->is_slave)
16                 temp = CFGR1_MASTER;
17         else
18                 temp = CFGR1_PINCFG;
19         if (fsl_lpspi->config.mode & SPI_CS_HIGH)
```

```
20                  temp |= CFGR1_PCSPOL;
21
22          writel(temp, fsl_lpspi->base + IMX7ULP_CFGR1);
23
24
25          //only reset fifos. the module will be enabled after
                the transmit fifo is filled
26          if((fsl_lpspi->is_slave) && (timesUsed == 0)){
27                  temp = readl(fsl_lpspi->base + IMX7ULP_CR);
28                  temp |= CR_RRF | CR_RTF;
29                  //temp |= CR_RRF | CR_RTF | CR_MEN;
30                  writel(temp, fsl_lpspi->base + IMX7ULP_CR);
31          }
32          else {//reset fifos and enable module
33                  temp = readl(fsl_lpspi->base + IMX7ULP_CR);
34                  //temp |= CR_RRF | CR_RTF;
35                  temp |= CR_RRF | CR_RTF | CR_MEN;
36                  writel(temp, fsl_lpspi->base + IMX7ULP_CR);
37          }
38
39          return 0;
40  }
41
42
43  static int fsl_lpspi_transfer_one(struct spi_controller *
        controller,
44                                    struct spi_device *spi,
45                                    struct spi_transfer *t)
46  {
47          struct fsl_lpspi_data *fsl_lpspi =
48                                  spi_controller_get_devdata(
                                        controller);
49          int ret;
50
51          fsl_lpspi->tx_buf = t->tx_buf;
52          fsl_lpspi->rx_buf = t->rx_buf;
53          fsl_lpspi->remain = t->len ;
54
55
56
57          reinit_completion(&fsl_lpspi->xfer_done);
58          fsl_lpspi->slave_aborted = false;
```

```
59          //write the transmit fifo
60          fsl_lpspi_write_tx_fifo(fsl_lpspi);
61          //activate the wait queue. If the module is master,
               it is set a timeout.
62          //if it is slave, it will wait until the right
               number of bytes is arrived.
63          ret = fsl_lpspi_wait_for_completion(controller);
64          if (ret)
65                  return ret;
66
67          //clear fifos and interrupt register and disable
               modules
68          fsl_lpspi_reset(fsl_lpspi);
69
70
71          return 0;
72  }
```

Il metodo fsl_lpspi_transfer_one imposta alcuni campi della struttura fsl_lpspi_data, ovvero il numero di byte da inviare/ricevere e gli indirizzi dei due buffer dove immagazzinare i dati ricevuti e quelli da inviare. Si inizializza la struttura per poi attendere il termine del trasferimento, si scrivono i dati da mandare nella fifo di trasmissione (tramite la chiamata al metodo fsl_lpspi_write_tx_fifo) e si aspetta che il trasferimento termini. Nel caso di modulo master, si attiva un timeout. Quando il tempo termina si smette di aspettare e si prelevano i dati contenuti nella fifo di ricezione anche se non sono pronti. Nel caso di slave il tempo di attesa è infinito, e si viene svegliati solo quando il numero richiesto di byte sono presenti nella fifo di ricezione.

Una volta terminato, si disabilitano gli interrupt, si porta il registro dello stato ai valori predefiniti, si cancella il contenuto delle fifo e si disabilita il modulo.

Il metodo per trasmettere i dati riempie la fifo di trasmissione e poi, nel caso di dispositivo slave al suo primo utilizzo, abilita il modulo. Se il registro di trasmissione è stato riempito completamente si abilita l'interrupt di trasmissione (i dati sono pronti per essere trasmessi), altrimenti si abilita l'interrupt di ricezione, che verrà quindi scatenato quando saranno presenti i dati richiesti sulla fifo di ricezione. Anche in questo caso il codice è stato aggiornato alla versione del kernel 4.14.62 [18], e ulteriormente modificato per permettere al modulo in modalità slave di essere abilitato solo dopo che la fifo di trasmissione è stata scritta.

```
1  static void fsl_lpspi_write_tx_fifo(struct fsl_lpspi_data *
       fsl_lpspi)
2  {
3          u8 txfifo_cnt;
4          u32 temp;
5
6          txfifo_cnt = readl(fsl_lpspi->base + IMX7ULP_FSR) &
               0xff;
7
8          while (txfifo_cnt < fsl_lpspi->txfifosize) {
9                  if (!fsl_lpspi->remain)
10                         break;
11                 fsl_lpspi->tx(fsl_lpspi);
12                 txfifo_cnt++;
13         }
14     if((fsl_lpspi->is_slave) &&(timesUsed == 0)){
15         temp = readl(fsl_lpspi->base + IMX7ULP_CR);
16         temp |= CR_MEN;
17         //temp |= CR_RRF | CR_RTF | CR_MEN;
18         writel(temp, fsl_lpspi->base + IMX7ULP_CR);
19         timesUsed++;
20     }
21
22         if (txfifo_cnt < fsl_lpspi->txfifosize) {
23                 if (!fsl_lpspi->is_slave) {
24                         temp = readl(fsl_lpspi->base +
                               IMX7ULP_TCR);
25                         temp &= ~TCR_CONTC;
26                         writel(temp, fsl_lpspi->base +
                               IMX7ULP_TCR);
27                 }
28                 fsl_lpspi_intctrl(fsl_lpspi, IER_RDIE);
29         } else
30                 fsl_lpspi_intctrl(fsl_lpspi, IER_TDIE);
31 }
```

Il metodo tx riempie la fifo di trasmissione con la parola data.

Una volta scatenato un interrupt viene invocata la interrupt service routine fsl_lpspi_isr, il cui codice è preso solo in parte dal kernel 4.14.62 [18]:

```
1  static irqreturn_t fsl_lpspi_isr(int irq, void *dev_id)
2  {
3          u32 temp_SR, temp_IER;
4          struct fsl_lpspi_data *fsl_lpspi = dev_id;
5
6          temp_IER = readl(fsl_lpspi->base + IMX7ULP_IER);
7          fsl_lpspi_intctrl(fsl_lpspi, 0);
8          temp_SR = readl(fsl_lpspi->base + IMX7ULP_SR);
9
10         //printk("isr ier %lu\n", temp_IER);
11         //printk("isr status %lu\n", temp_SR);
12
13         fsl_lpspi_read_rx_fifo(fsl_lpspi);
14
15         if ((temp_SR & SR_TDF) && (temp_IER & IER_TDIE)) {
16                 fsl_lpspi_write_tx_fifo(fsl_lpspi);
17                 return IRQ_HANDLED;
18         }
19
20         if (temp_SR & SR_RDF && (temp_IER & IER_RDIE)) {
21                 complete(&fsl_lpspi->xfer_done);
22                 return IRQ_HANDLED;
23         }
24
25         return IRQ_NONE;
26 }
```

Infatti in quella versione la interrupt service routine sfruttava il flag di completamento frame, in questo caso si utilizza il flag di lettura.

Questa isr legge il contenuto del registro di interrupt e può quindi identificare la causa. Inoltre disabilita gli interrupt e procede con la lettura della fifo di ricezione, tramite fsl_lpspi_read_rx_fifo. Se la causa dell'interrupt è stato il flag di trasmissione allora si richiama la funzione per inviare dati, mentre se la causa è stato il flag di ricezione allora viene svegliata la completion in attesa.

Di seguito è riportato il codice del metodo per leggere dalla fifo di ricezione. Il metodo rx legge una parola alla volta dal registro. Viene letta quindi una parola alla volta fino a quando la fifo non è vuota. [18]

```
1 static void fsl_lpspi_read_rx_fifo(struct fsl_lpspi_data *
    fsl_lpspi)
2 {
3         while(!(readl(fsl_lpspi->base + IMX7ULP_RSR) &
            RSR_RXEMPTY)){
4                 fsl_lpspi->rx(fsl_lpspi);
5         }
6 }
```

3.3 Il protocollo implementato

Il protocollo di comunicazione scelto è simile a un protocollo sviluppato da Bosch per il suo componente EMGS625 (SC900714AE), come descritto nel manuale del suddetto prodotto. [22]

3.3.1 Protocollo lato master

Di seguito è riportato il protocollo implementato dalla scheda i.MX8QM lpddr4 arm2 in modalità master. La suddetta scheda è collegata alla centralina sviluppata dall'azienda, la quale comunica con la prima tramite SPI di tipo slave, utilizzando il modulo lpspi3.

Il master invia allo slave due parole di 16 bit ciascuna, contenenti un indirizzo, il numero di parole da mandare/ricevere ed un bit che indica se l'operazione sarà una scrittura o una lettura. In particolare, i dati sono così strutturati:

bit	15	14 - 0
contenuto	0 se si vuole scrivere 1 se si vuole ricevere	indirizzo

Tabella 3.2. La prima parola di comando in modalità master

bit	15 - 0
contenuto	numero di parole da trasmettere - ricevere

Tabella 3.3. La seconda parola di comando in modalità master

Il bit più significativo indica il tipo di richiesta (0 se il master intende scrivere sulla memoria dello slave, 1 se intende leggere parte della memoria). I successivi 15 bit indicano l'indirizzo della memoria dello slave alla quale il master vuole leggere o scrivere. I 16 bit della seconda parola indicano il numero di parole che si intende leggere o scrivere.

Supponendo che questo numero sia n, si procede con uno dei casi:

- Nel caso il master abbia richiesto una scrittura, il master invia n parole allo slave, che verranno scritte all'indirizzo (più un eventuale offset indicato dalla centralina slave) dato in precedenza.

- Nel caso il master abbia richiesto una lettura, lo slave invia n parole al master. Queste parole sono prese dalla memoria dello slave a partire dall'indirizzo dato in precedenza (più un eventuale offset indicato dalla centralina slave).

Completano il protocollo dei controlli di integrità per verificare che lo slave abbia restituito dati validi.

3.3.2 Protocollo lato slave

Il protocollo implementato dalla scheda i.MX8QM lpddr4 arm2 in modalità slave rispecchia il comportamento della centralina dell'azienda nel caso precedente. Sono inoltre introdotte delle parole di sincronizzazione ed una parola che indica l' inizio del trasferimento, per permettere alla scheda di comprendere quando il master intende iniziare una comunicazione.

Inizialmente, viene definito un indirizzo, che corrisponde all'indirizzo di spazio utente di partenza in cui verranno effettuate tutte le operazioni. L'indirizzo che arriva ad ogni messaggio sarà un offset applicato all'indirizzo di partenza. Anche in questo caso le parole hanno come dimensione 16 bit.

Di seguito è riportata una tabella che mostra il contenuto delle prime quattro parole trasmesse dalla centralina master e ricevute dal fusion processor in modalità slave:

Parola	Contenuto
0	start word
1	parola di sincronizzazione
2	prima parola di comando, strutturata come il caso precedente
3	seconda parola di comando, strutturata come il caso precedente

Tabella 3.4. La sequenza iniziale con le parole di comando trasmesse dalla centralina HDF

La scheda Aurix quando intende iniziare una conversazione invia una parola iniziale (start word) univoca. Successivamente viene inviata una parola di sicronizzazione, che può avere qualsiasi valore, e dopo di che sono mandate le parole di comando. Queste due parole sono strutturate analogamente al caso precedente: una parola contenente un bit per indicare se effettuare una lettura o una scrittura e un indirizzo, e una seconda parola con un numero, n, che indica quante saranno le parole successive. Seguono quindi le n parole.

- In caso di lettura (il master richiede di leggere dei dati) lo slave trasmette i dati al master a partire dall'indirizzo inviato. I dati sono prelevati dallamemoria utente.
- In caso di scrittura (il master intende scrivere dati nella memoria dello slave) le n parole trasferite verrano copiate nella memoria utente.

Infine il master invia un'ultima parola di sincronizzazione per indicare la fine di un trasferimento.

3.4 I driver sviluppati

Per implementare il protocollo precedentemente descritto sono stati sviluppati due driver, uno per la comunicazione in modalità slave e l'altro per la modalità master. Il primo driver è registrato con il nome "spi-slave" e ha major number 236, il secondo con il nome "spi-master" con major number 237.

Come per il modulo spi_fsl_lpspi, anche i driver creati appositamente per il progetto, spi-master e spi-slave, possiedono una funzione probe. All'avvio vengono eseguite le loro funzioni di inzializzazione, che registrano i driver con il bus spi. Per registrarli viene chiamata la funzione spi_register_driver [16], definita in spi.c. Questa funzione, come platform_driver_register, chiama driver_register e da qui viene invocata la funzione probe dei driver. Per questi due moduli sono definite anche delle funzioni richiamate quando sono rimossi. Queste funzioni sono indicate da ___exit, ed eseguono le operazioni inverse a quelle di spi_init. Il metodo spi_register_driver non solo invoca driver_register ma associa i metodi da chiamare quando si effettuano operazioni di probe, remove e shutdown.

La funzione spi_unregister_driver (definita in spi.c) chiama driver_unregister, l'inverso di driver_register.

Le funzioni probe dei driver spi-master e spi-slave sono strutturate in modo equivalente. Di seguito è riportata l'implementazione dei metodi probe, remove, spi_init e spi_exit per quanto riguarda il modulo di tipo slave:

```
1  static int spislave_probe(struct spi_device *spi){
2          spi_device = spi;
3          int ret = -1;
4          if(!spi_device)
5          spi_device->bits_per_word= 16;
6          spi_device->max_speed_hz = 100000;
7          spi_device->mode = SPI_MODE_1;
8          ret = spi_setup(spi_device);
9          return ret;
10 }
11
12 static int spislave_remove(struct spi_device *spi){
13         spi_device = NULL;
14         return 0;
15 }
16
```

```
17 static const struct of_device_id spislave_dt_ids[] = {
18         { .compatible = "linux,spislave" },
19         {},
20 };
21 MODULE_DEVICE_TABLE(of, spislave_dt_ids);
22
23 static struct spi_driver spislave_driver = {
24         .driver = {
25                 .owner = THIS_MODULE,
26                 .name = "spislave",
27                 .of_match_table = of_match_ptr(
                       spislave_dt_ids),
28         },
29         .probe = spislave_probe,
30         .remove = spislave_remove,
31 };
32
33 static int __init spi_init(void)
34 {
35         alloc_chrdev_region(&spislave_dev, 0, 1, DEVICE_NAME
              );
36         spislave_class = class_create(THIS_MODULE,
              DEVICE_NAME);
37         cdev_init(&spislave_cdev, &spislave_fops);
38         spislave_cdev.owner = THIS_MODULE;
39         cdev_add(&spislave_cdev, spislave_dev, 1);
40         spi_register_driver(&spislave_driver);
41     return 0;
42 }
43
44
45 static void __exit spi_exit(void)
46 {
47         cdev_del(&spislave_cdev);
48         unregister_chrdev_region(spislave_dev, 1);
49     spi_unregister_driver(&spislave_driver);
50 }
```

Il metodo probe richiama spi_setup, definito in spi.c. Questo metodo è utile per registrare i parametri con cui viene effettuata la comunicazione, quali il modo di utilizzo (polarità e fase di clock), velocità, bit per parola ecc, e verrà richiamato ogni volta che questi parametri devono essere modificati.

Una struct di tipo of_device_id permette di collegare il driver ad un dispositivo del device tree, quindi viene definito quale driver associare al determinato dispositivo. Nel caso del driver slave, l'etichetta associata è "linux,spislave". Questa etichetta è segnata anche in un nodo del modulo lpspi0 nel file fsl-imx8qm-lpddr4-arm2.dts (come indicato alla sezione 3.2.1). In questo modo il suddetto nodo verrà associato al driver "spi-slave", e quando si intende aprire il device file "/dev/spi-slave" ed effettuare operazioni allora i dati verranno trasferiti tramite il modulo lpspi0. Analogamente per il modulo master, l'etichetta è "linux,spimaster", la stessa identificata nel nodo di lpspi3.

Infine per la registrazione dei moduli viene utilizzata una struct spi_driver, che registra il nome, la struct of_device_id, i metodi probe e remove. In questo caso il metodo remove assegna NULL al puntatore della struct spi_device precedentemente registrata.

3.4.1 Il driver master

Il file spi-module.c implementa il codice del driver registrato come "spi-master". Di seguito è riportata la struttura file_operations in cui sono indicati i metodi richiamabili dall'utente. In questo caso sono richiamabili i metodi open, release (invocato tramite il metodo close), write, read e ioctl. A seconda della configurazione del sistema quando si effettua una chiamata a ioctl viene scelta unlocked_ioctl oppure compat_ioctl. In ogni caso compat_ioctl semplicemente invoca al suo interno unlocked_ioctl.

```
1 struct file_operations spimaster_fops = {
2         .owner = THIS_MODULE,
3         .read = spimaster_read,
4         .write = spimaster_write,
5         .open = spimaster_open,
6         .release = spimaster_release,
7         .unlocked_ioctl = spimaster_ioctl,
8         .compat_ioctl = spimaster_compat,
9 };
```

Il metodo open semplicemente stampa un avviso che il modulo è stato aperto. Analogamente il metodo release stampa l'avviso opposto.

```
static int spimaster_open(struct inode *inode, struct file *
    filp){
        printk("spimaster open \n");
        return 0;
}

static int spimaster_release(struct inode *inode, struct
    file *filp){
        printk("spimaster release \n");
        return 0;
}
```

Il metodo ioctl è utilizzato in questo modulo per effettuare modifiche ai parametri che configurano la comunicazione. Sono state introdotte delle macro, nel file spi-module.h, per rendere le richieste più semplici. Di seguito sono riportate le suddette macro, con i loro usi nei commenti:

```
#define SPI_RD_SPEED 10
#define SPI_WR_SPEED 12
#define SPI_RD_BPW 20
#define SPI_WR_BPW 22
#define SPI_RD_MODE 30
#define SPI_WR_MODE 32
```

I modi di utilizzo sono definiti come segue: [17]

```
#define SPI_CPHA        0x01                    /* clock
    phase */
#define SPI_CPOL        0x02                    /* clock
    polarity */
#define SPI_MODE_0      (0|0)                   /* (original
     MicroWire) */
#define SPI_MODE_1      (0|SPI_CPHA)
#define SPI_MODE_2      (SPI_CPOL|0)
#define SPI_MODE_3      (SPI_CPOL|SPI_CPHA)
```

Di seguito è riportato il codice del metodo ioctl:

```
1  static long spimaster_ioctl(struct file *filp, unsigned int
       cmd, unsigned long arg){
2          int ret;
3          int spd;
4          int tmp;
5
6          switch(cmd){
7          case SPI_WR_SPEED://Set max speed in hertz
8                  spd = __get_user(spi_device->max_speed_hz, (
                       uint32_t __user*) arg);
9
10                 ret = spi_setup(spi_device);
11
12                 break;
13         case SPI_RD_SPEED:
14         //read the actual speed value
15                 ret = __put_user(spi_device->max_speed_hz, (
                       uint32_t __user*) arg);
16                 break;
17
18         case SPI_WR_BPW://Set the number of bits per word.
               Default is 16 (from the function probe)
19                 spd = __get_user(spi_device->bits_per_word,
                       (uint8_t __user*) arg);
20
21                 ret = spi_setup(spi_device);
22
23                 break;
24         case SPI_RD_BPW://read the number of bits per word
25                 ret = __put_user(spi_device->bits_per_word,
                       (uint8_t __user*) arg);
26
27                 break;
28
29         case SPI_WR_MODE://set the mode of operation (
               polarity, phase of clock...)
30                 spd = __get_user(spi_device->mode, (uint8_t
                       __user*) arg);
31                 ret = spi_setup(spi_device);
32
```

```
33                  break;
34          case SPI_RD_MODE: //read the actual mode of operation
35                  ret = __put_user(spi_device->mode, (uint8_t
                        __user*) arg);
36
37                  break;
38
39
40          default:
41                  ret = -1;
42                  break;
43          }
44
45          return ret;
46  }
```

Le comunicazioni vengono effettuate tramite i metodi read e write. Di seguito è riportato il codice del metodo read:

```
1  ssize_t spimaster_read(struct file *filp, char __user *buf,
       size_t count, loff_t *f_pos){
2          char character;
3          uint16_t control_word;
4          uint8_t channel_mask = 0b00000111;
5          int count_b = count;
6          int index;
7
8          struct spi_transfer* transfers = kmalloc((count+1)*
               sizeof(struct spi_transfer), GFP_KERNEL);
9
10         if(count>65535){
11                 return -1;
12         }
13
14         copy_from_user(&character, buf +1 , 1);
15         control_word = character;
16         control_word = control_word << 8;
17         copy_from_user(&character, buf, 1);
18         control_word |= character;
19
20         control_word = control_word | 0b1000000000000000;
21
```

```
22          char* rbuffer = kmalloc(4 , GFP_KERNEL);
23          memset(rbuffer, 0, sizeof(rbuffer));
24          char* tbuffer = kmalloc(4 , GFP_KERNEL);
25          memset(tbuffer, 0, sizeof(tbuffer));
26
27          memcpy(tbuffer, &control_word, 2);
28
29          memcpy(tbuffer+2, &control_word, 2);
30
31          char* receiveb = kmalloc(2*count , GFP_KERNEL);
32          memset(receiveb, 0, sizeof(receiveb));
33
34          transfers[0].tx_buf= tbuffer;
35          transfers[0].rx_buf= rbuffer;
36          transfers[0].delay_usecs = 10;
37          transfers[0].bits_per_word = 16;
38          transfers[0].cs_change = 1;
39          transfers[0].len = 4;
40          transfers[0].speed_hz = spi_device->max_speed_hz;
41          transfers[0].rx_nbits = SPI_NBITS_SINGLE;
42          transfers[0].tx_nbits = SPI_NBITS_SINGLE;
43
44          int pos = 0;
45           for(index=1; index<=count; index++)//configuration
                of the "count" transfers
46      {
47            transfers[index].rx_buf = &receiveb[pos];
48            transfers[index].tx_buf = NULL;
49            transfers[index].len = 2;
50            transfers[index].delay_usecs = 10;
51            transfers[index].cs_change = 1;
52            transfers[index].bits_per_word = 16;
53            transfers[index].speed_hz = spi_device->
                 max_speed_hz;
54            transfers[index].rx_nbits = SPI_NBITS_SINGLE;
55            transfers[index].tx_nbits = SPI_NBITS_SINGLE;
56            pos+=2;
57      }
58
59          struct spi_message *m = kmalloc(sizeof(struct
              spi_message *), GFP_KERNEL);
60          spi_message_init(m);
```

```
61
62          spi_message_init_with_transfers(m, transfers, count
               +1);
63
64          int status= spi_sync(spi_device, m);
65
66          index = copy_to_user(buf, receiveb, 2*(count));
67          if(index != 0)
68            {
69                    return -1;
70            }
71
72          kfree(transfers);
73          kfree(rbuffer);
74          kfree(tbuffer);
75          kfree(m);
76
77          return index;
78  }
```

Il prototipo del metodo indica che l'utente deve fornire un indirizzo ed un numero:

```
1  ssize_t spimaster_read(struct file *filp, char __user *buf,
      size_t count, loff_t *f_pos);
```

In base al protocollo scelto, l'utente richiede di poter leggere la memoria della centralina slave a partire dall'indirizzo fornito buf e per un numero count di parole. Le parole ricevute vengono poi restituite alla memoria utente. Alla riga 8 viene allocato un vettore di spi_transfer di numero pari a count. Questo perchè il trasferimento avviene una parola alla volta, e per ogni parola corrisponde un spi_transfer. Alla riga 10 viene poi fatto un controllo sul numero massimo di parole. Dalla riga 14 alla riga 20 viene creata la prima parola di comando: il bit più significativo è 1, ad indicare un'operazione di lettura. I restanti bit rappresentano l'indirizzo. La seconda parola di comando sarà comporta dal numero di parole. Vengono poi allocati i vari buffer necessari per i trasferimenti: due buffer di quattro byte ciascuno per il trasferimento delle parole di comando (uno per ricezione e l'altro per trasferimento), ed un buffer di count parole per ricevere tutti i dati necessari. Il trasferimento viene effettuato tramite la chiamata ad spi_sync. Infine le strutture allocate dinamicamente vengono liberate.

Il comportamento del metodo write è analogo. Con questo metodo l'utente intende scrivere un numero di parole nella memoria della centralina slave a partire dall'indirizzo fornito.

Di seguito ne è riportato il codice:

```
1  ssize_t spimaster_write(struct file *filp, const char __user
       *buf, size_t count, loff_t *f_pos){
2          char character;
3          uint16_t control_word;
4          unsigned char* bp;
5          uint8_t channel_mask = 0b00000111;
6          int count_b = count;
7          int index;
8
9          struct spi_transfer* transfers = kmalloc((count+1)*
               sizeof(struct spi_transfer), GFP_KERNEL);
10
11
12         if(count>65535){
13                 return -1;
14         }
15
16         copy_from_user(&character, buf +1 , 1);
17         control_word = character;
18         control_word = control_word << 8;
19         copy_from_user(&character, buf, 1);
20         control_word |= character;
21
22         control_word = control_word & 0b0111111111111111;
23
24         char *tbuffer = kmalloc(4, GFP_KERNEL);
25         memcpy(tbuffer, &control_word, 2);
26
27         control_word = (uint16_t)count;
28         memcpy(tbuffer+2, &control_word, 2);
29
30         char *rbuffer = kmalloc(4, GFP_KERNEL);
31         memset(rbuffer, 0, sizeof(rbuffer));
32
33         //the transfer for the control words, with length of
               4 bytes
34         transfers[0].tx_buf= tbuffer;
```

```
35          transfers[0].rx_buf= rbuffer;
36          transfers[0].delay_usecs = 10;
37          transfers[0].bits_per_word = 16;
38          transfers[0].cs_change = 1;
39          transfers[0].len = 4;
40          transfers[0].speed_hz = spi_device->max_speed_hz;
41          transfers[0].rx_nbits = SPI_NBITS_SINGLE;
42          transfers[0].tx_nbits = SPI_NBITS_SINGLE;
43
44          char* sendb = kmalloc(2*count , GFP_KERNEL);
45          memset(sendb, 0, sizeof(sendb));
46
47          int status = copy_from_user(sendb, 2 + buf, 2*(count
                ));
48
49          if(status != 0)
50            {
51                    return -1;
52            }
53
54
55          int pos = 0;
56           for(index=1; index<=count; index++)
57        {
58            transfers[index].tx_buf = &sendb[pos];
59            transfers[index].rx_buf = NULL;
60            transfers[index].len = 2;
61            transfers[index].delay_usecs = 10;
62            transfers[index].cs_change = 1;
63            transfers[index].bits_per_word = 16;
64            transfers[index].speed_hz = spi_device->
                  max_speed_hz;
65            transfers[index].rx_nbits = SPI_NBITS_SINGLE;
66            transfers[index].tx_nbits = SPI_NBITS_SINGLE;
67            pos+=2;
68         sendb[pos+1]);
69        }
70
71          struct spi_message *m = kmalloc(sizeof(struct
               spi_message *), GFP_KERNEL);
72          spi_message_init(m);
73
```

```
74          spi_message_init_with_transfers(m, transfers, count
               +1);
75
76          status= spi_sync(spi_device, m);
77
78          kfree(transfers);
79          kfree(tbuffer);
80          kfree(rbuffer);
81          kfree(m);
82
83          return 0;
84  }
```

Il prototipo è equivalente a quello del metodo per leggere: l'utente fornisce un indirizzo buf ed un numero di parole count.

```
1  ssize_t spimaster_write(struct file *filp, const char __user
       *buf, size_t count, loff_t *f_pos);
```

Il trasferimento avviene una parola alla volta, quindi è allocato un vettore di spi_transfer di dimensione count. Le righe da 16 a 22 costruiscono la prima parola di comando, con 0 come bit più significativo, ad indicare una scrittura. I dati da inviare sono presi dall'indirizzo fornito dall'utente, alla riga 47. Infatti la prima parola a quell'indirizzo rappresenta l'indirizzo in cui scrivere i dati nella centralina slave, le successive count parole sono i dati da trasmettere. Il trasferimento avviene tramite la chiamata a spi _sync.

3.4.2 Il driver slave

Il modulo slave è stato progettato in modo analogo. Il file che ne implementa il codice ha nome spi-slave.c. La struttura file_operations è strutturata allo stesso modo di quella del modulo spi-master, ad eccezione dei metodi read e write, che non sono richiamabili dall'utente. Questo perchè è presente un metodo particolare che rimane in ascolto ciclicamente dei comandi inviati dalla centralina master. Questo metodo è chiamato spislave_protocol.

```
1 struct file_operations spislave_fops = {
2         .owner = THIS_MODULE,
3         .open = spislave_open,
4         .release = spislave_release,
5         .unlocked_ioctl = spislave_ioctl,
6         .compat_ioctl = spislave_compat,
7 };
```

Il metodo open stampa un avviso che il modulo è stato aperto. Analogamente il metodo release stampa l'avviso che il modulo è stato chiuso.

```
1 static int spislave_open(struct inode *inode, struct file *
    filp){
2         printk("spislave open \n");
3         return 0;
4 }
5
6 static int spislave_release(struct inode *inode, struct file
     *filp){
7         printk("spislave release \n");
8         return 0;
9 }
```

Il metodo ioctl è utilizzato in questo modulo non solo per effettuare le modifiche ai parametri che configurano la comunicazione(anche in questo caso sono state introdotte delle macro, nel file spi-slave.h), ma anche per richiamare il metodo spislave_protocol. Di seguito sono riportate le macro in spi-slave.h e il codice di spislave_ioctl:

```
#define SPI_RD_SPEED 10
#define SPI_WR_SPEED 12
#define SPI_RD_BPW 20
#define SPI_WR_BPW 22
#define SPI_RD_MODE 30
#define SPI_WR_MODE 32
```

```
static long spislave_ioctl(struct file *filp, unsigned int
    cmd, unsigned long arg){

        int ret=0;
        int tmp;

        switch(cmd){
                case SLAVE_RD_SPEED://read the actual speed
                    value
                ret = __put_user(spi_device->max_speed_hz, (
                    uint32_t __user*) arg);

                break;
                case SLAVE_WR_SPEED://Set max speed in hertz
                ret = __get_user(spi_device->max_speed_hz, (
                    uint32_t __user*) arg);

                ret = spi_setup(spi_device);

                break;

                case SLAVE_RD_MODE://read the actual mode of
                     operation
                ret = __put_user(spi_device->mode, (uint8_t
                    __user*) arg);

                break;
                case SLAVE_WR_MODE://set the mode of
                    operation (polarity, phase of clock...)
                ret = __get_user(spi_device->mode, (uint8_t
                    __user*) arg);

                ret = spi_setup(spi_device);

                break;
```

```
28                  case SLAVE_PROT://start the protocol
29
30                  spislave_protocol((char __user*) arg);
31                  break;
32                  default://start the protocol
33                  spislave_protocol((char __user*) arg);
34                  break;
35          }
36
37          return ret;
38  }
```

Il metodo spislave_protocol implementa il vero e proprio protocollo di comunicazione. Ne viene di seguito inserito il codice:

```
1   static void spislave_protocol(char __user* user_buf ){
2           u16 address;
3           u16 count;
4           int index, j;
5
6           struct spi_transfer transfers;
7
8           char c = 80;
9           u8 tx[80];
10          u8 sendbuffer[1000];
11
12          for(j=0;j<500;j+=1){
13                  sendbuffer[j] = c;
14                  sendbuffer[j+1] = c;
15                  c++;
16          }
17
18          c=0;
19          for(j=0;j<8;j+=1){
20                  tx[j] = c;
21
22                  c++;
23          }
24          char cmd_word[6];
25          char start_word[2];
26          char tmp1, tmp2;
27          u16 temp;
```

```
28          int tx_index=0;
29
30          struct spi_message *m ;
31
32          bool rw;
33          bool first = true;
34          m = kmalloc(sizeof(struct spi_message *),
              GFP_KERNEL);
35
36          int indice=0;
37
38          transfers.tx_buf = tx;
39          transfers.rx_buf = start_word;
40          transfers.len = 0;
41          transfers.delay_usecs = 0;
42
43          transfers.bits_per_word = 16;
44          transfers.speed_hz = spi_device->max_speed_hz;
45          transfers.rx_nbits = SPI_NBITS_SINGLE;
46          transfers.tx_nbits = SPI_NBITS_SINGLE;
47
48          spi_message_init(m);
49
50          spi_message_init_with_transfers(m, &transfers, 1);
51
52          int status= spi_sync(spi_device, m);
53
54
55          while(1){
56                  do{
57                          transfers.tx_buf = tx;
58                          transfers.rx_buf = start_word;
59                          transfers.len = 2;
60                          transfers.delay_usecs = 0;
61
62                          transfers.bits_per_word = 16;
63                          transfers.speed_hz = spi_device->
                              max_speed_hz;
64                          transfers.rx_nbits =
                              SPI_NBITS_SINGLE;
65                          transfers.tx_nbits =
                              SPI_NBITS_SINGLE;
```

```
66
67                  spi_message_init(m);
68
69                  spi_message_init_with_transfers(m, &
                        transfers, 1);
70
71                  status= spi_sync(spi_device, m);
72
73              } while( (start_word[0] != 0xca) && (
                    start_word[1] != 0xfe) );
74
75              int dim;
76
77              dim = 6;
78
79              transfers.tx_buf = tx+2;
80
81              transfers.rx_buf = cmd_word;
82              transfers.len = dim;
83              transfers.delay_usecs = 10;
84
85              transfers.bits_per_word = 16;
86              transfers.speed_hz = spi_device->
                    max_speed_hz;
87              transfers.rx_nbits = SPI_NBITS_SINGLE;
88              transfers.tx_nbits = SPI_NBITS_SINGLE;
89
90              spi_message_init(m);
91
92              spi_message_init_with_transfers(m, &
                    transfers, 1);
93
94              status= spi_sync(spi_device, m);
95
96              tmp1 = cmd_word[4];
97              tmp2 = cmd_word[5];
98              cmd_word[4] = tmp2;
99              cmd_word[5] = tmp1;
100             memcpy(&count, cmd_word + 4, 2);
101
102             char result = (cmd_word[2] & 0b10000000);
103
```

```
104
105                 if(result){//read: the master needs to read,
                        so the slave has to send data
106                     tmp1 = cmd_word[2];
107                     tmp2 = cmd_word[3];
108                     cmd_word[2] = tmp2;
109                     cmd_word[3] = tmp1;
110                     memcpy(&address, cmd_word+2, 2);
111
112                     char *tx_buffer = kmalloc(2*count,
                            GFP_KERNEL);
113                     char *ricevuto = kmalloc(2*count,
                            GFP_KERNEL);
114
115                     int tmp_tx=0;
116                     while(1){
117                         if((count -64) > 0) {tmp_tx
                                = 64; count-= 64;} else {
                                tmp_tx = count; count
                                =0;}
118
119                         copy_from_user(tx_buffer,
                                user_buf+address, tmp_tx
                                *2);
120
121                         transfers.tx_buf = tx_buffer
                                ;
122
123                         transfers.rx_buf = ricevuto;
124                         transfers.len = tmp_tx*2;
125                         spi_message_init(m);
126                         spi_message_init_with_transfers
                                (m, &transfers, 1);
127
128                         status= spi_sync(spi_device,
                                m);
129
130                         address+=tmp_tx*2;
131                         printk("tmp_tx %d\n", tmp_tx
                                );
132
133                         if(count<=0) break;
```

```
134                     }
135                 }
136                 else {//write:the master writes, and the
                        slave receives data
137                     int tmp_recv;
138                     char *rx_buffer= kmalloc(2*count,
                            GFP_KERNEL);
139
140                     tmp1 = cmd_word[2];
141                     tmp2 = cmd_word[3];
142                     cmd_word[2] = tmp2;
143                     cmd_word[3] = tmp1;
144                     memcpy(&address, cmd_word+2, 2);
145
146                     int ptr=0;
147                     while(1){
148                         if((count - 64 )<0) {
                                tmp_recv = 2*count;} else
                                 {tmp_recv = 128; }
149
150                         transfers.tx_buf =
                                sendbuffer;
151
152                         transfers.rx_buf = rx_buffer
                                + ptr;
153                         transfers.len =tmp_recv;
154                         spi_message_init(m);
155 spi_message_init_with_transfers(m, &
                        transfers, 1);
156
157                         status= spi_sync(spi_device,
                                m);
158                         ptr+=tmp_recv;
159
160
161                         count -= (tmp_recv >> 1);
162
163                         if(count<=0) {break;}
164
165                     }
166
```

```
167                         int retv =copy_to_user(user_buf +
                                address, rx_buffer, ptr);
168
169                         kfree(rx_buffer);
170                 }
171                 tx_index=0;
172                 //the next transfer cleans the registers and
                        prepare the module for the next
                        comunication cycle.
173
174                 transfers.tx_buf = tx;
175
176                 transfers.rx_buf = start_word;
177                 transfers.len = 0;
178                 transfers.delay_usecs = 0;
179
180                 transfers.bits_per_word = 16;
181                 transfers.speed_hz = spi_device->
                        max_speed_hz;
182                 transfers.rx_nbits = SPI_NBITS_SINGLE;
183                 transfers.tx_nbits = SPI_NBITS_SINGLE;
184
185                 spi_message_init(m);
186
187                 spi_message_init_with_transfers(m, &
                        transfers, 1);
188
189                 status= spi_sync(spi_device, m);
190
191         }
192         kfree(m);
193 }
```

Il metodo riceve come parametro un indirizzo appartenente allo spazio utente. Questo sarà l'indirizzo di partenza alla quale poi verrà sommato un offset, ovvero il campo indirizzo che viene trasmesso durante una comunicazione. Una volta composto questo indirizzo, si possono effettuare le operazioni di lettura e scrittura.

Alle righe 38-48 si può notare un trasferimento, ma la lunghezza indicata è zero: infatti questa operazione è effettuata per realizzare un software reset del modulo, così da prepararlo al suo utilizzo (vedi sezione 3.2.4). A questo punto

il dispositivo è pronto per entrare in attesa. Una volta ricevuta una parola si controlla che questa equivalga alla parola di inizio trasferimento. Se la parola è quella giusta allora il trasferimento ha inizio. Il modulo si mette in attesa di 6 byte, corrispondenti ad una parola necessaria alla sincronizzazione dei due dispositivi e le due parole di comando. Alle righe 96-102 le parole sono interpretate. In caso il bit più significativo della prima parola di comando sia 1 si ha un'operazione di lettura (ovvero il master richiede una lettura e lo slave deve trasmettere i dati, a partire dall'indirizzo fornito nella prima parola di comando, nell'area di memoria riempita inizialmente), altrimenti si effettua una scrittura (il master invia i dati e lo slave li scrive nell'area di memoria allocata all'inizio, all'indirizzo fornito dalla prima parola di comando).

Dal momento che le code del dispositivo hanno capienza limitata a 64 parole, i trasferimenti avvengono a blocchi di 64 parole. Tutti i trasferimenti avvengono tramite la chiamata a spi_sync.

Una volta terminato un trasferimento si effettua un ulteriore software reset, tramite un trasferimento di dimensione zero. A questo punto il modulo attende una nuova parola iniziale.

Questo procedimento continua fino a quando la scheda non viene spenta o il programma che ha chiamato il metodo spislave_protocol non viene abortito.

3.5 I risultati ottenuti

Per testare il lavoro svolto sono stati creati alcuni file di prova, alcuni per testare la comunicazione della scheda lato master, altri la comunicazione lato slave. I file sono stati cross-compilati sfruttando la eSDK generata tramite Yocto e Bitbake, e destinata proprio al sistema utilizzato.

Per testare la comunicazione master sono stati implementati dei file in cui venivano inviati delle sequenze di parole alla centralina slave, che ha ricevuto i valori in modo corretto. Altre prove hanno dimostrato che i dati inviati dalla centralina sono interpretati correttamente dal modulo master. Inoltre per diverse parole di comando i test sono stati positivi e il modulo si comporta seguendo correttamente il protocollo implementato.

Riguardo alla comunicazione di tipo slave, è stato sviluppato un test che semplicemente chiama il metodo ioctl e quindi la funzione spislave_protocol. Inserendo degli opportuni ritardi tra una parola e l'altra e tra le parole di comando e le parole successive, da parte della centralina master, il modulo sviluppato risponde in modo corretto. La scheda sviluppata dall'azienda è molto più veloce a svolgere le varie operazioni quindi è stato opportuno inserire adeguati ritardi tra una parola e l'altra, altrimenti la scheda i.MX8 non riusciva a prelevare i dati ricevuti in tempo e di conseguenza non rispondeva opportunamente alle richieste.

Di seguito è riportato un esempio di file di test per il driver spi-master. In questo caso il driver è aperto, e si impone come modo di compionare i dati il modo SPI_CPHA. Dopo di che si riempie un vettore di carattari con il valore di un indirizzo nei primi due byte, e 4 parole negli otto byte successivi. Questi dati sono trasmessi tramite una chiamata al metodo write. Successivamente è invocato il metodo read, che invia l'indirizzo precedente e riceve 4 parole (situate all'indirizzo inviato, nella scheda HDF), che saranno copiate ai byte successivi del vettore di caratteri. Infine il driver viene chiuso.

```
1 #include <linux/spi-module.h>
2 #include <linux/spi/spi.h>
3
4 int main(int argc, char **argv)
5 {
6     char *name = "/dev/spi-master";
7     int fd;
8     unsigned char buf[64];
```

```
9
10      fd = open(name, O_RDWR);
11      if (fd < 0) {
12          perror("open");
13          return 1;
14      }
15
16      uint8_t mode = SPI_CPHA;
17      status = ioctl(fd, SPI_WR_MODE, &mode);
18
19      //ADDRESS
20      buf[0]=0xf8;
21      buf[1]=0x03;
22
23      //DATA
24      buf[2]=0x15;
25      buf[3]=0xff;
26      buf[4]=0x16;
27      buf[5]=0xff;
28      buf[6]=0x15;
29      buf[7]=0xff;
30      buf[8]=0x16;
31      buf[9]=0xff;
32
33      write(fd, buf, 4);
34
35      read(fd, buf, 4);
36
37      close(fd);
38      return 0;
39  }
```

Di seguito è invece riportato un esempio di test per spi-slave. Il driver come prima cosa è aperto, e tramite il metodo ioctl è impostato il modo di campionamento dei dati al valore SPI_CPOL. Un'area di memoria (referenzata dal vettore buffer) è allocata. Questa sarà l'area destinata alle operazioni del driver, quindi dove i dati saranno scritti, e dove saranno prelevati. Un'ulteriore chiamata al metodo ioclt, stavolta con parametro SLAVE_PROT, invoca il metodo spislave_protocol, che ascolta i vari messaggi provenienti dalla centralina master.

```
1  #include <linux/spi-slave.h>
2  #include <linux/spi/spi.h>
3
4  int main(int argc, char **argv)
5  {
6      char *name = "/dev/spi-slave";
7      int fd;
8
9      char *buffer=malloc(0xffffff*sizeof(char));
10
11     fd = open(name, O_RDWR);
12     if (fd < 0) {
13         perror("open");
14         return 1;
15     }
16
17     uint8_t mode = SPI_CPOL;
18     ioctl(fd, SLAVE_WR_MODE, &mode);
19
20     ioctl(fd, SLAVE_PROT, buffer);
21
22     free(buffer);
23     close(fd);
24     return 0;
25 }
```

3.6 La creazione della patch

Per questo progetto sono state effettuate numerose modifiche ai file del device tree (per includere i dispositivi SPI e abilitarli) e ai file del kernel (per includere il supporto per SPI slave). Le modifiche effettuate possono essere racchiuse in una patch, che permette di poterle applicare facilmente in un sistema equivalente con un unico passo.

Per creare la patch ed infine terminare le modifiche e cancellare la cartella Workspace, bisogna inizialmente includere i file modificati, tramite il comando "git add":

```
1 git add
```

In seguito, una volta inclusi tutti i file modificati, si esegue il comando dal percorso <build-dir>/workspace/sources/linux-imx:

```
1 git commit
```

Questo visualizza un elenco dei file modificati inclusi, e dei file modificati non inclusi. Dopo aver inserito un messaggio che identifica la patch, essa è pronta per essere creata nel layer scelto, e la cartella Workspace distrutta. Per la creazione della patch è stato creato il layer meta-new-patch nella cartella <build-dir>, chiamata build-imx.

```
1 devtool update-recipe linux-imx -a meta-new-patch
2 devtool reset linux-imx
```

La patch è stata creata in

build-imx/meta-new-patch/recipes-kernel/linux/linux-imx, con il nome del messaggio indicato in precedenza, quindi 0001-thesis_project.patch [23]

Capitolo 4

Conclusioni

Il protocollo sviluppato in questo progetto verrà utilizzato all'interno della scheda i.MX8QM lpddr4 arm2, per permettere la comunicazione con la centralina HDF sviluppata dall'azienda Ideas&Motion s.r.l. nel merito del progetto AutoDrive. L'obiettivo della tesi era quello di realizzare un protocollo che permettesse alle due schede la comunicazione, e un codice con il quale potessero scambiare le informazioni in modo chiaro e univoco. Il lavoro svolto ha realizzato gli obiettivi prefissi, e le due centraline possono scambiarsi dati correttamente nelle modalità stabilite, in entrambi i collegamenti, quindi sia quando il fusion processor si trova a ricoprire il ruolo master, che viceversa.

Ulteriori miglioramenti possono essere introdotti per rendere il sistema più veloce e quindi ottenere una migliore risposta alle richieste della centralina nel caso slave. Uno di questi miglioramenti è senz'altro l'introduzione del supporto per l'utilizzo del DMA. Il DMA si occupa del trasferimento dei dati, togliendo il suddetto carico alla CPU. Al momento il progetto non prevede questa possibilità. Si possono includere controlli di integrità dei dati, impostando delle opportune parole scelte, da trasmettere quando si ricevono i dati importanti, così da garantire che il trasferimento stia procedendo correttamente. Una modifica importante riguarda il protocollo lato slave, in cui per ridurre il tempo tra un messaggio e l'altro è possibile caricare all'avvio una sezione di memoria di spazio utente nello spazio kernel, e lavorare con i dati caricati. In questo modo si evita di caricare dati da spazio kernel a spazio utente o viceversa ad ogni messaggio. Possono essere introdotti limiti superiori ed inferiori riguardo le parole di comando, ovvero sul numero di parole da considerare e sugli indirizzi effettivi che possono essere scelti (dimensione del blocco di indirizzi accettabili, indirizzi vietati).

Bibliografia

[1] "Sito web di yocto." https://www.yoctoproject.org/software-overview/.

[2] "Openembedded." https://www.openembedded.org/wiki/Main_Page.

[3] "Panoramica di yocto." https://www.yoctoproject.org/docs/2.6.1/overview-manual/overview-manual.html.

[4] "La distribuzione di riferimento poky." https://www.yoctoproject.org/software-overview/reference-distribution/.

[5] "Lista layers openembedded." http://layers.openembedded.org/layerindex/branch/master/layers/.

[6] "Guida per lo sviluppo del bsp." https://www.yoctoproject.org/docs/2.6.1/bsp-guide/bsp-guide.html.

[7] "Yocto mega manual versione 2.1." http://www.yoctoproject.org/docs/2.1/mega-manual/mega-manual.html.

[8] *i.MX Yocto Project User's Guide Rev. L4.9.51$_i$mx8qm – beta*1. *NXPSemiconductors*, 2017.

[9] "L'interfaccia spi." https://www.laurtec.it/tutorial/elettronica-digitale/129-interfaccia-spi.

[10] *i.MX 8QuadMax Applications Processor Reference Manual Rev. E.* NXP Semiconductors, 2018.

[11] S. Venkateswaran, *Essential Linux Device Drivers.* PRENTICE HALL, 2008.

[12] "Tutorial spi." https://www.corelis.com/education/tutorials/spi-tutorial/.

[13] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers.* O'Reilly, 2005.

[14] "Yocto kernel developmant manual." https://www.yoctoproject.org/docs/2.6.1/kernel-dev/kernel-dev.html.

[15] "How to add a new layer and a new recipe in yocto." https://community.nxp.com/docs/DOC-331917.

[16] "Linux kernel, file spi.c, versione 4.9.62." https://elixir.bootlin.com/linux/v4.14.62/source/drivers/spi/spi.c.

[17] "Linux kernel, file spi.h, versione 4.9.62." https://elixir.bootlin.com/linux/v4.14.62/source/include/linux/spi/spi.h.

[18] G. Pan, "Linux kernel, file lpspi.c, versione 4.9.62." https://source.codeaurora.org/external/imx/linux-imx/tree/drivers/spi/spi-fsl-lpspi.c?h=imx_4.14.62_1.0.0_beta.

[19] "Linux kernel, file fsl-imx8qm.dtsi." https://source.codeaurora.org/external/imx/linux-imx/tree/arch/arm64/boot/dts/freescale/fsl-imx8qm.dtsi?h=imx_4.9.51_imx8_beta1.

[20] "Linux kernel, file fsl-imx8qm-lpddr4-arm2-lpspi.dts." https://source.codeaurora.org/external/imx/linux-imx/tree/arch/arm64/boot/dts/freescale/fsl-imx8qm-lpddr4-arm2-lpspi.dts?h=imx_4.9.51_imx8_beta1.

[21] "Documentazione su lpspi nel device tree." https://source.codeaurora.org/external/imx/linux-imx/tree/Documentation/devicetree/bindings/spi/spi-fsl-lpspi.txt?h=imx_4.14.98_2.0.0_ga.

[22] R. Bosch, *EMGS625 (SC900714AE) COMPONENT SPECIFICATIONS*. Bosch, 2015.

[23] "Yocto project - patching the source for a recipe." https://wiki.yoctoproject.org/wiki/TipsAndTricks/Patching_the_source_for_a_recipe.