UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Ingegneria

TESI DI LAUREA IN INGEGNERIA ELETTRONICA

Elaborazione in tempo reale su DSP di flussi audio in una rete wireless Bluetooth per motoveicoli

Candidato:

Fabio Sebastiano

Relatori:

Prof. Roberto Saletti

Prof. Roberto Roncella

Anno Accademico 2002/2003

Indice

Introduzione

Il Bluetooth è uno standard per la comunicazione wireless, nato per consentire la connessione tra i dispositivi più disparati, con particolare attenzione per quelli portatili. La tecnologia che implementa lo standard risulta a basso costo e a basso consumo: rappresenta l'ideale per la sostituzione dei cavi di collegamento con una connessione a radiofrequenza. Con l'ausilio del Bluetooth scompaiono i problemi relativi alla compatibilità dei connettori e all'ingombro dei cavi: ciò porta naturalmente ad applicare tale tecnologia in quelle situazioni in cui si desidera ottenere la massima comodità e libertà di movimento dell'utente. Da questo punto di vista, un'applicazione interessante risulta essere un sistema per la comunicazione a bordo di un motoveicolo. I due motociclisti hanno bisogno di poter comunicare tra loro e l'utilizzo di una connessione Bluetooth tra i loro caschi appare un'alternativa molto interessante rispetto all'uso di un cavo.

Si pensa allora di creare una rete Bluetooth tra i due caschi, risolvendo il problema della comunicazione tra pilota e passeggero. Per realizzare la rete viene utilizzata un'architettura con tre nodi: due posti sui caschi e uno sulla moto. In questo modo si possono aggiungere ulteriori potenzialità al sistema semplicemente collegando ad esso altri dispositivi presenti a bordo della moto: basti pensare al collegamento con un telefono cellulare per effettuare conversazioni telefoniche oppure alla possibilità di ascoltare le istruzioni di un navigatore satellitare o, più banalmente, la radio. Il sistema presentato dispone di queste funzionalità, che possono essere selezionate dall'utente attraverso la pressione di pochi tasti posti sul cruscotto.

Per quanto riguarda l'architettura, il Bluetooth impone che un nodo della rete abbia un ruolo privilegiato rispetto agli altri e diriga la connessione. Di conseguenza questo nodo ha bisogno di una potenza di calcolo maggiore, perché lo standard stabilisce che tutti i flussi audio passino da esso e vengano qui opportunamente miscelati. E' necessario allora che nel nodo centrale sia presente un altro processore collegato al dispositivo Bluetooth: quest'ultimo si preoccupa della connessione fisica e della gestione dei protocolli di comunicazione, mentre il processore si interfaccia con le sorgenti audio esterne e opera opportunamente sui flussi audio. La potenza di calcolo richiesta ha portato alla scelta di un DSP (Digital Signal Processor), per il quale è stata sviluppata un'applicazione per il controllo real-time della rete Bluetooth. L'applicazione si interfaccia con le sorgenti audio a bordo della moto, con gli utenti e con il dispositivo Bluetooth del nodo principale: ha così il controllo di tutte le risorse del sistema.

L'oggetto di questo lavoro è lo sviluppo di tale applicazione. La trattazione successiva è organizzata in 3 capitoli. Nel primo capitolo viene descritta l'architettura globale del sistema e sono individuati i requisiti dell'applicazione per il DSP. Il secondo capitolo è dedicato all'analisi della struttura del software e alla discussione delle scelte implementative compiute. Il terzo e ultimo capitolo affronta le problematiche relative alla messa a punto del sistema e riporta i risultati ottenuti.

Capitolo 1

Architettura del sistema

Questo capitolo descrive il sistema nel suo complesso: vengono date brevi descrizioni delle varie parti che lo compongono, facendo riferimento agli standard utilizzati (in particolare al protocollo Bluetooth). Particolare attenzione è dedicata al DSP, che costituisce la piattaforma hardware per l'applicazione descritta in dettaglio nel capitolo 2.

1.1 Specifiche del sistema

Il sistema che viene descritto in questo lavoro permette la gestione di una rete wireless che collega i caschi di due motociclisti con un punto di accesso principale posto sulla moto. L'instauarazione di tale rete rende disponibili agli utenti della moto una quantità di servizi audio. In particolare essa permette la comunicazione tra i due motociclisti (cioè implementa un classico interfono) ed inoltre dà la possibilità di interfacciare ciascun motociclista con altri dispositivi audio posti sulla moto, come ad esempio una radio, un cellulare o un navigatore satellitare; il motociclista è posto in grado di selezionare queste funzionalità grazie alla pressione di pochi tasti posti sul cruscotto.

Tutto ciò può essere implementato grazie all'esistenza di tecnologie a basso costo per la comunicazione wireless. In particolare lo standard Bluetooth

(descritto brevemente nel seguito) permette la creazione di reti wireless come quella richiesta dal nostro problema. Tale rete verrà realizzata usando degli appositi dispositivi Bluetooth che saranno presenti sia sui caschi sia sulla moto. La natura stessa del problema porta naturalmente all'adozione della seguente topologia: un nodo della rete sulla moto, che si interfaccia direttamente con gli altri dispositivi (radio, telefono cellulare e navigatore satellitare), e due nodi sui caschi.

Poiché il Bluetooth prevede delle reti di tipo centralizzato, in cui tutti i nodi comunicano con quello principale, sarà necessario assegnare ad un nodo la caratteristica di nodo *master* della *piconet* [1], dal quale dipenderanno i restanti due nodi. Tra le possibili alternative, la scelta migliore pare quella di assegnare il controllo della rete al nodo posto sulla moto; per tale nodo infatti non ci sono problemi di consumo di potenza come nel caso dei caschi ed è quindi possibile dotarlo di capacità elaborativa maggiore. Per questo nodo devono passare tutti i dati della rete ed esso deve inoltre provvedere a particolari miscelazioni dei flussi audio: esse sono necessarie in quanto l'utente potrebbe voler usare la modalità di interfono e nel contempo ascoltare la radio oppure potrebbe attivare l'interfono e il collegamento con il cellulare (si realizzerebbe così il meccanismo della chiamata a tre). Tutte queste caratteristiche implicano una opportuna dose di elaborazioni numeriche sui campioni dei segnali audio e, come verrà chiarito in seguito, il dispositivo Bluetooth utilizzato non possiede la potenza di calcolo necessaria per tali operazioni.

Si affianca quindi al dispositivo Bluetooth un DSP e si realizza quello che nel gergo dello standard è detto interfacciamento tra *host controller* e *host*, dove per host controller si intende il dispositivo che si preoccupa del controllo della rete e con host si identifica il DSP, destinato ad eseguire le elaborazioni in tempo reale sui diversi flussi audio e comunicante con l'host controller mediante una ben definita interfaccia a comandi/eventi. L'host risulta inoltre

[1] Una *piconet* è semplicemente una rete Bluetooth. Per maggiori delucidazioni vedere il paragrafo 1.2.

collegato in questo caso alle sorgenti audio esterne e all'interfaccia utente.

Per quanto detto, l'architettura del sistema, a cui si farà riferimento nel seguito, è quella illustrata in figura 1.1.

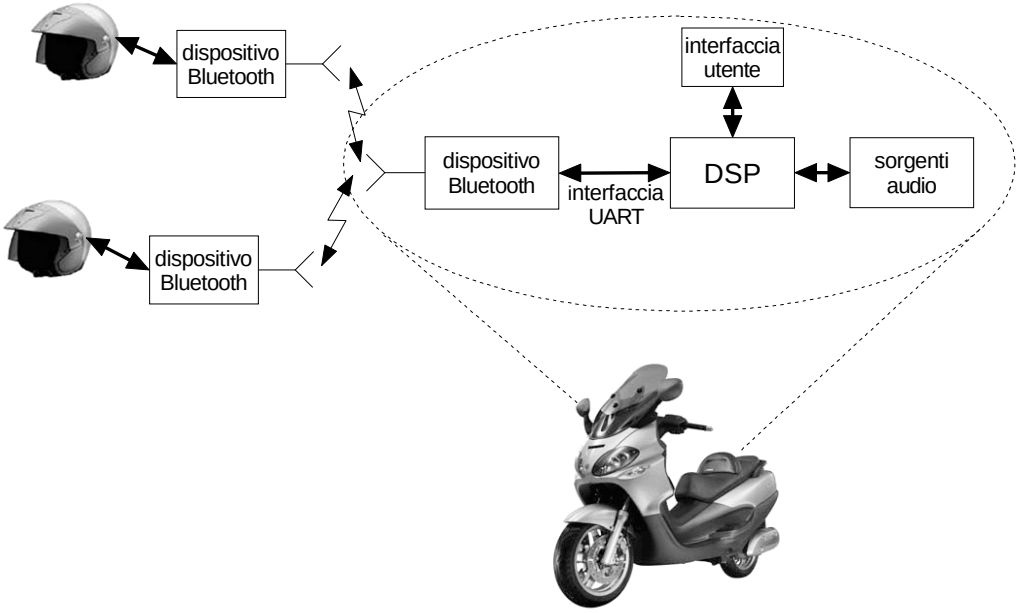

Figura 1.1: L'architettura del sistema

1.2 Cenni al protocollo Bluetooth

Bluetooth è uno standard per la comunicazione wireless a breve raggio, di basso costo, a bassa potenza, operante nella banda ISM (Industrial Scientific Medical). E' stato sviluppato a partire dal 1994 e la definizione delle specifiche è gestita da un gruppo di imprese, il Bluetooth Special Interest Group (BSIG), di cui fanno parte Ericsson Mobile Communications, Intel corporation, IBM corporation, Toshiba corporation, Nokia Mobile Phones, Microsoft, Lucent, 3Com, Motorola.

La tecnologia Bluetooth si presenta come un'alternativa al gran numero di cavi necessari alla connessione di una varietà di dispositivi, come notebook, PDA (Personal Digital Assistant), telefoni cellulari e simili. Inoltre essa

permette la realizzazione di un'interfaccia uniforme per l'accesso a risorse comuni: si pensi alla possibilità di connettersi con un qualsiasi dispositivo personale ad un punto di accesso di una rete locale.

Il Bluetooth usa una trasmissione FHSS (Frequency Hopping Spread Spectrum) a 1.600 hop/s sfruttando le frequenze (2402 + k) MHz, con k=1..79, che vengono occupate secondo una sequenza pseudo-casuale. La modulazione usata è una GFSK (Gaussian Frequency Shift-Keying) binaria con una baud rate di 1 Msymbol/s. Le specifiche indicano anche la potenza trasmessa, che deve necessariamente essere ridotta (da 0 dBm a 20 dBm) dato che la banda ISM è utilizzabile senza licenza a patto di trasmettere a bassa potenza. E' utilizzato un canale TDD (Time Division Duplex) per una comunicazione full-duplex, con slot temporali di lunghezza nominale di 625 μs. Ogni pacchetto è trasmesso su un diverso salto di frequenza ed è generalmente contenuto in un unico slot, anche se è possibile trasmettere pacchetti occupanti 3 o 5 slot. Bluetooth supporta un canale dati asincrono, un massimo di tre canali simultanei per l'audio o un canale che supporta simultaneamente dati asincroni e audio sincrono. Ogni canale audio supporta flussi full-duplex a 64 kbps; il canale asincrono supporta un massimo unidirezionale di 732,2 kbps (e 57,6 kbps nella direzione inversa) o una trasmissione simmetrica a 433,9 kbps.

1.2.1 La piconet Bluetooth

I dispositivi Bluetooth comunicano tra loro formando delle piccole reti, le *piconet*, che includono un dispositivo chiamato *master* della piconet e un massimo di altri 7 dispositivi detti *slave*, che comunicano attivamente con il master. Ogni dispositivo può fungere indifferentemente da master o da slave e il suo ruolo viene deciso nell'ambito di un protocollo di comunicazione che viene eseguito all'instaurarsi di una piconet, come specificato nel seguito. Oltre agli slave possono essere presenti nella piconet altri dispositivi che non sono momentaneamente attivi ma in modalità *parked* (vedi sezione 1.2.2). Inoltre ogni dispositivo può far parte di più piconet: si parla in questo caso

di *scatternet*, reti Bluetooth composte da più piconet che condividono dei dispositivi (notare che lo slave di una piconet può anche essere master di un'altra piconet).

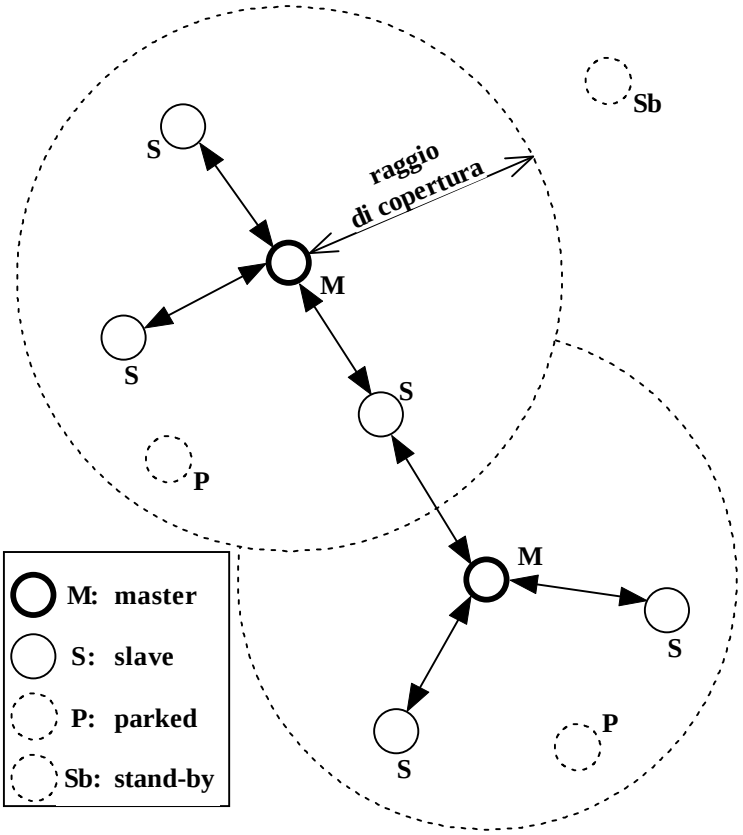

Figura 1.2: Le piconet Bluetooth

In ogni piconet il canale di comunicazione è individuato da una particolare sequenza di salti frequenziali che si succedono ad intervalli di 625 μs. In ognuno di questi slot temporali viene trasmesso un pacchetto o parte di esso nel caso di pacchetti multi-slot; il master e ciascuno slave comunicano a slot alterni, in modalità TDD. La sequenza di salto è individuata dal master e ciascuno slave deve sincronizzarsi ad esso: per questo si utilizza il Bluetooth *clock*, un orologio di sistema a 28 bit presente in ogni apparato Bluetooth; in questo modo gli slave devono conoscere l'offset del proprio clock rispetto a

quello del master. Si noti come sia il master ad individuare in modo univoco la piconet.

Il processo che porta alla creazione di una piconet comporta diverse procedure: l'*inquiry*, il *paging* e lo *scan*.

Inquiry E' un processo attraverso il quale il master di una futura piconet cerca altri dispositivi nelle proprie vicinanze e richiede, tra le altre cose, l'indirizzo hardware di ciascun dispositivo, che è essenziale per la connessione.

Scan E' la procedura operata dagli slave che cercano sulle diverse frequenze un messaggio di inquiry da parte di un potenziale master.

Paging E' lo scambio di informazioni tra master e slave che permette la sincronizzazione e la creazione del canale sul quale avverrà la comunicazione.

1.2.2 Il risparmio energetico

I dispositivi Bluetooth possono assumere diverse modalità a basso consumo quando non sono impegnati in una comunicazione attiva.

Nella modalità *sniff* uno slave si accorda con il master per ascoltare periodicamente le trasmissioni del master.

Nella modalità *hold* un dispositivo interrompe la comunicazione con il proprio partner nella piconet per un dato periodo di tempo.

Nella modalità *park* uno slave esce dalla piconet fino a nuovo avviso. Il dispositivo rilascia l'indirizzo che gli era stato assegnato come componente attivo della piconet e periodicamente ascolta le trasmissioni del master; in seguito può essere invitato nuovamente dal master a rientrare nella piconet.

Nonostante queste modalità siano state espressamente progettate per il risparmio energetico, i dispositivi in queste modalità possono assolvere ad altri compiti, come operare procedure di inquiry o partecipare attivamente ad altre piconet.

1.2.3 Lo stack di protocollo e i profili

Lo sviluppo di applicazioni Bluetooth prevede l'uso di una serie di protocolli definiti dallo standard e organizzati nel *protocol stack*. Applicazioni differenti possono usare protocol stack differenti ma ognuno di questi usa lo stesso *layer* per la connessione fisica e lo scambio di dati; infatti lo stack è stato progettato per favorire il riutilizzo a livelli più elevati di protocolli esistenti, "invece di ri-invetare ogni volta la ruota" [Mettala 99]. Lo stack può essere partizionato in quattro livelli:

- Bluetooth Core Protocols: Baseband, LMP, L2CAP, SDP
- Cable Replacement Protocol: RFCOMM
- Telephony Control Protocols: TCS Binary, AT-commands
- Adopted Protocols: PPP, UDP/TCP/IP, OBEX, WAP, vCard, vCal, IrMC, WAE

Baseband

Questo protocollo si occupa di stabilire la connessione fisica RF tra le unità di una piconet. Poiché viene usata una trasmissione FHSS in cui i pacchetti sono trasmessi in intervalli temporali definiti a frequenze fissate, questo layer usa le procedure di inquiry e paging per sincronizzare i clock dei dispositivi e la frequenza di trasmissione.

Audio

Occorre qui notare che l'audio si appoggia direttamente sul baseband e ciò semplifica la creazione e la gestione di una connessione audio.

Link Manager Protocol

Il LMP è il responsabile della connessione tra due dispositivi. Ciò include aspetti come la sicurezza (autenticazione e cifratura grazie alla generazione,

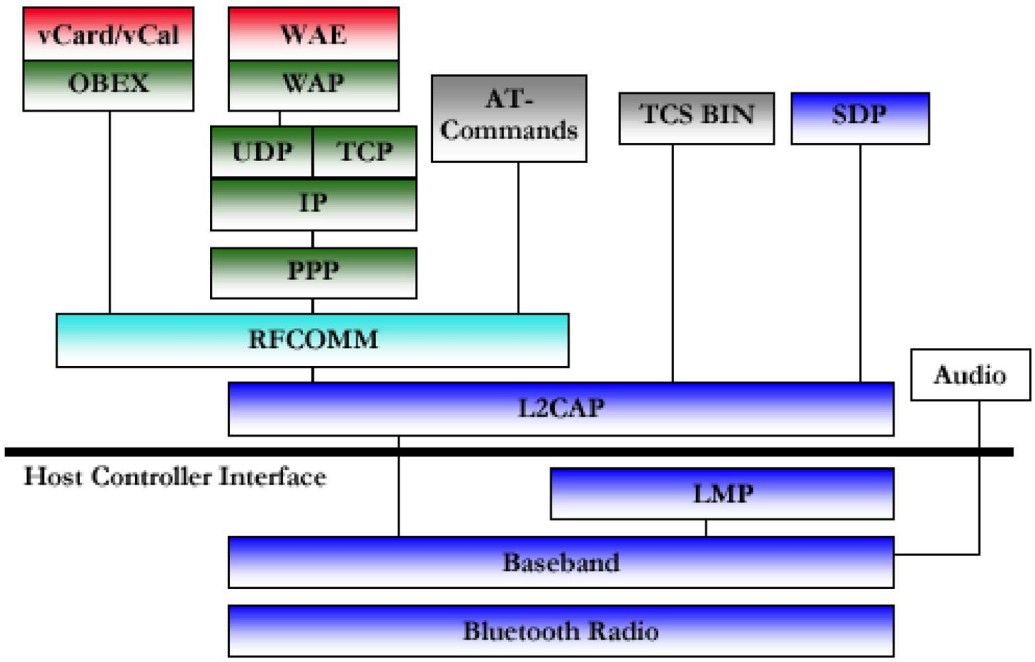

Figura 1.3: Bluetooth Protocol Stack [Mettala 99]

lo scambio e il controllo di chiavi crittografiche) e il controllo della dimensione dei pacchetti del baseband.

Logical Link Control and Adaptation Protocol

Il L2CAP è necessario per adattare i protocolli di livello più elevato sul baseband: esso infatti offre funzionalità più avanzate rispetto al baseband, quali la possibilità di gestire pacchetti di tipo L2CAP, che possono essere molto più lunghi rispetto ai pacchetti realmente trasmessi in aria.

Service Discovery Protocol

E' il protocollo che implementa alcune caratteristiche fondamentali del Bluetooth: la richiesta di informazioni sui dispositivi, l'interrogazione sui servizi disponibili e le relative caratteristiche. Solo dopo queste procedure può essere stabilita una connessione.

Inoltre in figura 1.3 è indicato l'HCI (Host Controller Interface) che fornisce un'interfaccia di comando per il baseband controller e il link manager. Essa riveste una certa importanza per il presente lavoro e alcuni aspetti dell'HCI saranno approfonditi nel capitolo 2.

Gli altri livelli sono necessari per applicazioni più evolute rispetto a quella descritta e di conseguenza non vengono analizzate. E' interessante però aggiungere che RFCOMM è il layer che implementa l'emulazione di una porta seriale, una delle funzionalità per la quale è nata la tecnologia Bluetooth.

Le specifiche comprendono, oltre i protocolli di comunicazione, un insieme di applicazioni che permettono di semplificare la progettazione e la interoperabilità tra i vari dispositivi. Le specifiche di queste semplici ma utili applicazioni sono chiamate *profili*. Tutti i profili discendono dal Generic Access Profile (GAP) che definisce le regole base e le condizioni per realizzare connessioni e creare canali L2CAP. Non ci addentriamo nella descrizione dei vari profili: ci limitiamo a mostrarne la gerarchia nella fig. 1.4 e a far notare che l'*headset profile* presente in figura è quello usato nel nostro caso per la gestione dei caschi. Esso infatti è un profilo creato appositamente per la gestione di auricolari (come quelli usati nei cellulari) ed è esattamente ciò che è richiesto dalla applicazione in esame.

1.3 Il dispositivo Bluetooth

Vengono ora date alcune informazioni sui dispositivi Bluetooth usati per lo sviluppo del sistema. Per l'implementazione del progetto è stato scelto il Bluecore 2 di CSR (Cambridge Silicon Radio), un sistema integrato su singolo chip con tecnologia CMOS a 0.18 μm. Questo componente è stato usato in fase di sviluppo montato all'interno di un modem Casira, una evaluation board che permette l'alloggiamento di un chip Bluecore e che è inoltre dotata di tutti quei componenti ausiliari che permettono il funzionamento del Bluecore stesso e che facilitano lo sviluppo delle applicazioni. Esso

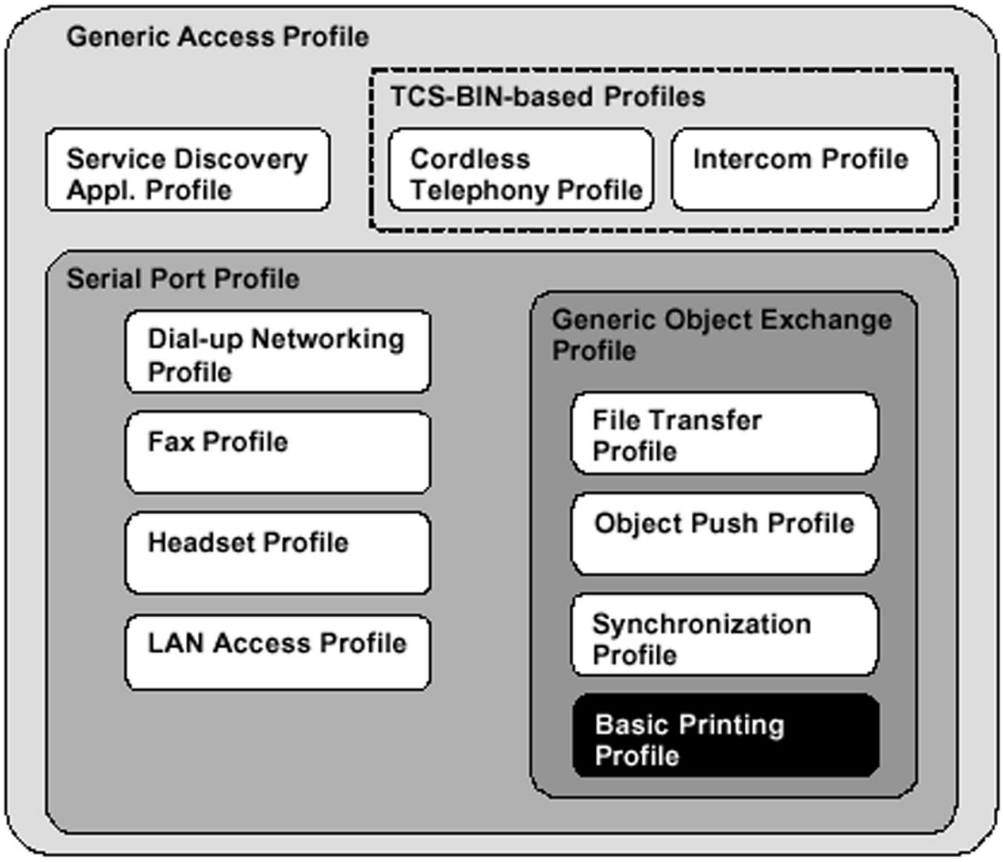

Figura 1.4: Bluetooth Profiles [Profile]

contiene un particolare alloggiamento nel quale vengono poste delle piccole schede contenenti il chip Bluecore, l'antenna, un oscillatore di riferimento e una memoria flash; la circuiteria ausiliaria comprende l'alimentazione, alcuni codec audio, circuiti di interfacciamento con la porta UART e con la porta USB. Il processore provvede all'esecuzione dello stack di protocollo Bluetooth, utilizzando per questo scopo 6 degli 8 MIPS disponibili; come accennato all'inizio del capitolo, la potenza di calcolo residua non è sufficiente per svolgere tutti i compiti richiesti al sistema e ciò richiede l'utilizzo del DSP.

Il chip BlueCore costituirà quello che è stato precedentemente chiamato host controller; sarà posto sulla moto e sarà collegato con il DSP attraverso

un'interfaccia UART e con i dispositivi Bluetooth sui caschi via radio. Per quanto riguarda i caschi, è possibile usare una coppia qualsiasi di auricolari commerciali che siano compatibili con l'*headset profile* (vedi paragrafo 1.2.3). Ad esempio durante lo sviluppo del software è stato possibile eseguire dei test usando alcuni auricolari EZClip della SkyPower; in questo modo i problemi relativi all'adattamento degli auricolari sui caschi o all'uso di caschi già dotati di auricolari sono stati rimandati ad una fase successiva dello sviluppo del sistema.

1.4 L'ADSP-2191

Il processore utilizzato insieme al BlueCore è l'ADSP-2191, un processore single-chip in virgola fissa prodotto dalla Analog Device e ottimizzato per l'elaborazione digitale dei segnali (DSP) e altre applicazioni che richiedono elaborazioni numeriche ad alta velocità; la sua potenza massima di calcolo è infatti di 160 MIPS.

Il Core

Il nucleo del processore contiene tre unità computazionali indipendenti: la ALU, il moltiplicatore/accumulatore (MAC) e lo shifter. Esse compiono operazioni sui dati contenuti nei registri a 16 bit del processore[2]: l'ALU si occupa delle operazioni logiche e aritmetiche, il MAC compie moltiplicazioni, moltiplicazioni/addizioni e moltiplicazioni/sottrazioni, lo shifter permette shift logici e aritmetici e operazioni sugli esponenti. Il DSP dispone anche di primitive per la divisione che permettono di ottenere quoziente e resto su 16 bit in 16 cicli di clock.

[2]Il MAC e lo shifter hanno ciascuno anche un accumulatore a 40 bit diviso in tre parti, ciascuna delle quali accessibile in un singolo ciclo.

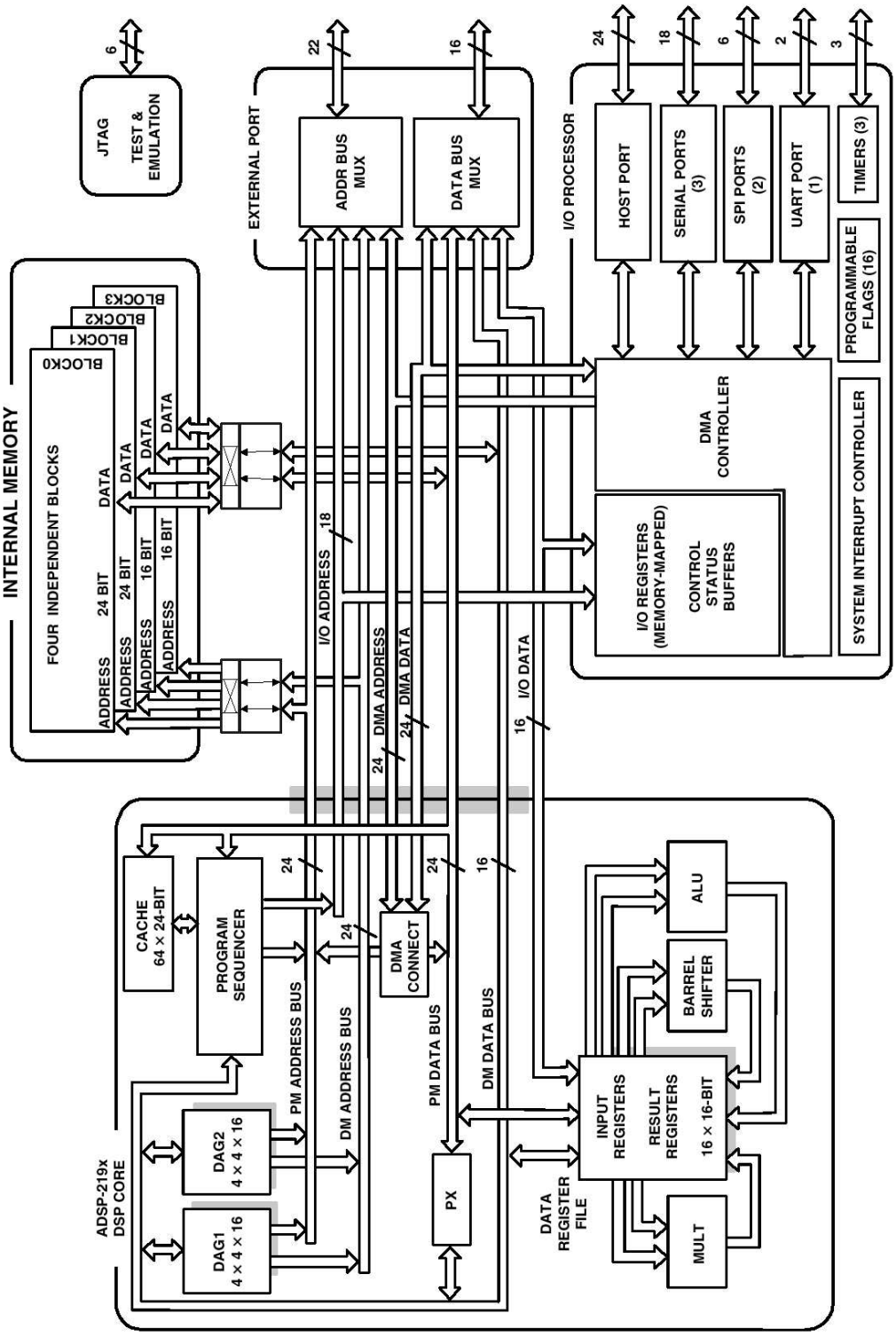

Figura 1.5: Architettura dell'ADSP 2191

L'architettura

L'architettura è di tipo Harvard, con due bus[3] e due memorie per i dati e il programma; l'architettura tuttavia si discosta da una Harvard "pura" per alcuni modi di utilizzo. Infatti è possibile caricare dei dati anche in memoria di programma (PM) e sfruttare la presenza di due bus per operare due trasferimenti dalla memoria in parallelo. Ciò è possibile grazie alla presenza di *multifunction instruction* che permettono l'esecuzione simultanea di più istruzioni semplici: in genere consentono di trasferire un dato dalla/alla PM, trasferire un dato dalla/alla DM (memoria dati) e operare un'elaborazione con una delle unità computazionali; il tutto in un singolo ciclo di clock.

L'accesso alla memoria avviene attraverso due *data address generator* (DAG) che generano gli indirizzi; essendo due permettono un doppio accesso alla memoria. Ognuno contiene dei registri puntatori a 16 bit che permettono l'indirizzamento indiretto con post-modifica o pre-modifica con il valore di uno dei quattro registri di modifica. Esistono anche coppie di registri base e lunghezza che permettono l'indirizzamento di buffer circolari. Una caratteristica molto utile, soprattutto perché elimina l'overhead dovuto al salvataggio dei registri in caso di interrupt o chiamata a subroutine, è la presenza di DAG e registri secondari; con opportune istruzioni assembler è possibile cambiare il banco di registri e i DAG che si stanno usando[4], senza sporcare così i registri e i DAG primari e permettere così un più veloce servizio degli interrupt.

La memoria

Lo spazio unificato di memoria (dati e programma) consiste di 16M locazioni accessibili attraverso i due bus a 24 bit DMA e PMA. Questo spazio è diviso in

[3]Bisogna ricordare che accanto a questi due bus dati ce ne sono altrettanti per l'indirizzamento, il *Program Memory Address Bus*(PMA bus) e il *Data Memory Address Bus* (DMA bus). Inoltre ci sono due bus, dati e indirizzi, per il controllore DMA.

[4]E' possibile cambiare il banco di registri cui si fa riferimento e i DAG in un singolo ciclo di clock usando la multifunction instruction `ENA SEC_REG,ENA SEC_DAG;`.

256 pagine da 64k parole (da pagina 0 a pagina 255). L'ADSP-2191 dispone di 64k parole di memoria SRAM sul chip, che risulta posizionata alla pagina 0 ed è divisa in 4 blocchi da 16k, di cui due con parole da 24 bit e due con parole da 16 bit. E' inoltre possibile aggiungere memoria esterna, indirizzata dai piedini `MS3-0` che selezionano uno dei quattro banchi di memoria possibili (pagine 1-63, 64-127, 128-191, 192-254). La boot-ROM da 1k sul chip occupa invece la parte alta della pagina 255.

Lo spazio di I/O è separato da quello di memoria: le pagine sono lunghe 1K e quelle da 0 a 7 sono riservate alle periferiche interne del processore.

Le periferiche

Il DSP ha una porta a 16 bit per l'interfacciamento con processori host che possono leggere direttamente sulla sua memoria. Dispone inoltre di un'interfaccia per la memoria esterna.

Ci sono poi diverse porte per la comunicazione con l'esterno, che costituiscono un punto di forza per questo DSP. Esso è infatti uno dei pochi che dispone di una porta UART, una periferica necessaria nel sistema in esame per l'interfacciamento con il Bluecore. Il DSP dispone inoltre di tre porte seriali sincrone (`SPORT0-2`). Le interfaccie possono essere gestite ad interruzione, ma anche in modo più vantaggioso tramite il controllore DMA, che dispone di due bus dedicati per dati e indirizzi e svincola il processore dall'onere di brutali trasferimenti di dati.

Il controllore d'interruzione può rispondere fino a 17 diversi interrupt: tre interni (stack, emulator kernel e power-down), due esterni (emulator e reset) e dodici generati dalle periferiche ma completamente configurabili dall'utente.

Sono presenti tre timer che generano interrupt periodici e possono essere inoltre usati per la generazione di forme d'onda e il conteggio di eventi esterni.

Il DSP dispone inoltre di 16 pin general-purpose I/O (GPIO), che possono essere programmati come input o output. Otto di essi sono pin *General Purpose Programmable Flag* dedicati. Gli altri otto sono multifunzionali, agendo come GPIO quando il DSP è connesso ad un bus dati a 8 bit e

agendo come pin per gli otto bit più significativi del bus dati esterno quando questo è a 16 bit. I pin Programmable Flag possono implementare interrupt sensibili al livello o al fronte; lo stato di alcuni può essere usato come test per l'esecuzione di istruzioni condizionali.

Una periferica ampiamente sfruttata durante la messa a punto del software è stata l'interfaccia JTAG. Il JTAG (Join Test Action Group) è uno standard IEEE (si veda [JTAG]) per il test di circuiti integrati: esso permette di testare il funzionamento senza l'utilizzo di uno specifico hardware per il test. Ciò è reso possibile grazie al *boundary scan* definito dallo standard: tutti i pin di input e output del dispositivo sono dotati di latch, che, appositamente istruiti, campionano il segnale in uscita o impongono quello in ingresso; questi latch sono tra loro collegati come uno shift register, creando in questo modo un'interfaccia parallelo-seriale tra i pin del chip e una porta seriale; quest'ultima è utilizzata per ricevere i dati o mandare i dati per l'esecuzione dei test. Questo sistema è controllato da un'opportuna circuiteria, che crea un'interfaccia tra l'esterno e la periferica per il test; i test vengono infatti eseguiti mandando opportune istruzioni ai piedini esterni del JTAG, che indicano il tipo di test da eseguire o le locazioni di memoria e i registri che interessa leggere. Tutto ciò rende possibile operare un debug del software direttamente sul processore, dando la possibilità di leggere direttamente il contenuto dei registri, sia quelli delle periferiche sia quelli del core.

1.4.1 L'EZ-KIT Lite

L'applicazione per il DSP è stata sviluppata e testata usando un kit di sviluppo della Analog Device, l'EZ-KIT Lite. In questo kit sono inclusi l'ambiente di sviluppo VisualDSP++ e una Evaluation Board con l'ADSP-2191 e alcuni componenti hardware che ne permettono il funzionamento.

In particolare la scheda comprende:

- Un ADSP-2191 con clock a 160MHz e un selezionatore per il fattore di moltiplicazione del clock

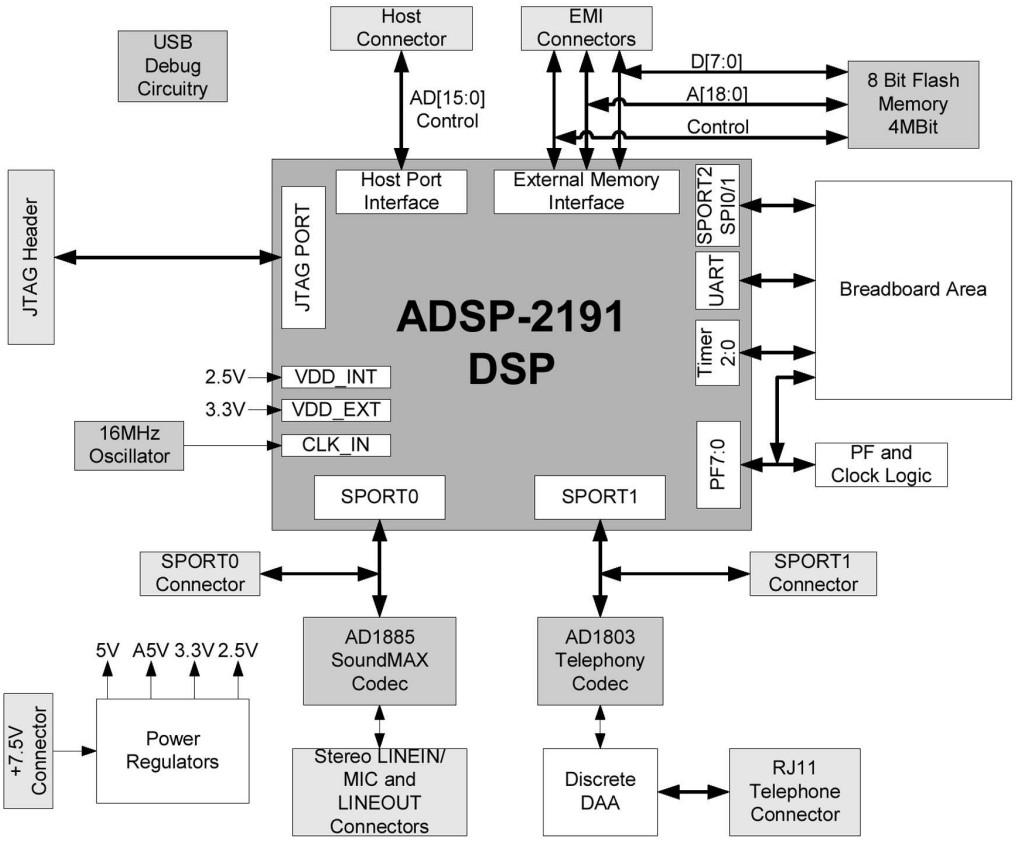

Figura 1.6: La struttura dell'evaluation board dell'EZ-KIT Lite

- Un'interfaccia USB per il debug via PC
- Un codec Analog Device AD1885 48 kHz AC'97 SoundMax®
- Un codec Analog Device AD1803 Low-Power Modem Codec
- 8 x 512K di memoria flash
- Connettori vari:
  - un connettore a 14 pin per l'interfaccia JTAG
  - connettori per la SPORT0 e la SPORT1
  - connettori per l'interfaccia con la memoria esterna

  - connettore per l'interfacciamento con la Host Port

- 2 pulsanti e 4 led collegati ai GPIO
- Regolatori di tensione della Analog Device ADP3338 e ADP3339
- Breadboard area per l'aggiunta di ulteriori connettori o integrati

1.4.2 Il Visual DSP Kernel

E' opportuno introdurre in questa sede alcune caratteristiche del sistema di sviluppo usato, affinché possa risultare chiaro il contenuto del capitolo 2 che illustra nel dettaglio il software sviluppato. Infatti il VisualDSP++ include il nucleo di un sistema operativo, il Visual DSP Kernel (VDK), sul quale è stata costruita l'intera applicazione.

Come noto dalla teoria dei sistemi operativi, il nucleo (o kernel) aggiunge alcune funzionalità alla macchina hardware, in modo che l'unione di macchina hardware e nucleo possa essere considerata come una macchina astratta, le cui capacità includono quelle della macchina hardware. In aggiunta, la macchina astratta presenta fondamentalmente due nuove caratteristiche: l'implementazione di processori virtuali e la presenza di meccanismi di sincronizzazione tra questi processori. Questo rende possibile l'esecuzione di più *processi* in parallelo che, nonostante vengano eseguiti su processori virtuali distinti, devono necessariamente girare sul processore reale, che è nel nostro caso unico.

Il kernel implementato sull'ADSP-2191 fornisce le funzionalità suddette, che quindi risulteranno *primitive* per lo sviluppatore. Questi ha la possibilità di implementare il multitasking nella propria applicazione in maniera semplice: il VDK, infatti, definisce una classe di tipo `VDK::thread` per la quale sono definite delle funzioni membro per l'inizializzazione, per l'esecuzione (cioè il programma vero e proprio di cui il thread è un'istanza) e per la gestione degli errori. I thread[5] possono essere creati automaticamente all'avvio

[5]Precedentemente si è parlato di processi, ma il sistema usato supporta solo i thread.

dell'esecuzione o possono essere istanziati dinamicamente, caratteristica che fornisce un'elevata flessibilità dato che possono essere anche istanziati più thread dello stesso tipo, ossia che condividono lo stesso codice.

Per quanto riguarda i meccanismi di sincronizzazione, il VDK offre diverse soluzioni: i semafori, i messaggi e gli eventi. I semafori sono i classici semafori presenti in ogni sistema operativo, su cui i thread possono eseguire due operazioni fondamentali, la `Pend` e la `Post`[6]. In questo modo possono facilmente essere realizzati punti in cui due thread si sincronizzano o intervalli temporali in cui agiscono in mutua esclusione. Consideriamo ad esempio il caso in cui vogliamo proteggere l'accesso ad una risorsa condivisa (tipicamente una variabile globale). Il semaforo è costruito come una classe che ha come membro privato un contatore, che inizializziamo ad 1 con il costruttore; se racchiudiamo ogni sezione del codice in cui viene usata la risorsa condivisa tra una coppia di istruzioni `Pend` e `Post`, abbiamo reso l'accesso alla risorsa mutuamente esclusivo, come si può facilmente verificare: in questo modo un solo thread alla volta può accedere alla risorsa, evitando gli errori che potrebbero scaturire da un accesso simultaneo.

Gli eventi sono una sorta di variabili booleane che possono causare il blocco di un thread in attesa che il loro valore risulti vero. Ogni evento viene posto pari all'AND o all'OR logici di una serie di bit, gli *Event bit*. Ognuno di essi può essere settato o resettato in qualunque parte del codice dell'applicazione e ciò influenza il valore degli eventi collegati ad essi. I thread possono eseguire l'istruzione `VDK::PendEvent()` che blocca la loro esecuzione

La differenza sostanziale tra i due è che ogni processo dispone di uno spazio virtuale di memoria privato, a cui gli altri processi non possono accedere. Ciò implica una complessa gestione della memoria e soprattutto un maggiore spreco di risorse nelle commutazioni di contesto, in quanto occorre cambiare ogni volta lo spazio di memoria virtuale al quale si fa riferimento. I thread hanno comunque un contesto di esecuzione privato come i processi, che include lo stack e il contenuto dei registri, ma al contrario di questi ultimi essi accedono tutti allo stesso spazio di memoria, semplificando l'uso di risorse condivise e la commutazione di contesto.

[6]Sono quelle operazioni che in altri contesti sono chiamate rispettivamente Wait e Signal.

fino a quando l'evento passato come parametro a tale funzione non risulta vero.

I messaggi sono segnali che un thread manda ad un altro, il cui payload è l'indirizzo di una zona di memoria in cui risiedono i dati che occorre scambiare.

Inoltre il VDK, offre la possibilità di scrivere dei *Device Driver* per la gestione dei dispositivi di I/O; ad essi sono legati i *device flag*, eventi il cui stato è regolato dai device driver. Il cuore di ogni Device Driver è la *dispatch function* che viene richiamata quando un thread vuole accedere alla risorsa gestita dal driver e che deve gestire anche le routine di interrupt relative all'I/O considerato. Gli interrupt, infatti, sono ancora visibili a questo livello, che risulta appena al di sopra del nucleo. E' necessario che il programmatore implementi le proprie routine di interrupt, che nel VDK devono essere scritte in Assembly; ciò per far si che il codice più critico da un punto di vista temporale (che dovrebbe essere quello delle routine di interrupt) sia eseguito con efficienza massima. Il resto del codice invece viene realizzato in C++, permettendo la gestione di problemi più complessi che non potrebbero essere risolti con il solo Assembly. Secondo il VDK, i device driver dovrebbero proprio svolgere la funzione di ponte tra i due domini dei thread e degli interrupt, anche se si può farne tranquillamente a meno[7].

In ogni sistema operativo riveste particolare importanza l'algoritmo (algoritmo di schedulazione) che decide quale processo deve essere eseguito dal processore reale ad un determinato istante. Per gestire più processori virtuali il nucleo provvede infatti a far eseguire alternativamente i processi, operando delle commutazioni di contesto. Queste sono delle operazioni con le quali si passa dall'esecuzione di un processo a quella di un altro; esse richiedono il salvataggio del contenuto dei registri e dello stack del primo processo e il caricamento dei valori dei registri e lo stack necessari al secondo processo per l'esecuzione.

[7]In alcuni contesti, come il nostro, i device driver costituirebbero solo un'inutile aumento della complessità dell'applicazione, come chiarito più avanti nel lavoro.

Il VDK dispone di un meccanismo di schedulazione di tipo *preemptive*, in cui ad ogni thread è assegnata una priorità, che può essere cambiata dinamicamente. Secondo quest'approccio un thread ha il controllo del processore reale finchè risulta il thread a maggiore priorità nella coda dei thread pronti (cioè che non sono bloccati su un semaforo, un evento o altro); la preemption avviene quando il thread pronto a priorità maggiore diviene un altro. Comunque sono disponibili anche altre metodologie di scheduling. Nel *cooperative scheduling* tutti i thread hanno la stessa priorità e la commutazione di contesto avviene quando il thread in esecuzione richiama la funzione `Yield()`: in questo modo va in esecuzione il primo thread della coda dei pronti e quello rimosso va in fondo a tale coda. Nel *round-robin scheduling* i thread continuano ad avere la stessa priorità, ma vengono loro assegnati degli slot temporali fissi per l'esecuzione.

E' possibile inoltre rendere *atomiche* alcune sezioni di codice, impedendo lo scheduling al loro interno; si ricorre ad una *Unscheduled Region* in cui lo schedulatore non viene richiamato o ad una *Critical Region* in cui vengono anche disabilitati gli interrupt.

1.5 L'interfaccia utente

Si è affermato che l'utente deve comunicare con il sistema attraverso una semplice interfaccia, di utilizzo facile e veloce. Questa è realizzata come in figura 1.7, dove è chiara la funzione dei led e dei pulsanti. Il pannello di comando è suddiviso in due parti: quella superiore permette la configurazione delle opzioni per il pilota (telefonata, ascolto del navigatore satellitare, radio), quella inferiore è riservata al passeggero. Il pulsante intercom serve per abilitare l'interfono mentre switch cambia la corrispondenza tra i caschi e le due righe dell'interfaccia. I led, rappresentati dai cerchi bianchi, si accendono per indicare la modalità di funzionamento: quelli della metà superiore corrispondono naturalmente al pilota, gli altri al passeggero; il led in basso a destra indica invece se il sistema è acceso oppure no.

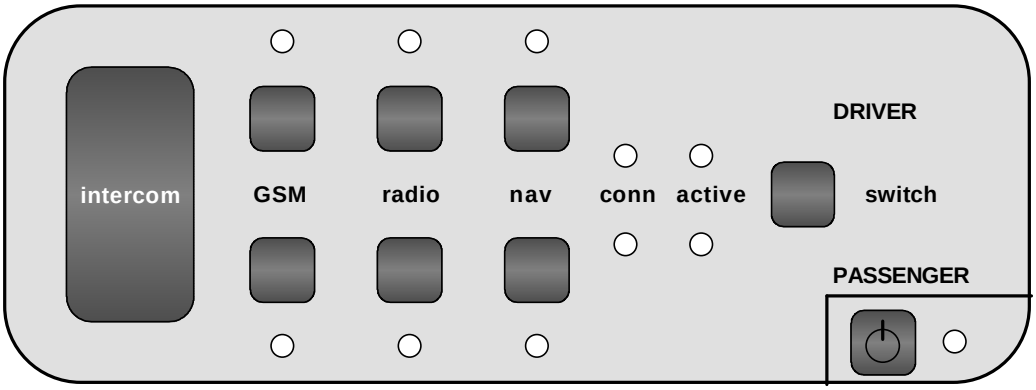

Figura 1.7: Il pannello di controllo

La gestione dell'interfaccia è interamente assegnata al DSP che è ad essa collegata attraverso alcuni GPIO. In figura 1.8 è riportato lo schema elettrico, che è necessario per capire come il DSP deve pilotare i GPIO per un corretto funzionamento.

L'integrato HEF4894B è un registro a scorrimento a 12 bit, collegato a 12 latch le cui uscite open-drain permettono la lettura dello stato dei pulsanti e l'accensione dei led. I cinque terminali STRo, Dout, CLKo, CLKi e Din sono collegati ad altrettanti GPIO che sono configurati opportunamente come input o come output. L'integrato collegato ai led provvede alla loro accensione: attraverso i terminali Dout e CLKo (rispettivamente collegati all'ingresso dello shift register e al piedino per il clock) viene caricata una parola di 12 bit; viene dato poi un segnale di strobe su STRo, attivando le uscite. Risultano allora accesi i led collegati alle uscite che sono a tensione bassa, cioè quelle per cui il bit corrispondente nel registro vale 1.

Per leggere i pulsanti deve essere caricata una parola nel secondo shift register usando Dout e CLKi. Affinché sia possibile leggere lo stato di un pulsante, il terminale dell'integrato collegato al particolare pulsante deve essere a tensione bassa, mentre gli altri terminali devono essere in alta impedenza (per esempio, per leggere il pulsante S2, occorre caricare la parola 000000010). In questo modo lo stato degli altri pulsanti non influenza la lettura di Din: campionando il segnale su tale terminale, si legge un valore

basso (dovuto all'uscita open-drain dell'integrato) se il pulsante è premuto, un valore alto (dovuto al pull-up resistivo) se il pulsante è aperto.

Una trattazione più approfondita del pilotaggio dell'interfaccia è comunque lasciata la capitolo 2.

1.6 Il prototipo

I componenti descritti fino a questo punto sono quelli che fanno parte del prototipo del sistema. Per quanto riguarda i caschi, abbiamo visto come sia possibile utilizzare qualsiasi headset commerciale, quindi non vengono trattati in questo lavoro. Il nodo master della piconet, cioè quello posizionato sulla moto, è composto dall'unione di DSP e BlueCore. La base di questo nodo è l'evaluation board dell'EZ-K Lite precedentemente descritta; a questa è collegata una piccola PCB con l'interfaccia utente da posizionare sul cruscotto della moto. Inoltre è predisposto un collegamento via UART con il Bluecore, che è anch'esso montato su una PCB ad hoc e non più nel modem Casira. I collegamenti necessari con l'ambiente circostante sono le entrate e l'uscita del codec AD1885, che sono collegate alle sorgenti audio (radio, telefono, ...), e l'alimentazione dei dispositivi.

Resta ancora da definire come lo stack Bluetooth risulti partizionato tra host e host controller. La scelta è stata quella di far girare l'intero stack di protocollo sul BlueCore e far risiedere l'applicativo in parte sull'host e in parte sull'host controller. Si deve allora specificare come avviene la comunicazione tra questi due dispositivi: essi si scambiano messaggi come pacchetti HCI, la cui forma è descritta in dettaglio nel capitolo 2 nell'ambito della descrizione della comunicazione tra DSP e BlueCore.

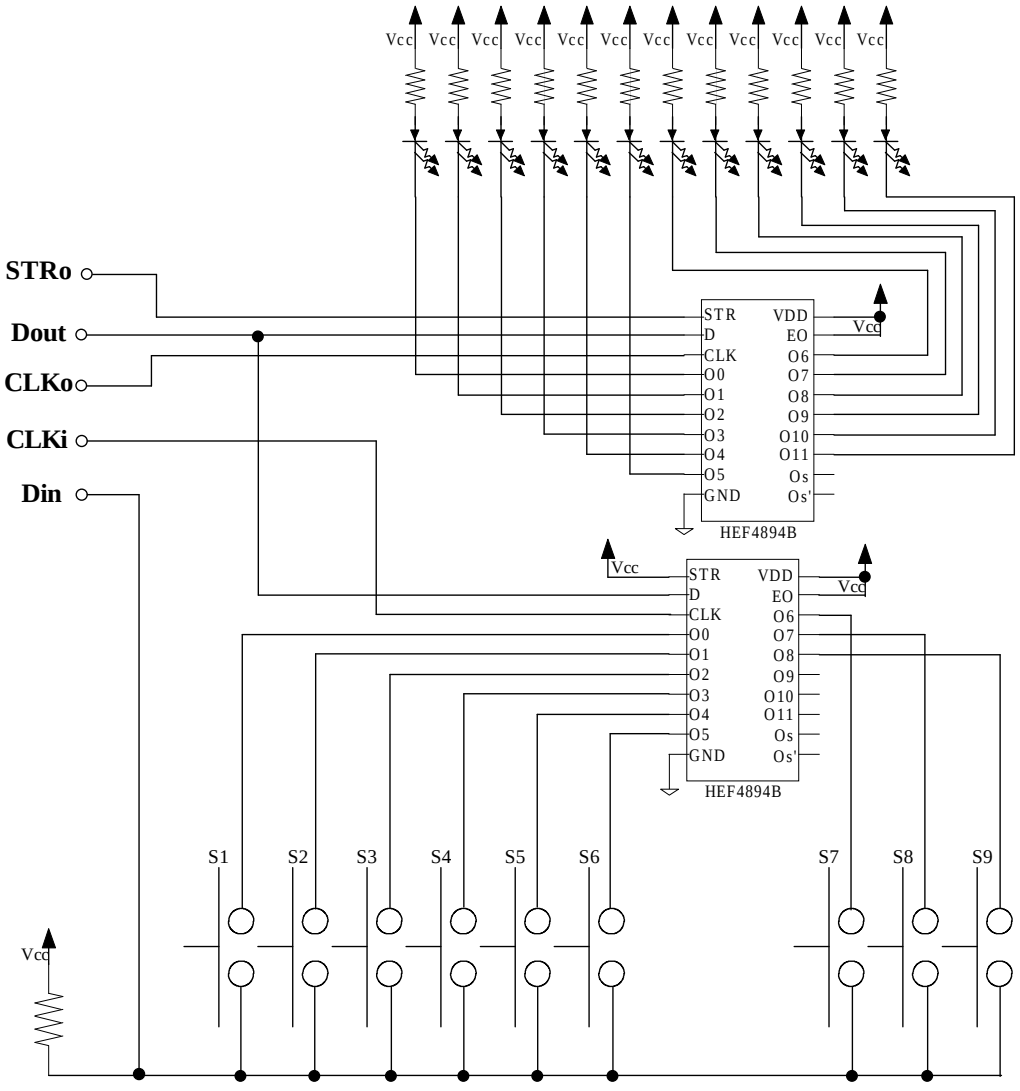

Figura 1.8: L'interfaccia utente

Capitolo 2

L'applicazione per il DSP

In questo capitolo viene descritta nel dettaglio l'applicazione sviluppata per il DSP che svolge i compiti di controllo e di elaborazione. Dapprima si illustrano le caratteristiche richieste all'applicazione e si discutono le scelte architetturali compiute. In un secondo tempo sono analizzati i singoli moduli e la specifica implementazione di ognuno.

2.1 La struttura

Il software eseguito sul DSP deve implementare le caratteristiche precedentemente descritte nel capitolo 1. Si tratta di interfacciarsi con l'host controller attraverso la porta UART e dirigere la sua attività attraverso uno scambio di messaggi, che possono essere comandi quando viaggiano dall'host all'host controller, eventi nella direzione opposta o dati audio in entrambe le direzioni. L'audio dall'host controller viene ricevuto dal DSP via UART, mentre quello proveniente dalle altre sorgenti via SPORT (che è direttamente collegata al codec); tutto l'audio viene poi elaborato secondo la modalità di funzionamento selezionata (interfono, chiamata cellulare, ...). Il processore riceve gli input dall'utente, che indicano tale modalità, sui pin GPIO, collegati all'interfaccia utente già descritta.

Partendo da queste specifiche generali, si è scelto di procedere utilizzando

un approccio di tipo top-down, suddividendo l'applicazione in adeguati blocchi funzionali indipendenti ed iterando tale procedimento di semplificazione su ciascun blocco, fino ad ottenere unità idonee all'implementazione.

Questo tipo di approccio, oltre a semplificare e ordinare lo sviluppo del software, porta ad una naturale modularità dell'intero progetto. In questo modo si è cercato di introdurre un certo grado di astrazione per facilitare le operazioni di test o modifica: infatti ciascun blocco comunica con gli altri attraverso un'interfaccia indipendente dall'implementazione del blocco.

La modularità cercata ha trovato un riscontro nei tool disponibili e nella stessa architettura hardware del microprocessore: ricordando che l'ambiente di sviluppo mette a disposizione il nucleo di un sistema operativo real-time (vedi capitolo 1), appare naturale mappare i diversi blocchi funzionali in altrettanti thread. Ciascuno di essi è caratterizzato da un flusso di programma completamente indipendente e la comunicazione con gli altri blocchi è assicurata dai meccanismi di sincronizzazione forniti dal sistema operativo, come semafori ed eventi. In aggiunta a questi sono necessarie solo un numero esiguo di strutture dati comuni che permettono lo scambio dei dati.

Nella figura 2.1 è indicata la struttura dell'applicazione con l'indicazione delle relazioni presenti tra i vari blocchi. Si nota la presenza, accanto ai thread, delle ISR (Interrupt Service Routine); nonostante non si fosse accennato a queste ultime in precedenza, esse sono vitali per il funzionamento del sistema. Infatti, come spiegato in maggior dettaglio nel seguito, si è scelto di far operare i thread che si interfacciano con la UART e la SPORT in stretta collaborazione con le ISR: ciò può apparire sconveniente ma ricordiamo che il software si appoggia direttamente sul substrato hardware del dispositivo (è presente solo il nucleo di sistema operativo) e deve comunque rimanere abbastanza a "basso livello".

Oltre a thread e ISR, nella figura sono indicate le strutture dati comuni e i dispositivi hardware (indicati in grassetto). La struttura appare quindi nella sua semplicità ed efficacia: una serie di blocchi è posta a diretto contatto con le periferiche e si occupa quindi delle operazioni di I/O (le routine di

interrupt, `UARTreader` e `UARTwriter`); i dati ricevuti o da inoltrare all'esterno sono posti in opportune strutture (`ScoIn1`, `ScoOut2`, `BcCommands`, ...); i due thread `AudioManager` e `ControlUnit` operano sui dati, il primo effettuando il miscelamento dei flussi audio, il secondo controllando la piconet e l'interfaccia utente.

Nella figura sono indicate tutte le relazioni tra i blocchi: sono state esplicitate le variabili globali, rappresentate da rettangoli, che fungono da collegamento tra i thread, mentre i collegamenti tra thread e ISR (ad esempio quella tra `UARTwriter` e `intDMAout`) sono solo indicati, lasciando ai paragrafi successivi un ulteriore approfondimento. Questo perché i meccanismi di collegamento con le routine scritte in Assembly sono più complessi e difficilmente comprensibili, visto che non si può ricorrere a strutture dati particolarmente complesse o a classi, come invece avviene per il codice in C++. Altra caratteristica dello schema è l'indicazione dei semafori e degli eventi utilizzati da ciascun blocco; questi rappresentano un ulteriore elemento di collegamento tra i blocchi e assicurano la sincronizzazione necessaria. E' bene notare che non viene però specificato il "verso" della sincronizzazione, cioè, ad esempio, quale thread fa una `Pend` su un semaforo e quale thread sblocca il primo con una `Post`. Quindi, per capire adeguatamente la struttura, è necessario fare riferimento ai prossimi paragrafi.

2.2 Connessione con l'host controller

Il modulo incaricato della comunicazione con l'host controller segue le indicazioni dell'HCI UART Transfer Layer (vedi sezione H:4 di [Bluetooth 2]), che sono illustrate nel paragrafo 2.2.3. Tralasciando la sua organizzazione interna, questo modulo deve interfacciarsi da un lato con l'hardware del dispositivo (la porta UART) e dall'altro con gli altri moduli dell'applicazione.

Per quanto riguarda la gestione della UART, l'architettura del DSP offre tre possibilità: trasferimenti a controllo di programma (o polling), trasferimenti a controllo d'interruzione e trasferimenti a DMA. E' stata scelta

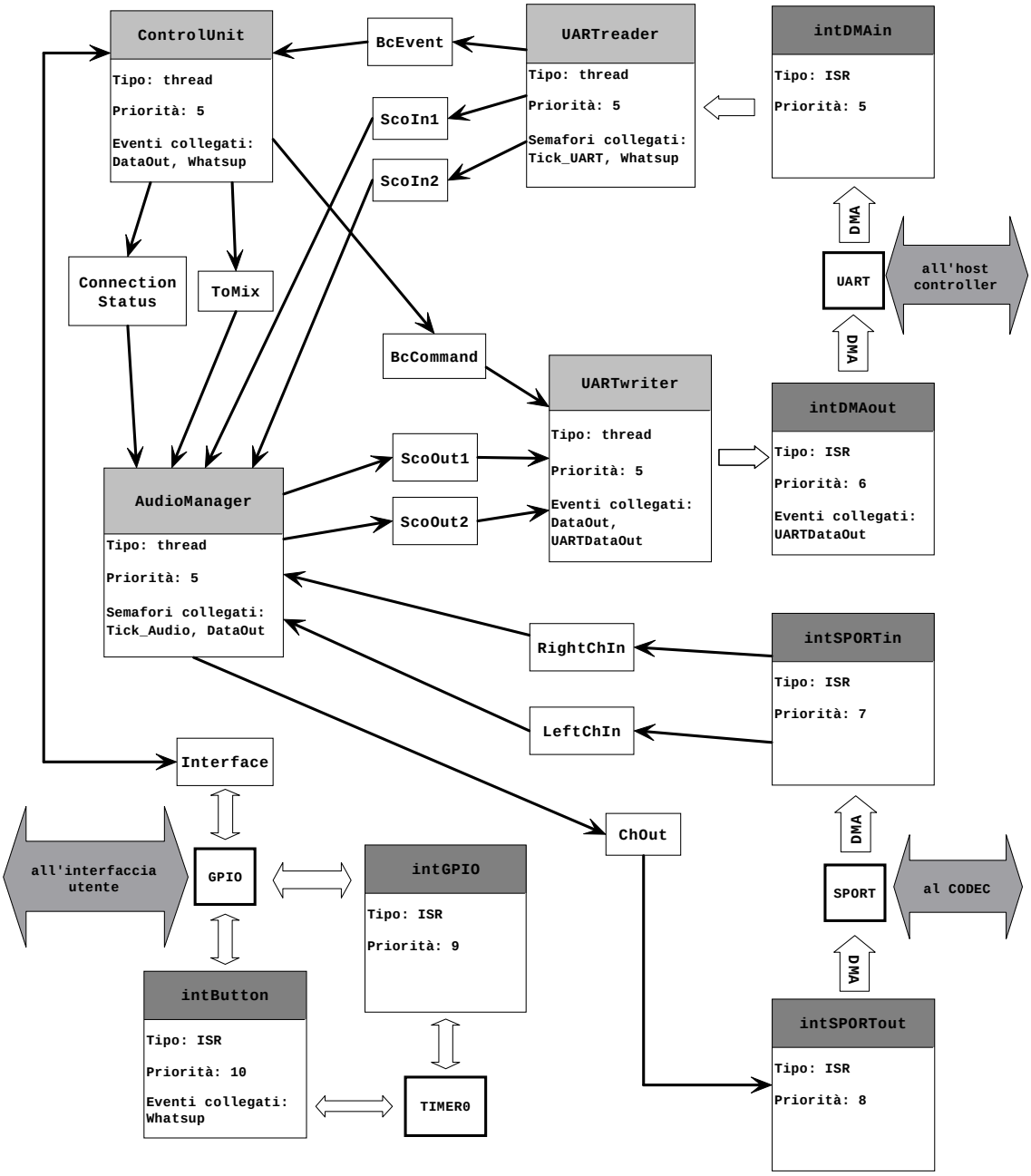

Figura 2.1: Architettura dell'applicazione

quest'ultima alternativa, in quanto le altre risultano inefficienti per il caso in esame. Infatti il polling della periferica è una tecnica assolutamente non utilizzabile in un contesto di tipo real-time come quello in esame: usarla implica controllare continuamente i bit di flag della periferica e trasferire i dati solo quando la periferica risulta pronta per la trasmissione o è stato ricevuto un dato valido. L'approccio con interrupt, nonostante sia una via praticabile, presenta un overhead eccessivo determinato dalla mole dei dati da trasferire; notare infatti che sarebbe stata richiesta l'esecuzione della appropriata ISR ad ogni byte trasmesso o ricevuto.[1] L'uso del controllore DMA è invece apparso idoneo proprio in conseguenza della gran quantità di dati: infatti questa periferica opera il trasferimento in maniera parallela rispetto al core del processore, anche se ha bisogno di essere configurato ogni volta che si trasmette un blocco di dati.[2] Quindi l'utilizzo è conveniente solo se la lunghezza del blocco dati è tale da rendere trascurabile il tempo necessario alla configurazione rispetto a quello impiegato nel trasferimento vero e proprio.

In riferimento all'organizzazione interna, il modulo poteva essere progettato come un device driver o in alternativa come un insieme di thread cooperanti, se si escludono le ISR che devono essere comunque presenti. Sembrerebbe questo il caso tipico di utilizzo di un device driver, dato che occorre gestire l'acceso in lettura o scrittura ad una risorsa quale il canale di comunicazione, ma questo approccio appare eccessivamente involuto: infatti in questa applicazione i thread che accedono alla risorsa sono sempre gli stessi ed ognuno di essi accede a flussi dati logicamente separati. E' stato allora preferito usare dei thread come tramite tra il dominio delle ISR e il dominio degli altri thread, implementando nello stesso modulo sia la gestione

[1] In questo caso la periferica genera un interrupt quando ha inviato correttamente un byte, ad indicare che è possibile inviare ulteriori dati; ne genera un altro quando riceve un byte per avvisare che c'è un dato disponibile nel registro che funge da buffer di ricezione.

[2] Quanto affermato non è vero in assoluto, ma dipende dalla modalità di utilizzo del controllore DMA. Per maggiori informazioni su tali modalità si faccia riferimento alla sezione 2.2.2

dell'hardware (la porta UART) sia il parsing dei pacchetti dati in ingresso e la codifica dei dati in uscita. In questo modo i moduli che compiono le elaborazioni trattano direttamente dei flussi di dati senza preoccuparsi della codifica dell'HCI UART Transfer Layer.

Come conseguenza della scelte fatte, sono presenti una parte composta da un thread e una ISR che gestisce la ricezione e una parte duale della prima che gestisce la trasmissione.

2.2.1 Le strutture per la gestione dei dati

Un'altra specifica del modulo, oltre all'interfacciamento con l'hardware, è la comunicazione con gli altri blocchi funzionali del sistema. Si è cercato di usare in questo caso un meccanismo di livello elevato (dal punto di vista dell'astrazione) per poter essere più liberi nelle scelte implementative riguardanti altre parti dell'applicazione. Questo meccanismo consiste nell'uso di alcune classi che permettono la gestione di buffer circolari FIFO (o code):

```
class AudioQueue {
public:
    AudioQueue(const int Len, const bool ex_sem);
    ~AudioQueue();
    void  AsmInterface(int ** p_Queue, int ** p_First, int ** p_Last);
    bool  Put(const int Dato);
    bool  Pick(int &Dato);
    int   Data();
    int   Spaces();
    bool  WriteArray(const int * Array, const int ArrayLength);
    bool  ReadArray(int * Array,const int ArrayLength);
    bool  Throw(const int Len);
};

class MsgQueue {
```

```
public:
    MsgQueue(const int Len);
    ~MsgQueue();
    bool  Put(const MsgHandle);
    bool  Pick(MsgHandle*);
};
```

Cominciamo dall'analizzare il funzionamento della classe `AudioQueue`, utilizzata nella gestione dei diversi flussi audio. I dati sono memorizzati in un'array di interi[3] di lunghezza `Len` che è gestito circolarmente attraverso l'uso di due indici per il primo e per l'ultimo elemento inserito; il prelievo e l'inserimento di un singolo dato è possibile con le rispettive funzioni membro `Pick()` e `Put()`. In questo modo sono implementate le funzionalità base di una coda ma non c'è nessuna ottimizzazione. Infatti la nostra applicazione richiederà sempre trasferimenti di blocchi di dati audio (con lunghezza dell'ordine di 80 campioni per volta[4]) e richiamare più volte la funzione `Put()` (o `Pick()`) implica la ripetizione del meccanismo di chiamata di funzione, che costituisce un overhead inaccettabile. Inoltre, come chiarito in seguito, ciascun accesso alla coda va gestito in mutua esclusione e di conseguenza ogni uso delle funzioni membro da luogo ad un ulteriore overhead per l'esecuzione dei meccanismi di protezione. Sono state allora implementate `ReadArray()`, `WriteArray()` e `Throw()` che rispettivamente prelevano, inseriscono e cancellano un numero qualsiasi di dati con una sola chiamata.

Per quanto riguarda la necessità di protezione a cui si è sopra accennato, si noti che le code usate (vedi le varie `ScoIn1`, `ScoOut2`, `BcEvents` della figura

[3]I dati audio trattati possono essere in formato lineare su 16 bit o in formato compresso su 8 bit. Dato che le parole usate dall'ADSP-2191 sono lunghe 2 byte, i campioni audio sono sempre memorizzati come interi (cioè su 16 bit); quando si trattano campioni compressi viene semplicemente usato il byte meno significativo mentre il più significativo rimane inutilizzato. Per ulteriori note sul formato dei campioni audio fare riferimento al paragrafo 2.6.1.

[4]Questa stima è così ottenuta: l'audio è campionato a 8 kHz e le elaborazioni avvengono con una cadenza tipica di 10ms. Quindi 8 kHz·10 ms = 80 campioni.

2.1) sono risorse condivise tra i vari thread. Come tutte le risorse condivise vanno allora protette, rendendo esclusive le operazioni compiute su di esse: ciò è realizzato con il classico meccanismo dei semafori di mutua esclusione. Quando viene istanziato un oggetto `AudioQueue` il costruttore dell'oggetto crea dinamicamente un semaforo inizializzato ad 1 e all'interno di ciascuna funzione membro vengono eseguite una `VDK::PendSemaphore()` prima di entrare in una sezione critica ed una `VDK::PostSemaphore()` alla sua conclusione. In realtà tale semaforo non viene creato per le code al cui costruttore è passato come parametro `ex_sem` un valore booleano `false`. Ciò è necessario perché alcune operazioni di inserimento in coda sono eseguite da ISR (è il caso di `RightChIn` e `LeftChIn`): in questo caso il meccanismo di protezione realizzato con il semaforo di mutua esclusione non è utilizzabile in quanto le ISR non possono bloccarsi sui semafori e ciascuna di esse potrebbe interrompere una funzione membro che sta operando su una particolare coda; se poi tale ISR va ad operare su tale coda, la potrebbe trovare in uno stato inconsistente e il risultato delle sue operazioni potrebbe risultare imprevedibile. Quindi si proteggono le sezioni critiche usando le API `VDK::PushCriticalRegion()` e `VDK::PopCriticalRegion()`, che disabilitano gli interrupt e impediscono la commutazione di contesto. C'è da aggiungere che le ISR che agiscono sulla coda non possono chiamare le funzioni membro (dato che sono scritte in Assembly) ma devono usare direttamente le variabili private della classe;ciò è possibile grazie a `AsmInterface` che restituisce gli indirizzi dell'array dei dati e dei puntatore al primo e all'ultimo elemento della coda.

La classe `MsgQueue`, utilizzata come coda per comandi ed eventi, è molto più semplice in quanto i comandi e gli eventi del BlueCore vengono trattati uno alla volta. C'è solo da puntualizzare che i dati accodati sono puntatori a strutture dati di tipo `MsgHandle` che vengono allocate e distrutte dinamicamente. La struttura `MsgHandle` è semplicemente un record i cui campi identificano i vari parametri dei comandi e degli eventi del Bluetooth ma una trattazione dettagliata di essi esula dallo scopo di questo lavoro.

2.2.2 Configurazione del controllore DMA

Il controllore DMA dell'ADSP-2191 può operare in due modalità nel trasferimento dati da e verso la porta UART. Nel modo con *autobuffer* viene indicato un buffer circolare in cui il DMA legge o scrive una volta configurato; in questo modo non sono necessari ulteriori interventi software per il suo funzionamento dopo l'inizio del trasferimento. E' necessario configurare solo tre parametri: *page*, *start* e *count*, che indicano rispettivamente la pagina di memoria dove risiede il buffer, il suo indirizzo e la sua lunghezza; queste informazioni identificano le locazioni di memoria di interesse e, una volta poste nei registri opportuni, è possibile far partire il trasferimento.

L'altro metodo è quello che fa uso dei *descrittori*: il controllore viene istruito ad operare un determinato trasferimento da una struttura dati detta descrittore DMA. Nella tabella 2.1 è indicata la struttura di un descrittore.

Indirizzo	Nome	Descrizione
HEAD	DMA Configuration	Proprietà del descrittore e configurazione
HEAD + 1	DMA Start Page	Indirizzo della pagina di memoria di interesse
HEAD + 2	DMA Start Addr	Indirizzo della prima locazione di memoria di interesse
HEAD + 3	DMA Word Cnt	Numero di parole da trasferire
HEAD + 4	Next Descriptor	Puntatore all'indirizzo del prossimo descrittore

Tabella 2.1: Struttura di un descrittore

Esso contiene le informazioni necessarie ad attuare un trasferimento di dati: come nel caso dell'autobuffer è sufficiente identificare il buffer in memoria da cui prendere o in cui porre i dati. Infatti si vede dalla tabella che si indicano la *start page*, lo *start address* e il *word count*. In aggiunta ci sono due ulteriori campi: una parola di configurazione e un puntatore ad un altro descrittore. Nella prima sono settate le proprietà del trasferimento; tra gli altri ci sono il

bit che, settato, indica di far partire un interrupt alla fine del trasferimento e un bit di proprietà, che vale 1 quando il descrittore appartiene al DMA (cioè il trasferimento è in corso) e 0 quando appartiene al core (cioè è in corso la configurazione del descrittore). Nella parola *Next Descriptor* è indicato l'indirizzo dell'HEAD di un altro descrittore che viene caricato quando il trasferimento è compiuto: ciò permette la modalità *chained* con la quale si può creare una catena di descrittori, di cui solo quello associato al primo trasferimento deve essere preventivamente caricato nel controllore DMA via programma. Il trasferimento di tipo chained termina quando il puntatore *Next Descriptor* di un descrittore punta ad una locazione di memoria che contiene 0x00. Il meccanismo attraverso il quale il programma deve configurare un trasferimento con i descrittori è leggermente più involuto del caso di autobuffer: bisogna scrivere nella pagina 0 di memoria le 4 parole consecutive da HEAD + 1 a HEAD + 4; quindi si scrive HEAD (ponendo il bit di proprietà a 1) e si carica il suo indirizzo nel registro Next Desciptor del canale DMA desiderato (trasmissione o ricezione); poi si setta un bit in un registro di I/O e questo fa partire il trasferimento. Al termine di questo, il DMA aggiorna il descrittore in memoria ponendo a 0 il bit di proprietà.

Durante le prove effettuate sul processore, si è notato che è buona norma, prima di far partire il trasferimento, settare nel registro di configurazione del canale DMA il bit di *flush*, che elimina da alcuni registri del controllore DMA i valori inerenti un precedente utilizzo.

2.2.3 Il formato dei dati

E' opportuno qui riportare il formato dei dati che vengono trasmessi sulla UART, descritto nell'HCI UART Transfer Layer. Si tratta di un flusso di pacchetti di 4 tipi diversi: *comando*, *evento*, *SCO* e *ACL*.

Prima della trasmissione del pacchetto vero e proprio viene trasmesso un byte, detto *indicatore* di pacchetto, che identifica il tipo del pacchetto trasmesso immediatamente dopo, come riportato nella tabella 2.2.

Ciascun pacchetto ha poi un particolare formato, riportato nelle figu-

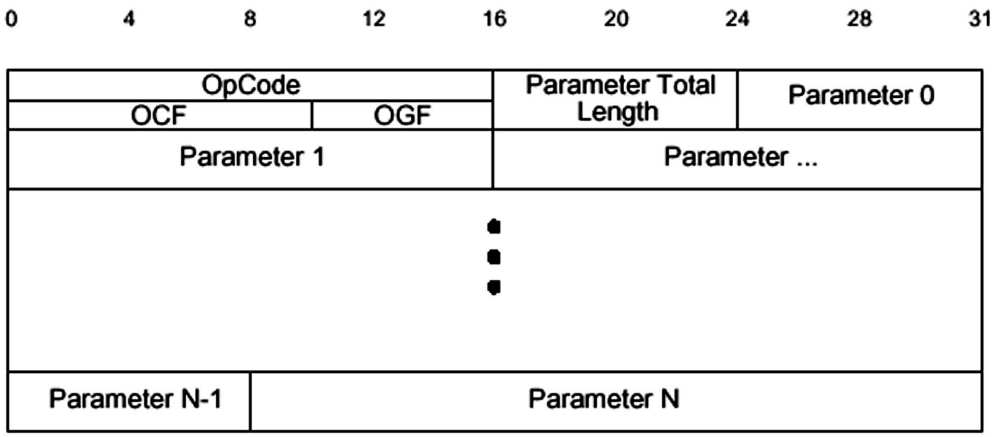

Figura 2.2: Pacchetto comando

re 2.2, 2.3, 2.4 e 2.5. Nella nostra applicazione non vengono mai usati pacchetti ACL, ma viene comunque implementato il loro riconoscimento per estensioni future: in questa versione del software, in caso si presentassero, sarebbero semplicemente buttati via. Ciascun pacchetto può essere pensato come unione di un *header* e di un *corpo* contenente i dati veri e propri. Nell'header sono contenute diverse informazioni: quelle da noi usate sono la lunghezza del pacchetto per tutti i tipi e per i soli pacchetti SCO il *Connection Handle*, che indica da quale dispositivo provengono i campioni audio del corpo che segue l'header. Importante è ricordare che i campi multibyte sono trasmessi in Little Endian (dal byte meno significativo al più significativo).

Tipo	Indicatore
Comando	0x01
ACL	0x02
SCO	0x03
Evento	0x04

Tabella 2.2: Indicatori di pacchetto HCI

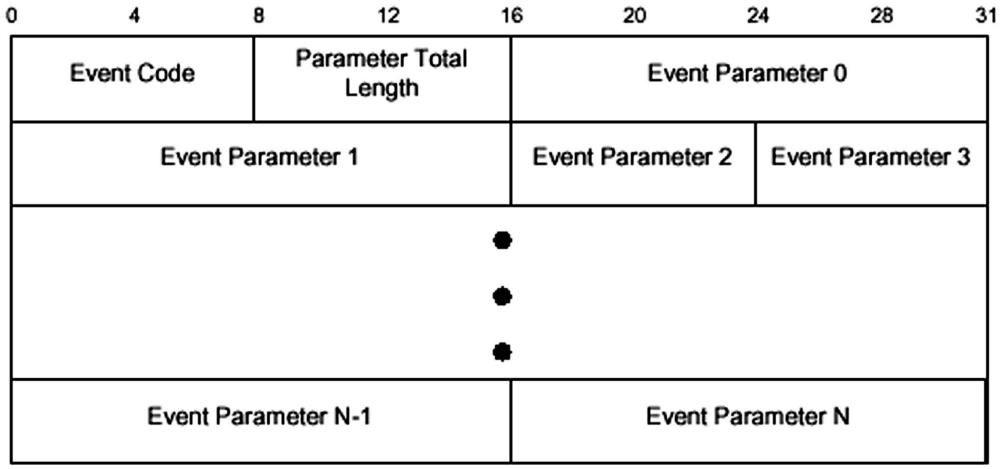

Figura 2.3: Pacchetto evento

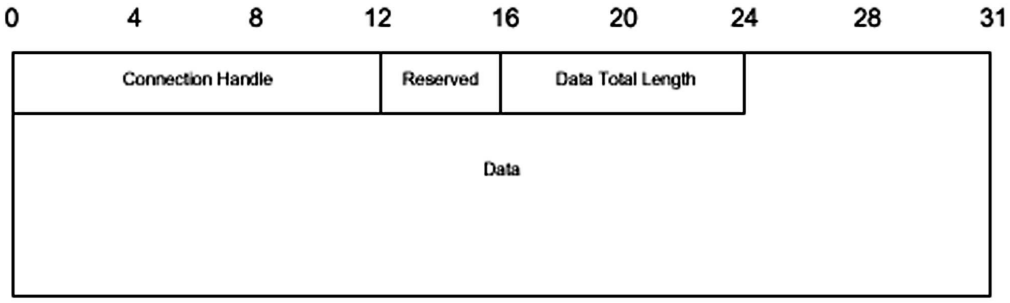

Figura 2.4: Pacchetto SCO

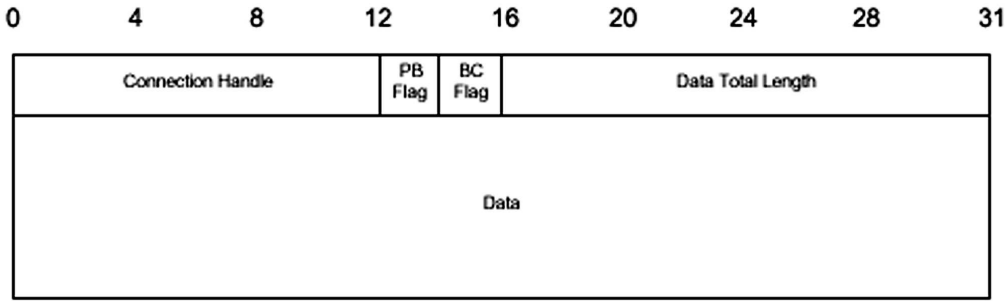

Figura 2.5: Pacchetto ACL

2.2.4 Le routine di gestione dell'interrupt

Abbiamo visto come il DMA possa essere gestito in due modalità, ambedue adatte in qualche modo al nostro caso: l'autobuffer e i descrittori concatenati. Ciò è vero per quanto concerne la lettura, ma non altrettanto per la scrittura. Si osserva infatti che l'autobuffer non può essere usato in scrittura per un semplice motivo: se il DMA cicla continuamente su una zona di memoria, trasmettendo quello che vi risiede, cosa accade quando non bisogna mandare nulla in uscita, ad esempio quando la connessione con i caschi è disabilitata? In questo caso il DMA continuerebbe a mandare i byte che trova in memoria: questi sarebbero nulli se il buffer fosse ancora vergine o sporchi se fosse stato già utilizzato. Sarebbero comunque trasmessi pacchetti non validi, causando un blocco del BlueCore. Ciò non avviene naturalmente in lettura perché con l'autobuffer il DMA scrive la memoria solo quando la UART riceve dei byte; in questo caso l'unico rischio è che il thread non sia abbastanza veloce da leggere i dati e questi vengano sovrascritti da una successiva passata del DMA. Per evitare questo problema occorre calibrare opportunamente la lunghezza del buffer e la velocità di esecuzione dell'`UARTreader`, come descritto nel capitolo 3.

Per quanto detto, sono state sviluppate entrambe le soluzioni, lasciando alla fase di messa a punto la scelta di adottare l'una o l'altra. Occorre una ISR solo nel caso di trasferimento a descrittori, dato che l'autobuffer, una volta inizializzato, non necessita di ulteriori configurazioni. Quindi, in caso si scelga l'autobuffer, dalla struttura dell'applicazione scompare `intDMAin`.

In caso di uso dei descrittori, la ISR deve rispondere all'interrupt generato dal DMA alla fine di un trasferimento e provvedere ad una opportuna configurazione dei descrittori. Consideriamo dapprima il caso di `intDMAout`. Il DMA prende i dati da un'opportuna area di memoria, in cui il thread `UARTwriter` scrive i dati da trasferire. Il compito della ISR è di sincronizzare la scrittura dei dati nell'area di memoria da parte del thread e la lettura degli stessi da parte del controllore DMA. Affincé ciò sia possibile, il thread deve comunicare alla ISR i suoi accessi all'area di memoria, in quanto la ISR deve

conoscere quali sono le locazioni a cui il thread ha già acceduto e che di conseguenza contengono dati validi. D'altra parte l'ISR deve indicare al thread le locazioni libere, cioè quelle che il DMA ha già letto. L'unico modo[5] per fare ciò è usare delle variabili globali, che indichino l'indirizzo del buffer condiviso tra thread e ISR, gli indirizzi della prima locazione da scrivere, dell'ultima locazione letta e il numero di byte disponibili per la scrittura. Questo compito è assegnato rispettivamente alle variabili `data_UARTout`, `first_to_write` (come si capisce dall'identificatore essa contiene l'indirizzo della prima locazione da scrivere), `lastDMAout` e `quantity`[6]. L'area di memoria, che nel codice del thread è dichiarata come un'array, viene gestita come una coda FIFO, attraverso l'aggiornamento di queste variabili. Il thread accoda nel buffer i dati da mandare sulla UART, mentre la ISR configura opportunamente il DMA per estrarre tali dati dalla coda e trasferirli al BlueCore.

Il trasferimento è governato da due descrittori concatenati circolarmente, cioè in cui il primo punta al secondo e il secondo al primo. Per il primo trasferimento, essi vengono settati dall'`UARTwriter`; successivamente sono aggiornati dalla ISR. Questa va in esecuzione quando il blocco indicato da un descrittore è stato completamente trasferito; inizialmente essa analizza i descrittori in memoria: uno di essi è quello che è stato appena usato e va quindi riconfigurato. Viene calcolato quanto deve essere lungo il blocco successivo: se ci sono abbastanza dati validi, si cerca di trasferire un blocco di lunghezza fissa; altrimenti si trasferiscono tutti i byte che ci sono. Bisogna anche considerare che un blocco non può trovarsi a cavallo della fine e dell'inizio del buffer, poiché il DMA considera, in questa modalità, solo buffer lineari; di conseguenza, si controlla se il numero di byte che si desidera trasferire in un unico blocco è maggiore del numero di byte che risiedono tra la locazione iniziale del blocco e la fine del buffer; in caso positivo, la lunghezza del blocco è data dal numero di dati posti tra la locazione iniziale del blocco e la fine del buffer.

[5] Ricordiamo che non è possibile passare parametri ad una ISR, dato che la sua esecuzione non è sincronizzata con il resto del programma.

[6] Quest'ultima potrebbe essere evitata nella gestione della coda, ma semplifica il codice.

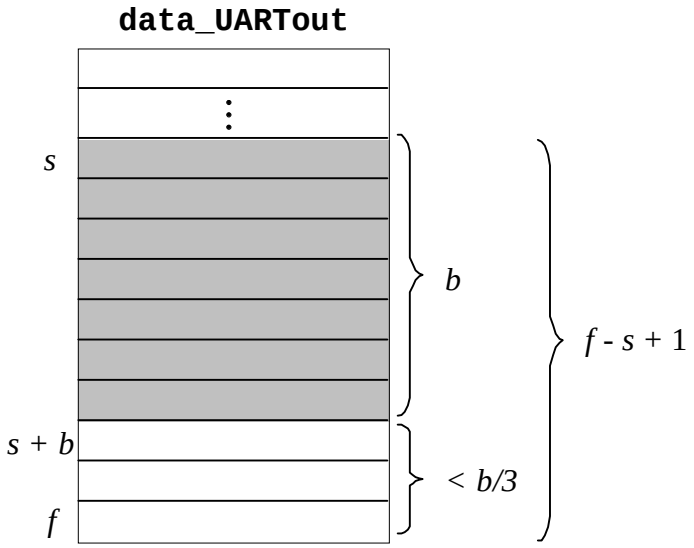

Figura 2.6: Esempio di configurazione dei descrittori

Un particolare accorgimento è stato utilizzato per evitare il trasferimento di blocchi troppo piccoli, che equivale ad un degrado delle prestazioni[7]. Facendo riferimento alla figura 2.6, si supponga di dover porre lo start address pari a s e di avere una lunghezza fissa del blocco pari b. Se il buffer finisce alla locazione f e fino in fondo del buffer ci sono dati validi, si controlla quanti byte rimarrebbero tra s e f, dopo aver trasferito b byte. Questi byte sarebbero letti in trasferimenti successivi: se il loro numero è inferiore a $b/3$, si setta il word count del descrittore a $f - s + 1$ invece che a b, includendo tutti i dati fino a f; in questo modo si evitano trasferimenti di blocchi con meno di $b/3$ byte. Si potrebbe pensare di compiere un'operazione analoga in un caso simile, quando ad esempio il numero di dati validi nel buffer e pari a $b + x$, con $x < b/3$. In questo caso la limitazione è costituita dall'esiguità di dati disponibili e non più dalla prossimità della locazione f, ma a regime si suppone che nuovi dati validi siano aggiunti tra un'esecuzione della ISR e la

[7]Si faccia riferimento alle considerazioni già fatte riguardo l'overhead dovuto al caricamento in memoria dei descrittori. Se i blocchi sono più piccoli occorre configurare più volte i descrittori.

successiva; questi nuovi dati impediscono che il blocco da creare sia troppo piccolo e rendono inutili accorgimenti del tipo descritto.

Oltre alla configurazione del nuovo descrittore, la ISR aggiorna i puntatori alla coda FIFO `data_UARTout` e setta l'event bit relativo all'evento `UARTDataOut`: su questo si blocca infatti `UARTwriter` quando non ci sono locazioni disponibili nel buffer. Quando La ISR va in esecuzione, si è in genere appena concluso un trasferimento, quindi si sono appena liberate delle locazioni e ciò viene comunicato al thread attraverso l'evento.

Una situazione particolare è quella in cui la ISR non può configurare il descrittore in quanto non ci sono dati da trasferire. In questo caso, ciò viene comunicato all'`UARTwriter` attraverso un flag. Infatti, una volta terminato il trasferimento di un blocco, se la ISR non ha più dati da trasferire non può riaggiornare il descrittore; quindi quando è terminato anche il trasferimento associato all'altro blocco, il meccanismo automatico di trasferimento si arresta. Allora è compito del thread chiamare l'interrupt via software dopo che ha scritto nuovi dati nel buffer. Inoltre il thread setta anche un particolare flag, ad indicare alla ISR che si tratta di un interrupt software: questo per evitare che la ISR creda che sia terminato un trasferimento e aggiorni in maniera errata i puntatori della coda.

La routine per la gestione del canale di ricezione è completamente duale di quella ora descritta e non viene perciò analizzata. Unica differenza è l'assenza in questa routine di un evento analogo a `UARTDataOut`. Quest'approccio sarebbe infatti risultato fallimentare in alcuni casi, come viene ora dimostrato. Supponiamo che il thread `UARTreader` si sia bloccato su tale ipotetico evento; tornerà pronto solo dopo che il DMA avrà generato un interrupt, cioè alla fine del trasferimento del blocco. Ma cosa succede se il controllore DMA non termina questa operazione e di conseguenza non genera l'interrupt? Naturalmente non vengono mai letti i dati che compongono quel blocco, dato che `UARTreader` risulta bloccato. Questa situazione può accadere realmente quando il BlueCore manda un pacchetto evento al DSP e si pone in uno stato di attesa passiva, in cui non manda dati finché non ha

ricevuto una risposta; in ingresso alla UART non ci sono quindi più dati e potrebbe accadere che l'evento inviato dal BlueCore non riempia un blocco per intero; allora il DMA rimarrebbe in attesa di dati e non genererebbe l'interrupt, causando una situazione di stallo. Si potrebbe pensare di ovviare al problema riducendo la dimensione dei blocchi: certamente diminuirebbe la probabilità di arresto del sistema, ma si avrebbe un sistema robusto solo se i blocchi fossero di lunghezza unitaria, che è inaccettabile. Allora l'`UARTreader` non si blocca su un evento simile a `UARTDataOut` ma sul semaforo periodico `Tick_UART`; così non aspetta la fine del trasferimento di un blocco per leggere i dati, ma accede ai registri del controllore DMA e dal loro contenuto ricava quali sono i dati ricevuti. Quest'operazione di lettura dai registri del DMA è necessaria anche quando si opera in autobuffer, dato che non viene usata alcuna ISR per aggiornare i puntatori della coda FIFO.

2.2.5 Il thread UARTreader

Questo thread si occupa di processare i dati ricevuti dall'host controller: esso accede in lettura all'area di memoria nella quale il controllore DMA scrive i dati provenienti dalla porta UART e accede in scrittura a tre diverse code (che sono classi del tipo `FifoQueue`) `ScoIn1`, `ScoIn2` ed `BcEvent`. Analizziamo dapprima la struttura principale del thread e passiamo in seguito ad indicare gli aspetti secondari.

Poiché sulla UART arrivano dati grezzi sotto forma di pacchetti eventi, SCO e ACL, esso deve provvedere, oltre che al puro trasferimento dei dati, all'analisi delle intestazioni dei pacchetti e al corretto instradamento dei dati nelle tre code. Per questo il programma è stato strutturato per funzionare come una macchina a stati finiti, come illustrato nella figura 2.7. Lo stato attuale del thread è indicato dalla variabile di tipo enumerato `stato`, che può assumere i valori `attendo_indicatore`, `attendo_header` e `attendo_pacchetto`, ciascuno dei quali codifica lo stato corrispondente.

All'inizio, nello stato `attendo_indicatore`, si cerca uno degli indicatori di pacchetto validi tra quelli indicati dall'HCI Tranfer Layer e riportati nella

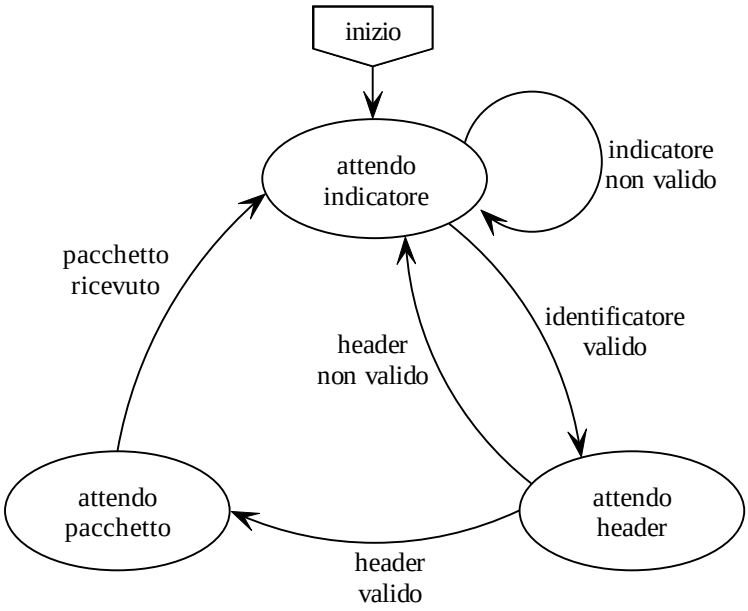

Figura 2.7: Schema a stati del funzionamento del thread UARTreader

tabella 2.2. Quando è stato riconosciuto un identificatore valido si passa all'analisi dell'header del pacchetto, dal quale si estraggono due informazioni: il tipo di pacchetto (e nel caso di dati SCO anche l'handle della connessione) e la lunghezza del pacchetto. A questo punto, se l'header risulta valido, si può trasferire l'intero pacchetto nell'opportuna coda (che è stata identificata grazie all'analisi dell'header) e si ritorna allo stato `attendo_indicatore`.

Naturalmente nel corso del parsing di un pacchetto è possibile che vengano esauriti i dati validi e occorre porsi in uno stato di attesa. Questo stato non è rappresentato in figura e non è codificato nella variabile `stato`. Si può pensare ad esso come una semplice sospensione del flusso di programma; infatti il thread si blocca sul semaforo periodico `Tick_UART` quando ha esaurito tutti i dati oppure quelli rimasti non sono sufficienti per processare il pacchetto corrispondente.

Il programma include anche dei controlli sullo stato della connessione. Infatti il parser si basa sul fatto che tutti i pacchetti arrivino in sequenza, cioè l'indicatore di pacchetto segua immediatamente l'ultimo byte del pac-

chetto precedente. In caso di disallineamento al flusso dei pacchetti, il parser andrebbe in errore e potrebbe dare in uscita dati non validi senza che il resto dell'applicazione possa accorgersene. Per evitare ciò è stato introdotto il meccanismo illustrato in figura 2.8. Quando il parser rileva un errore, si passa ad uno stato di funzionamento di errore nel quale viene cercato un nuovo pacchetto. Cercare un nuovo pacchetto significa riconoscere un indicatore tra quelli validi e interpretare i byte che lo seguono come header. Questo evento non fa uscire il programma dallo stato di errore, perché il parser potrebbe aver interpretato come indicatore un qualsiasi byte nel flusso dati pari a 0x02, 0x03 o 0x04. Allora si guarda se subito dopo questo ipotetico pacchetto c'è un indicatore valido; se c'è, la probabilità di essersi riallineati è abbastanza grande e si torna al funzionamento normale. Nello stato di errore viene settata una particolare variabile che può essere letta dalla `ControlUnit`, che quindi può conoscere lo stato della connessione tra DSP e host controller e che di conseguenza può intraprendere opportune azioni, come il reset del BlueCore, se la condizione di corretto funzionamento non viene ristabilita in un tempo prefissato. In realtà gli errori sulla UART saranno praticamente assenti durante il funzionamento del sistema; comunque questo meccanismo si rivela molto utile nella messa a punto del software.

Il flusso dei dati in ingresso è costituito per la maggior parte da dati audio e per il resto da eventi. E' utile allora comunicare alla `ControlUnit` l'arrivo di un evento, per evitare che questo thread vada continuamente a leggere `BcEvent` in cerca di nuovi eventi. Viene allora settato l'event bit `NewEvent` relativo all'evento `Whatsup`.

2.2.6 Il thread UARTwriter

Questo thread svolge una funzione duale di `UARTreader` in quanto provvede a prendere i dati dalle code `ScoOut1`, `ScoOut2` e `BcCommand` e a formattarli secondo le indicazioni date nel paragrafo 2.2.3. Di conseguenza esso si blocca sull'evento `DataOut`, che è dato dall'OR logico dei due event bit `AudioDataOut` e `CommandOut`, settati rispettivamente dall'`AudioManager` e dalla `ControlUnit`.

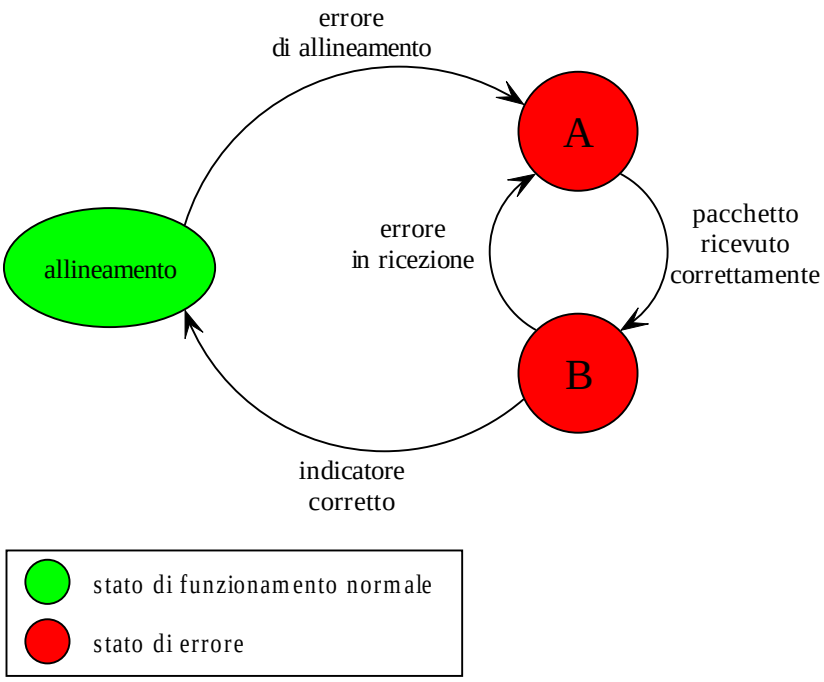

Figura 2.8: Stati di errore e di corretto funzionamento nel parsing dei pacchetti

In questo modo il thread va in esecuzione solo quando c'è bisogno di trasferire veramente dei dati. Un altro evento utilizzato è `UARTDataOut` di cui si è parlato nell'ambito delle ISR.

Nonostante il funzionamento sia molto semplice, occorre specificare la priorità con la quale vengono mandati i pacchetti in uscita. Ad ogni esecuzione del thread potrebbero esserci più fonti con dati validi e non potrebbe esserci abbastanza spazio sul buffer in uscita per scriverli tutti. Si da allora priorità maggiore ai comandi, mentre i dati SCO si trasmettono in maniera proporzionale al riempimento delle rispettive code. In pratica, dapprima si trasmettono i comandi, poi si calcola lo spazio disponibile rimasto d, tenendo conto dello spazio occupato dagli header dei pacchetti SCO; infine, detti n_{sco1} e n_{sco2} i dati presenti in `ScoOut1` e `ScoOut2`, si pone semplicemente:

$$n_1 = \left\lfloor d \cdot \frac{n_{sco1}}{n_{sco1} + n_{sco2}} \right\rfloor$$

$$n_2 = d - n_1$$

Così non si rischia che ad ogni attivazione del thread vengano trasmessi i dati di un solo canale ritardando la trasmissione degli altri. Ciò evita un possibile deterioramento della qualità dell'audio del canale che si troverebbe in ritardo.

2.3 Interfacciamento con il codec

Nel capitolo 1 si è accennato che le sorgenti audio esterne vengono collegate alla scheda di sviluppo utilizzando il codec AD1885 presente sulla scheda. Questo dispositivo dispone di un certo numero di ingressi analogici, ma è in grado di convertire contemporaneamente solo una coppia di sorgenti, alle quali ci riferiremo come canale destro e canale sinistro. Di conseguenza, visto che si intende collegare al codec tre dispositivi (radio, telefono cellulare e navigatore), vengono di volta in volta selezionati due dei tre ingressi disponibili in modo da soddisfare le richieste degli utenti; di questo si deve tenere conto nella gestione del modo di funzionamento, ricordando che non si possono ricevere contemporaneamente i tre flussi audio suddetti. In questo modo, si ottiene la gestione di due canali in ingresso e un canale in uscita, che nell'applicazione sono collegati a tre code, rispettivamente `RightChIn`, `LeftChIn` e `ChOut`.

Il protocollo utilizzato dal codec è basato sulla specifica Audio Codec '97 Revision 2.1 della Intel [AC'97], che impone severi vincoli temporali sulla gestione del collegamento. La specifica prevede lo scambio continuo e sincrono di frame di bit, con un baud-rate fissato di 12.288 Mbit/s, regolato da un segnale di sincronizzazione generato dal DSP. Ogni frame ha una lunghezza di 256 bit ed è suddiviso in 12 slot, ciascuno con una particolare funzione. Si noti che la frequenza di trasmissione dei frame risulta di 48 kHz, pari alla massima frequenza di campionamento del codec. La comunicazione avviene mandando frame opportunamente formattati al codec: questi contengono sia i comandi per la configurazione del dispositivo, sia i campioni audio; il

codec risponde inviando altri frame con l'audio proveniente dall'esterno della scheda.

E' bene notare, che nonostante ci interessi campionare l'audio a 8 kHz, occorre comunque gestire la trasmissione dei frame a 48 kHz. Questo è un vincolo temporale abbastanza forte e ha indotto ad utilizzare la modalità di trasferimento autobuffer del controllore DMA. Inoltre la porta SPORT è configurata per lavorare in multichannel, cioè utilizzando una tecnica Time Division Multiplexing, attraverso la quale vengono configurati 16 canali da 16 word ciascuno, che nel complesso permettono la ricezione o la trasmissione di un intero frame. Sono allora utilizzati due buffer da 16 parole, gestiti dalle ISR `intSPORTin` e `intSPORTout`, che vengono richiamate ogni volta che il DMA ha letto o ha scritto il rispettivo buffer. La prima estrapola dai frame i campioni audio da accodare a `RightChIn` e a `LeftChIn`, mentre la seconda si limita a formattare i frame inserendoci i campioni audio di volta in volta prelevati dalla coda `ChOut`.

Oltre alle ISR, occorre una funzione che provveda alla abilitazione degli opportuni ingressi, in modo da selezionare i due flussi audio d'interesse per ogni modo di funzionamento. Questa funzione, chiamata `CodecChangesSources()`, interrompe temporaneamente le due ISR e manda al codec opportuni frame con i comandi per la configurazione degli ingressi. Essa viene richiamata dalla `ControlUnit`, che si preoccupa di configurare gli ingressi audio necessari alla soddisfazione delle richieste dell'utente.

2.4 L'interfaccia utente

2.4.1 La tastiera

Vediamo com'è gestita la tastiera, partendo dal livello dell'hardware fino ad arrivare alla classe C++ che implementa la lettura dello stato dei pulsanti e la scrittura dello stato dei led.

Nel primo capitolo si è visto come leggere lo stato di un pulsante alla volta, imponendo una tensione bassa sull'opportuna uscita dell'integrato

HEF4894B e andando a campionare la tensione su Din. Se si volesse operare in questo modo, occorrerebbe andare ad osservare ciclicamente lo stato di ogni pulsante. Un modo alternativo è invece il seguente: partendo dalla condizione nella quale nessun tasto è premuto, si portano a tensione bassa le uscite O0-8 dell'integrato e si configura il GPIO collegato a Din per generare un interrupt sul fronte in discesa. In questo modo, alla pressione di un qualsiasi tasto viene generato un interrupt, in particolare l'interrupt 9. A questo punto non si conosce qual è il pulsante che è stato premuto: di conseguenza la routine `intGPIO` setta il TIMER0 per generare interrupt ad intervalli di 1 ms[8]. La ISR che gestisce questo interrupt è `intButton`; essa va a leggere singolarmente lo stato di ogni pulsante e definisce il valore dell'event bit `ChangeButton` (relativo all'evento `Whatsup`) per notificare alla `ControlUnit` l'intervento dell'utente.

In fase di inizializzazione viene caricato nello shift register dell'integrato la parola 0x1FF, che porta a tensione bassa tutte le uscite collegate ai pulsanti. Quando viene invocata, la routine `intGPIO` smaschera l'interrupt 9, ossia quello associato al TIMER0, che viene opportunamente configurato, e genera anche la temporizzazione indicata in figura 2.9. Secondo questa temporizzazione, vengono prodotti 8 cicli di clock come quelli indicati in figura, in modo che solo O8 sia a tensione bassa e gli altri piedini in alta impedenza.

A questo punto, il controllo passa a `intButton`, che provvede a campionare Din e a fornire un altro ciclo di clock per inserire un 1 nello shift register. Nelle successive 8 esecuzioni, la routine carica 0 nello shift register e sposta il bit 1 inserito precedentemente lungo tutto il registro. All'esecuzione successiva la ISR torna ad inserire un 1 e si continua così; in questo modo si fa scorrere ciclicamente il bit pari ad 1 lungo tutto il registro, per poter effetture la lettura dello stato di un singolo pulsante, come descritto nel primo capitolo. Si può notare che, messo a 1 il bit i-esimo del registro, lo stato del pulsante relativo viene campionato alla successiva esecuzione della ISR, cioè dopo 1

[8]Come chiarito in seguito, viene analizzato lo stato di un pulsante ad ogni esecuzione della relativa ISR; di conseguenza lo stato di ogni pulsante è campionato ogni 9 ms.

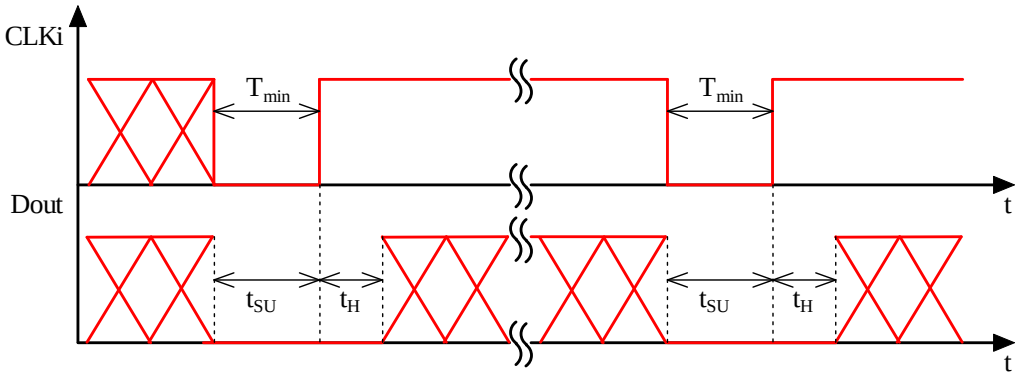

Figura 2.9: Temporizzazione comandata dalla routine `intGPIO`

ms; in questa maniera il circuito sarà sicuramente a regime e non occorre introdurre cicli di attesa nella ISR. Nella figura 2.10 vengono mostrati, non in scala, i segnali generati in due esecuzioni successive della ISR. La freccia nera indica il campionamento di Din, che avviene all'inizio della ISR.

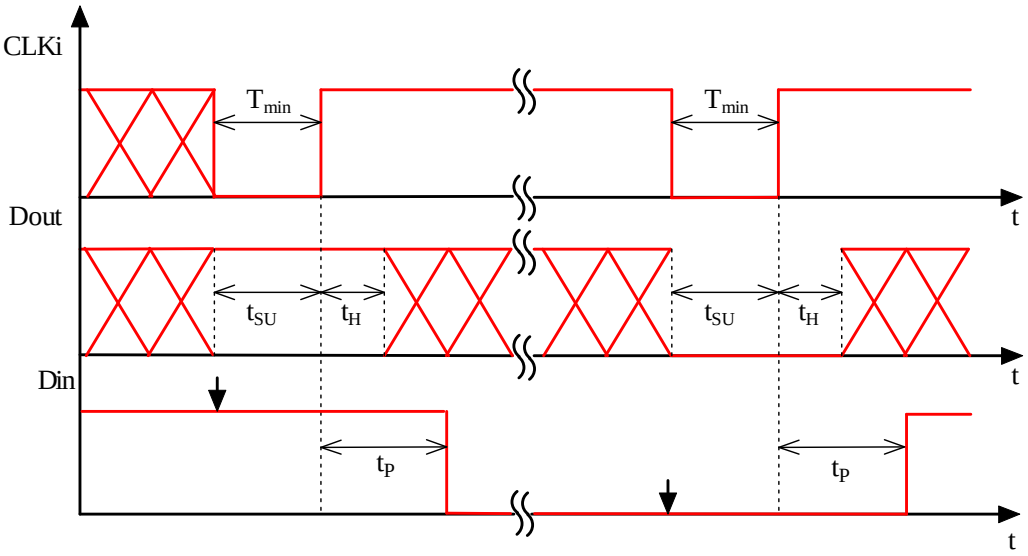

Figura 2.10: Temporizzazione comandata dalla routine `intButton`

La ISR si preoccupa di risolvere un altro problema, quello dei rimbalzi. Un pulsante è infatti un elemento meccanico che, all'atto della pressione o del rilascio, può essere sottoposto a microrimbalzi che causano dei glitch

nel segnale utile: ciò può dare dei problemi perchè potrebbero esserci delle commutazioni spurie. Di conseguenza è stata progettata una macchina a stati finiti che risponde al seguente requisito: si ha una commutazione dallo stato pulsante premuto allo stato pulsante rilasciato, solo se il pulsante risulta aperto in due campionamenti successivi; analogamente si ha il passaggio inverso quando il pulsante è premuto per due isatnti di campionamento contigui. Lo schema a stati è riportato nella figura 2.11, da cui si può facilmente sintetizzare la macchina usando le due variabili di stato A e B. Nella figura U è la variabile d'uscita (U = 0 indica il pulsante premuto e U = 1 il pulsante aperto) e IN quella di ingresso (IN = 1 significa pulsante campionato aperto e IN = 0 pulsante campionato chiuso).

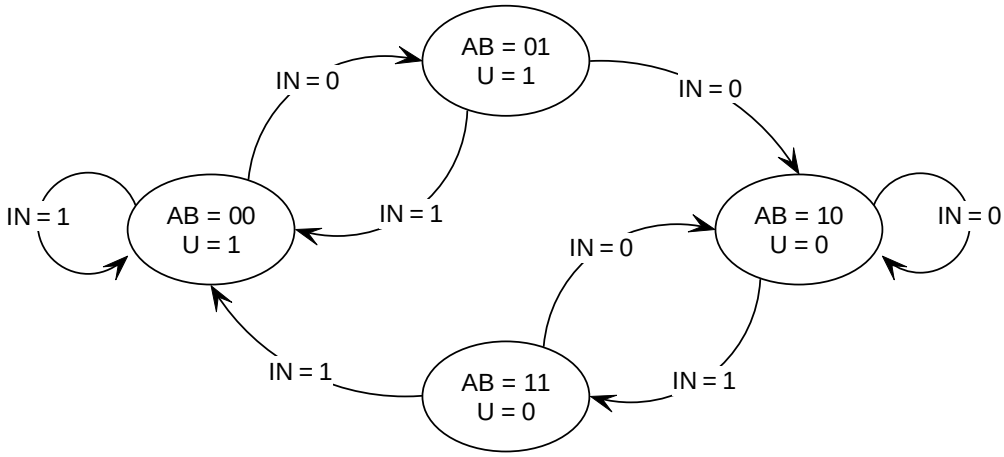

Figura 2.11: Macchina a stati per leggere lo stato dei pulsanti `intGPIO`

Per riuscire a generare opportunamente i segnali di controllo dello shift register si temporizzano le commutazioni inserendo un numero congruo di istruzioni tra di esse. Ciò richiede di tenere conto del clock del processore e calcolare di conseguenza il numero di istruzioni necessarie ad un corretto comando dell'integrato HEF4894B. Dal suo datasheet si ricavano i parametri indicati nelle figure, considerando il caso peggiore:

t_{SU} =5 ns t_H =5 ns T_{min} =5 ns
t_P =320 ns f_{clock} =5 MHz

Quando varia lo stato dei pulsanti, la ISR accoda un byte in cui è codificato tale stato ad una coda FIFO e genera l'evento `ChangeButton`. Questo è utilizzato dalla `ControlUnit`, che legge lo stato attraverso l'oggetto `Interface`. La routine `intButton` continua ad andare in esecuzione all'interrupt del timer, fino a quando ci sono cambiamenti dello stato dei pulsanti. Quando tutti i pulsanti sono stati rilasciati, disabilita il timer, riporta lo shift register allo stato di inizializzazione e maschera l'interrupt 9, ritornando così allo stato di partenza.

Vediamo com'è definito l'oggetto `Interface`:

```
typedef struct
{
    bool l_Intercom;
    bool l_ON;
    bool l_Nav_Passenger;
    bool l_Radio_Passenger;
    bool l_GSM_Passenger;
    bool l_Nav_Driver;
    bool l_GSM_Driver;
    bool l_SCO_Driver;
    bool l_RFCOMM_Driver;
    bool l_SCO_Passenger;
    bool l_RFCOMM_Passenger;
    bool l_Radio_Driver;
} LedStatusType;

typedef struct
```

```
{
    bool b_Intercom;
    bool b_ON;
    bool b_GSM_Driver;
    bool b_Radio_Driver;
    bool b_Nav_Driver;
    bool b_GSM_Passenger;
    bool b_Radio_Passenger;
    bool b_Nav_Passenger;
    bool b_Switch;
} SelectorType;

class UserInterfaceType
{
public:
    UserInterfaceType();
    void AsmInterface(int** ,int** ,int**);
    void WriteLeds(const LedStatusType);
    LedStatusType ReadLeds(void);
    //restituisce falso quando la coda degli stati dei bottoni è vuota
    bool ReadSelector(SelectorType&);
};

UserInterfaceType Interface;
```

Le funzioni membro sono molto semplici: leggono o scrivono lo stato dei led usando la struttura `LedStatusType`, in cui ad ogni campo corrisponde un determinato led; leggono lo stato dei pulsanti attraverso l'uso del tipo `SelectorType`, i cui campi booleani indicano lo stato dei pulsanti; permettono l'interfacciamento con una routine Assembly (la `intButton`) che deve usare direttamente dei puntatori per agire sulla coda degli stati dei pulsanti, come accadeva per le code di tipo `AudioQueue`. La coda per lo stato dei pulsanti è

usata per evitare che la `ControlUnit` possa perdere degli eventi.

2.4.2 I led

La scrittura dei led avviene attraverso l'opportuna funzione membro della classe `Interface`. Questa richiama una routine Assembly (sfruttando le regole di collegamento Assembly - C++ fornite nel manuale) che comanda i GPIO. Questa routine deve necessariamente mascherare gli interrupt 9 e 10, per evitare che le ISR per la lettura dei pulsanti vadano ad usare i GPIO. Notare che è necessario scrivere la routine in Assembly perché occorre tenere conto dei cicli di clock per realizzare la corretta temporizzazione. Per quanto riguarda questa temporizzazione, essa è molto semplice: deve dare su CLKo 12 cicli di clock e variare Dout per scrivere l'opportuna parola nello shift register; infine viene dato un segnale di strobe su STRo per far variare le uscite. I vincoli temporali sono praticamente gli stessi già citati nel paragrafo precedente con l'aggiunta che l'impulso di strobe deve essere sul livello basso con una durata minima di 80 ns.

2.5 Il thread ControlUnit

Questo thread ha il compito di dialogare con l'host controller della piconet Bluetooth, indicandogli le operazioni da svolgere. Queste sono semplicemente quelle che riguardano l'instaurazione e la gestione della piconet, cioè la ricerca di slave (ovvero i dispositivi Bluetooth sui caschi), l'identificazione degli slave trovati, l'attivazione di canali audio e così via. Affinché il codice svolga adeguatamente la propria funzione, è necessario conoscere nel dettaglio l'implementazione dell'applicativo che gira sul chip BlueCore; di conseguenza non vengono ulteriormente approfondite le tematiche appena citate in quanto esulano dal lavoro svolto.

Occorre però dare un breve cenno all'interazione che sussiste tra la `ControlUnit` e gli altri moduli dell'applicazione. Questo thread si occupa infatti di gestire l'interfaccia utente, recependo i comandi dati dall'utente attraverso i

pulsanti del pannello di comando e fornendo all'utente stesso un feedback alle sue azioni attraverso i led. Come conseguenza di questi comandi, il thread dialoga con il BlueCore con comandi ed eventi, accodando i primi in `BcCommand` e estraendo i secondi da `BcEvent`; provvede così a creare gli opportuni canali audio SCO nell'ambito della piconet. La `ControlUnit` deve poi dirigere l' operato di `AudioManager`, che si interfaccia direttamente con i flussi audio di ingresso e di uscita.

Queste informazioni sono fornite all'`AudioManager` attraverso `Connection-Status` e `ToMix`. Queste due variabili sono così dichiarate:

```
typedef struct
{
    bool SCO1;
    bool SCO2;
    bool Right;
    bool Left;
} ToMixType;

ToMixType ToMix[3];

typedef struct
{
    bool                        ActiveRFCOMM;
    bool                        ActiveSCO;
    unsigned int                Trusted_index;
    bd_addr_t                   Addr;
    ag_sco_connection_handle_t  SCO_hdl;
    bool                        SCOpending;
} ConnectionStatusType;

ConnectionStatusType ConnectionStatus[3];
```

Le tre componenti del vettore `ToMix` fanno riferimento alle tre uscite, che sono rispettivamente `ChOut`, `ScoOut1` e `ScoOut2`. Ciascun campo della struttura è posto uguale a `true` se la sorgente indicata da tale campo deve dare un contributo al rispettivo canale audio d'uscita. In pratica, se accade che `ToMix` è pari a

```
{{false,false,false,false},
{false,true,false,false},
{true,false,false,false}}
```

significa che è attiva la funzione di interfono: in `ScoOut1` va posto il flusso audio proveniente da `ScoIn2` e in `ScoOut1` l'audio di `ScoIn2`.

`ConnectionStatus` ha invece due sole componenti significative, la seconda e la terza, scelte per comodità di riferimento in quanto sono legate ai canali SCO1 e SCO2. Il campo booleano `ActiveSco` viene letto dall'AudioManager e indica l'attivazione o l'assenza del rispettivo canale SCO. Gli altri campi indicano la presenza di una connessione fisica (`ActiveRFCOMM`), l'handle della connessione (`SCO_hdl`) e altri parametri che sono usati nel codice per la gestione della piconet.

La parte relativa all'interfaccia utente viene gestita attraverso l'oggetto `Interface`, già descritto nella sezione 2.4, che offre le funzionalità per il pilotaggio dei led e la lettura dello stato dei pulsanti.

Infine, la sincronizzazione del thread è regolata dall'evento `Whatsup`, che dipende da due event bit: il primo settato dalla ISR `intButton` quando trova un cambiamento dello stato dei pulsanti; il secondo settato da `UARTreader` alla ricezione di un evento.

2.6 Il thread AudioManager

Questo thread si occupa della miscelazione dei flussi audio provenienti dal codec e dalla porta UART. Quindi i suoi dati di ingresso sono prelevati dalle code `ScoIn1`, `ScoIn2`, `RightChIn` e `LeftChIn` mentre gli output sono

restituiti in `ScoOut1`, `ScoOut2` e `ChOut`, come mostrato nella figura 2.1. Oltre ai campioni audio, il thread deve ricevere istruzioni riguardo a come operare sui dati audio: in particolare deve conoscere le sorgenti dalle quali attingere e quali miscelazioni occorre fare. Tali informazioni sono acquisite durante l'accesso a `ToMix` e `ConnectionStatus`.

L'`AudioManager` ha il corpo costituito da un ciclo infinito, all'inizio del quale c'è un'istruzione `VDK::PendSemaphore(kTick_Audio,0)`, che fa bloccare il thread ad ogni iterazione su un semaforo periodico che si attiva ad intervalli, la cui durata è fissata e compresa tra 5 e 10 ms. Quando viene sbloccato, il thread si preoccupa di leggere le due variabili `ToMix` e `ConnectionStatus`. Dalla prima ricava le seguenti informazioni, che vengono codificate in opportune variabili per consentire un più agevole utilizzo nel codice: quali flussi audio occorrono e come vanno miscelati. Nella seconda variabile il thread vede quali canali SCO sono attivi e si sincronizza con uno attivo. Con *sincronizzazione* si intende questo: poiché i vari dispositivi che costituiscono le sorgenti audio (il BlueCore e il codec, a sua volta collegato a radio, telefono cellulare, ...) non sono tra loro sincronizzati e comunicano in maniera asincrona, il numero di campioni provenienti da ogni sorgente sarà diverso ad ogni attivazione dell'`AudioManager`. Dato che in caso di miscelazione di due sorgenti audio per ottenere un terzo flusso occorre che il numero di campioni per ogni sorgente sia lo stesso, si procede ad una operazione di *zero padding*. Per fare ciò occorre decidere qual è il numero di campioni che si considera di volta in volta: esso può essere una costante ma si è preferito usare come parametro il numero di campioni presenti nel flusso audio di uno dei due caschi. Notare infatti che almeno uno dei due caschi è attivo nella comunicazione in tutte le possibili configurazioni di funzionamento. In questo modo si ottiene che per alcune sorgenti si dovrà operare uno zero padding mentre per altre ci saranno dei campioni in eccesso che non sono utilizzati, almeno a questa attivazione del thread. Le alternative sono allora due: buttare definitivamente i dati usando la funzione membro `AudioQueue::Throw()` o lasciare i campioni in eccesso per il prossimo ciclo di miscelazione. Questa scelta dipende dalla

qualità dell'audio che si ottiene in uscita e di conseguenza viene rimandata alla fase di collaudo (vedi capitolo 3).

Vengono allora caricati in memoria, in array di appoggio, solo i campioni che devono essere effettivamente utilizzati. Infatti potrebbe accadere che uno dei caschi mandi dati che però non sono richiesti nè dall'altro casco nè dall'uscita del codec; risulterebbe inutile copiare anche questi dati. La copia dei dati avviene usando semplicemente `AudioQueue::ReadArray()`. Si passa quindi alla fase di miscelazione, che viene ottenuta con una *somma con saturazione*. Questa consiste nel sommare i campioni audio in ingresso, che possono essere 2 o 3 [9], vedere se il risultato rientra nella dinamica del segnale e operare di conseguenza. Ciò significa che, poiché i campioni all'atto di essere sommati sono rappresentati come interi con segno su 16 bit, occorre usare la seguente formula, dove s indica il risultato della somma semplice e r il risultato finale:

$$r = \begin{cases} \min\{2^{15}-1, s\} & : \quad s \geq 0 \\ \max\{-2^{15}, s\} & : \quad s < 0 \end{cases}$$

Naturalmente le operazioni di somma vengono eseguite su campioni espressi in formato lineare e non compressi. Inoltre occorre notare che, essendo la lunghezza di parola del DSP pari a 16 bit, sommando dei campioni espressi su 16 bit si potrebbero avere errori di overflow e underflow. Una possibile soluzione è quella di dividere i campioni in ingresso per un opportuno coefficiente, sommarli, fare un controllo sulla dinamica e poi, in caso, rimoltiplicare il campione ottenuto per il coefficiente scelto. Si trova che, dovendo sommare n campioni, è sufficiente dividere ogni campione per n per assicurarsi che la loro somma sia nella dinamica consentita. Per trovare il risultato della somma con saturazione, occorre confrontare quanto ottenuto con $(2^{15}-1)/n$ se

[9] Le sorgenti audio sono 4. Si deve però considerare che se si ha a che fare con un solo dato è sufficiente un'operazione di copia, mentre non sarà mai necessario sommare 4 sorgenti dato che non si deve includere nel flusso di output l'audio proveniente dal dispositivo al quale tale output è indirizzato. Comunque, per come è stato organizzato il codice, è possibile anche sommare 4 diverse sorgenti.

abbiamo un numero positivo o con $-2^{15}/n$ se il numero è negativo. Naturalmente tutte queste operazioni hanno un costo computazionale e il peso maggiore lo avrebbe la divisione; questa operazione richiede infatti più cicli di clock per essere eseguita se il divisore non è potenza di 2. Si potrebbe allora pensare di usare sempre 2 o 4 come coefficienti ma la ALU del DSP ci mette a disposizione una saturazione di tipo hardware: settando il bit `AR_SAT` del registro di stato del core `MSTAT` si configura la modalità *ALU saturation* che esegue automaticamente l'operazione da noi voluta. In questa modalità, se viene superata la dinamica imposta dalla lunghezza di parola, il risultato viene bloccato ai valori estremi della dinamica.

Viene usata la somma con saturazione perchè permette un corretto trattamento dei dati audio. Infatti la somma brutale non è possibile per problemi di underflow e overflow, come si è visto, mentre la somma di campioni scalati di un opportuno fattore comporta l'inconveniente che, nel caso uno dei campioni fosse nullo, l'altro risulterebbe attenuato senza alcun motivo.

Una volta operata la miscelazione dell'audio, i dati vengono copiati dai nostri array temporanei nelle code d'uscita e viene settato il bit di evento `AudioDataOutBit`, che comunica all'`UARTwriter` la presenza di dati da essere trasmessi via UART. Questo è il funzionamento del thread ma non è ancora stato considerato il problema della disomogeneità nel formato dei dati. Infatti i campioni provenienti dal BlueCore sono compressi (vedi paragrafo 2.6.1) mentre i campioni dal codec sono interi con segno su 16 bit. Allora è necessario decomprimere i dati delle code `ScoIn1` e `ScoIn2` per poter effettuare la somma (che può essere fatta solo se tutti gli operandi sono in formato lineare) e successivamente comprimere i dati da accodare a `ScoOut1` e `ScoOut2`. Come spiegato nell'opportuno paragrafo, le operazioni di compressione e decompressione implicano un determinato impiego di risorse di calcolo ed è quindi bene limitare tali operazioni solo quando effettivamente necessario. Ad esempio in modalità interfono non occorre decomprimere e comprimere i campioni ma è sufficiente la copia dei campioni compressi. Si trovano quindi le seguenti condizioni, espresse come espressioni logiche C++, che decidono

la decompressione dei dati dalle code SCO. Quando ritornano un valore logico `true`, occorre la decompresione, rispettivamente di `ScoIn1`e `ScoIn2`:

```
ToMix[0].SCO1 || (ToMix[2].SCO1 && (ToMix[2].Left||ToMix[2].Right))
```

```
ToMix[0].SCO2 || (ToMix[1].SCO2 && (ToMix[1].Left || ToMix[1].Right)
```

Queste condizioni sono derivate dal seguente ragionamento: se la coda ScoX deve andare in output a `ChOut` (cioè `ToMix[0].SCOX == true`), deve essere decompressa; se SCOX deve essere mandata all'altro canale SCO e a quest'ultimo vengono mandati anche dati provenienti dal codec, SCOX deve essere decompressa poiché deve subire una miscelazione con altri campioni.

Analogamente si trovano condizioni che indicano la necessità di comprimere i risultati parziali per `ScoIn1` e `ScoIn2`.

2.6.1 Compressione e decompressione logaritmica

I campioni audio che devono essere trattati dall'host controller Bluetooth sono codificati secondo l'A-law. Questo è uno standard di compressione che permette di ottenere una buona codifica della voce riducendo il numero di bit necessari alla rappresentazione del valore audio campionato. Infatti si osserva sperimentalmente che la voce assume più frequentemente valori piccoli in modulo piuttosto che grandi: si opera allora con un quantizzatore non uniforme che permette una codifica più accurata per i valori bassi e che fa decrescere l'accuratezza all'aumentare del livello. Usare un quantizzatore non uniforme equivale all'uso in cascata di un *compressore* e di un quantizzatore uniforme; il compressore usato in questo caso ha una caratteristica ingresso-uscita di tipo logaritmico ed è per questo che la codifica A-law è detta logaritmica (si veda il paragrafo 3.7 di [Haykin]).

L'ambiente di sviluppo VDSP++ offre due funzioni nella libreria `filter.h` che permettono la compressione e decompressione di campioni audio secondo la specifica ITU G.711:

```
void a_compress(
    const short in[],  //array di input
    short out[],       //array di output
    int n              //numero di campioni
  )
void a_expand(
    const short in[],  //array di input
    short out[],       //array di output
    int n              //numero di campioni
  )
```

Queste funzioni trattano campioni compressi su 8 bit e campioni in forma lineare su 13 bit (incluso il bit di segno). Nel nostro caso occorre operare la decompressione dei campioni per poter operare la somma di più flussi audio (operazione che può chiaramente essere applicata solo su dati in formato lineare) e successivamente si usa la compressione sul risultato ottenuto. Gli altri dati che il thread AudioManager deve trattare sono però in forma lineare su 16 bit, quindi è necessaria una traslazione per poter utilizzare le due funzioni di libreria. Per ottimizzare il codice si è scelto di modificare i sorgenti delle due funzioni suddette, che sono forniti con l'ambiente di sviluppo. Ciò è vantaggioso perchè le due funzioni sono scritte in Assembly e trattano un vettore di dati per volta. Modificando un paio di istruzioni nelle routine `a_expand` e `a_compress` si evita di dover eseguire nel codice C++ ulteriori scorrimenti dei vettori dei dati. Le operazioni da fare sono poi banali in quanto si tratta di fare uno shift logico a destra o a sinistra usando il *barrel shifter*.

2.7 Inizializzazione

Quanto è stato descritto finora rappresenta il funzionamento del sistema a regime. Naturalmente occorre una fase di inizializzazione, che include la

configurazione delle periferiche del DSP e la creazione dei thread descritti in precedenza.

Il VDK offre due alternative per quanto riguarda l'inizializzazione dei thread: possono essere creati automaticamente dal sistema operativo al reset del processore oppure possono essere istanziati dinamicamente. In quest'ultimo caso, occorre comunque designare un thread che viene generato automaticamente e si preoccupa di far partire gli altri, chiamando semplicemente una funzione del nucleo. E' stato allora implementato il `MainThread`, che fa partire gli altri 4 thread dell'applicazione. Affinché vengano dapprima eseguite tutte le funzioni di inizializzazione e solo in seguito si proceda con la fase operativa, è stato usato l'evento `AllReady`, i cui event bit relativi vengono settati da ciascuno dei thread alla conclusione della propria fase di inizializzazione. Ciascun thread si blocca poi su `AllReady`; si sbloccano tutti solo quando tutti gli event bit sono settati.

Per quanto riguarda l'inizializzazione delle periferiche, essa avviene all'interno delle sezioni di inizializzazione dei vari thread. Si ha così che la porta UART e il relativo canale DMA sono configurati dall'`UARTreader`; l'`AudioManager` contiene le istruzioni per l'inizializzazioni del codec; l'interfaccia utente viene inizializzata dalla `ControlUnit`.

Capitolo 3

Test del sistema

Nel seguito vengono inizialmente descritti nel dettaglio gli strumenti di sviluppo usati. Ciò è necessario per la comprensione della parte successiva, riguardante il debug e il collaudo del sistema, svolti proprio con l'ausilio di tali strumenti.

3.1 Gli strumenti di sviluppo

Per la scrittura del codice è stato utilizzato l'ambiente integrato VisualDSP++, fornito con il kit di sviluppo della Analog Device. Le funzioni da esso fornite aiutano lo sviluppatore nella gestione di un progetto complesso come quello presentato nel presente lavoro.

Esso dispone dell'editor e del compilatore per codici Assembly, C e C++ e permette la facile interazione nello stesso progetto di routine scritte in linguaggi diversi. Una caratteristica molto utile è la presenza di un evoluto sistema per il linking dei file oggetto. Infatti ogni progetto deve essere accompagnato da un file di configurazione del processo di linking da eseguire: vanno indicati i segmenti di dati e di codice presenti nei vari file oggetto e il mappaggio sulla memoria che si intende fare con questi. Questo file può divenire piuttosto complicato all'aumentare delle dimensioni del progetto; è opportuno allora usare l'*Expert Linker*, un tool che genera automatica-

mente il file per il linker. In seguito è possibile modificare tale file attraverso un'interfaccia grafica user-friendly per adeguarlo alle proprie esigenze.

Per quanto riguarda il debug, è fornito il supporto per l'interfaccia JTAG (si veda il paragrafo 1.4), in modo da poter eseguire i test direttamente sul processore. Accanto a ciò, è presente un simulatore del processore per eseguire il debug senza scaricare il software sul DSP. In entrambi i casi è possibile usare l'ambiente grafico per analizzare il contenuto della memoria di programma, della memoria dati e dei registri di macchina, sia del core sia delle periferiche.

Ci sono poi strumenti più evoluti che permettono un'analisi dell'efficienza del codice o facilitano il debug in caso di utilizzo del VDK. Nel primo caso si tratta del *linear profiling* che valuta il tempo speso dal processore nell'esecuzione delle varie istruzioni e routine per dare la possibilità di individuare i punti critici. Come supporto al VDK ci sono la *Status Window* e la *History Window*: la prima da informazioni sullo stato attuale del sistema (stato dei thread, dei semafori, degli eventi); la seconda fornisce graficamente la storia dei vari thread, indicando le commutazioni di contesto avvenute e i cambiamenti dello stato di semafori, eventi e thread.

3.1.1 Il simulatore e l'interfaccia JTAG

Vengono ora analizzate nel dettaglio le differenze tra l'uso del simulatore e l'uso dell'interfaccia JTAG.

La presenza di un simulatore è stata fondamentale in alcune fasi del debug. Infatti, nonostante il DSP usato dia la possibilità di eseguire il debug sul chip, si è ricorso spesso al simulatore, soprattutto nei primi test del codice. I vantaggi sono molteplici: è infatti possibile bloccare l'esecuzione del programma in qualsiasi punto, sia a causa dell'esecuzione di un'istruzione particolare (breakpoint) sia a causa dell'utilizzo di una particolare locazione di memoria o di un registro (watchpoint). Quest'ultima opzione non è disponibile se il programma è in esecuzione sul processore reale, ma d'altro canto alcune prove devono necessariamente essere eseguite sulla scheda di sviluppo, ad esempio

quando occorre interfacciarsi con altri dispositivi come il BlueCore. L'uso del simulatore offre anche una soluzione ad un particolare inconveniente: nel caso di esecuzione sulla scheda di sviluppo la presenza di un breakpoint può provocare l'arresto definitivo del sistema, in quanto alcune periferiche dell'ADSP-2191 funzionano in maniera indipendente dal core (ad esempio il controllore DMA); l'arresto del programma può provocare il definitivo blocco di tali periferiche. Una conseguenza di questo fatto è la presenza, dopo il blocco del programma, di valori non significativi in alcuni registri delle periferiche. Ciò è accaduto con il controllore DMA: per verificarlo, è stata inserita nel programma un' istruzione che trasferiva il contenuto di uno dei registri di configurazione del DMA in un registro del core; ponendo un breakpoint all'istruzione successiva si è notata una discrepanza tra i valori dei due registri. Il registro del DMA indicava un'errore di buffer full, che non era presente un ciclo di clock prima dell'arresto, come testimoniava il registro del core. La spiegazione è semplice: il tempo intercorso tra il blocco del processore e la lettura dei registri di macchina, eseguita attraverso l'interfaccia JTAG, è stato sufficiente al riempimento del buffer del controllore DMA. Nel corso dei test è stato opportuno tenere conto di questo fattore per valutare l'effettivo stato della macchina negli istanti di blocco.

3.2 Il collaudo

La messa a punto è stata un'attività parallela alla scrittura del codice. Infatti lo sviluppo dell'applicazione è avvenuto per gradi: ciò ha permesso di prendere confidenza con gli strumenti e con l'hardware e ha facilitato la scoperta e l'eliminazione di errori, difficilmente individuabili nell'applicazione finale.

Le fasi principali dello sviluppo possono essere così individuate: la messa a punto della configurazione delle periferiche, la successiva realizzazione delle routine principali e l'unione finale del codice nel contesto del VDK. In questo modo si è cercato di testare singolarmente il codice dei vari thread e ISR, prima che questi fossero fatti agire in concorrenza. Ciò non è sempre stato

possibile dato che alcuni moduli necessitavano della presenza di altri per poter essere eseguiti significativamente: è il caso della `ControlUnit` che ha bisogno dei thread per la gestione della UART per comunicare con il BlueCore.

Quando il sistema è stato completato ed è stato in grado di operare, sono state effettuate prove per definire il valore di alcuni parametri in funzione delle prestazioni desiderate; tali prestazioni consistono nella qualità dell'audio in uscita e nel tempo di risposta agli eventi esterni. Come già accennato nel capitolo 2, occorre decidere la dimensione dei vari buffer per i dati (le code delle classi `AudioQueue` e `MsgQueue` e i buffer per le ISR), la politica di gestione della porta UART, i periodi dei semafori `Tick_UART` e `Tick_Audio` e alcuni aspetti della miscelazione dei flussi audio.

Dalle prove effettuate, risulta che l'applicazione riesce a smaltire i dati ad essa forniti dall'esterno, se si impone un periodo di esecuzione dell'`Audio-Manager` e dell'`UARTreader` di 10 ms. A riguardo è stato osservato un comportamento anomalo del VisualDSP++: per ottenere un periodo di 10 ms bisogna configurarne uno doppio; ciò è stato verificato facendo generare da un thread temporizzato da un semaforo periodico un'onda quadra su uno dei GPIO e andando a visualizzare tale segnale su un oscilloscopio.

Il dimensionamento dei buffer è avvenuto in conseguenza della scelta di un periodo di 10 ms per i semafori periodici. Considerando che il tempo di campionamento per l'audio è di 8 kHz, in un periodo di 10 ms si ricevono mediamente $10\text{ ms} \cdot 8\text{ kHz} = 80$ campioni per canale audio. Visto che la memoria disponibile lo permette, le code audio sono dimensionate per poter contenere 255 campioni, in modo da avere un ampio margine di sicurezza rispetto agli 80 campioni che ci si aspetta. Per le code di messaggi sono invece più che sufficienti 31 locazioni. Sui buffer per la porta UART sono invece presenti al contempo pacchetti audio dei due canali Sco1 e Sco2, eventi o comandi. Tenendo conto anche delle intestazioni dei pacchetti, i buffer hanno una lunghezza di 1024 parole.

Per la gestione della ricezione dei dati dalla porta UART, sono stati testati sia l'autobuffer sia la tecnica dei descrittori concatenati. Infine è stato

scelto l'autobuffer, in quanto il sistema riesce facilmente a seguire il flusso dei pacchetti e a non perderne nessuno; di conseguenza è stata preferita la modalità che implica minori configurazioni e minor dispendio di cicli di clock per la gestione.

Riguardo la miscelazione dei flussi audio, bisognava decidere cosa fare dei campioni audio non utilizzati nella miscelazione. Infatti, come descritto nel secondo capitolo, l'`AudioManager` si sincronizza su un flusso audio principale e fa in modo che tutti gli altri flussi contengano un numero di campioni pari a quelli del principale. Ciò è possibile con lo zero padding o con l'eliminazione dei campioni in eccesso. I campioni in eccesso possono essere buttati via definitivamente oppure possono essere utilizzati in successive miscelazioni. Nella pratica non si è notata una differenza sensibile nella qualità dell'audio prodotto nei due modi: di conseguenza è stato scelto in maniera arbitraria di eliminare definitivamente i campioni in eccesso.

Nelle due figure 3.1 e 3.2 sono illustrate due fasi significative della vita dell'applicazione. Esse sono state generate dalla History Window durante due prove effettuate con la scheda di sviluppo. E' possibile osservare l'andamento dell'esecuzione, evidenziando lo stato dei vari thread (indicato dal colore delle barre) e gli eventi che si succedono nel tempo (indicati dalle frecce colorate). Tra questi è interessante notare come alcuni eventi `SemaphorePosted` e `EventBitSet` causino una variazione dello stato dei thread che erano bloccati su tali semafori ed eventi. La scala temporale sulle ascisse è espressa in `Ticks`: questa è l'unità di misura del VDK, che può essere configurata e nel nostro caso è pari a 500 μs. Nella figura 3.1 è visualizzata una finestra temporale di circa 1 tick, nella figura 3.2 una di circa 20 tick. La barra indicata con `Idle Thread` è verde quando nessun thread è pronto e quindi il processore è in idle; grazie alla presenza di questo thread fittizio si può osservare come ci sia sempre un unico thread in esecuzione.

Nella prima figura è presentata la fase di inizializzazione. Il `MainThread` crea un thread alla volta; il thread creato si blocca sull'evento `AllReady` e cede nuovamente il passo al `MainThread`. Non è visibile la fine dell'inizializzazione

in cui il `MainThread` viene distrutto; questa avviene dopo molti cicli di clock, in quanto per l'inizializzazione del codec occorre un tempo dell'ordine dei μs. Si noti infatti che il thread `AudioManager` è in stato `Sleeping`, cioè la sua esecuzione è bloccata per un determinato intervallo di tempo, dopo il quale sarà svegliato: questa attesa è necessaria per fornire un opportuno segnale di reset al codec.

Nella seconda figura è mostrato il funzionamento a regime. E' in corso un collegamento audio con almeno uno dei due caschi: si nota infatti l'andamento periodico con il quale vanno in esecuzione `UARTwriter`, `UARTreader` e `AudioManager`.

3.2.1 Risoluzione dei problemi

Durante la fase di collaudo è stato riscontrato un problema. L'applicazione funziona secondo le specifiche ed è stata completamente testata la funzionalità di interfono; accade però che l'audio si interrompa, dopo un intervallo di tempo in cui tutto pare funzionare correttamente. Dall'analisi dello stato del processore prima e dopo questi blocchi, si è concluso che l'interruzione del trasferimento audio è dovuto ad un blocco del controllore DMA: è stato allora necessario un riesame completo di tutto il codice, ma non sono stati trovati errori. Contattando la casa produttrice del DSP, si è venuti a conoscenza di un'anomalia non documentata sui datasheet e sul manuale hardware. Tale anomalia consiste nel possibile blocco del controllore DMA se è in corso un trasferimento DMA su una delle SPORT e nel contempo viene eseguita un' operazione di lettura/scrittura su un registro di I/O o viene eseguita una lettura sulla memoria esterna. Tale problema hardware può essere aggirato solo adottando una diversa politica di gestione della SPORT: è stata allora abbandonata la gestione a DMA ed è stato riscritto parte del codice per implementare il controllo ad interrupt. A causa della natura del codec, occorre rispondere agli interrupt della SPORT continuamente, con una cadenza di

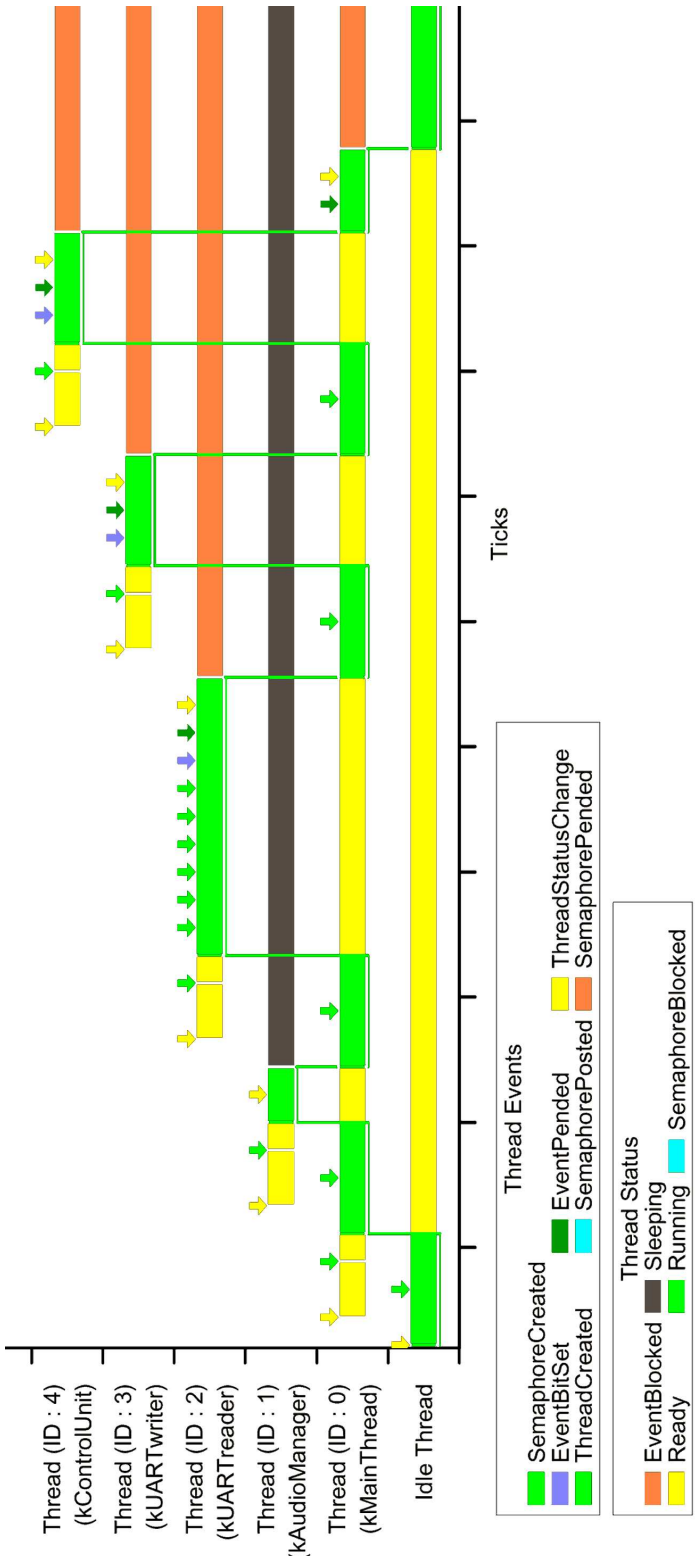

Figura 3.1: La fase di inizializzazione

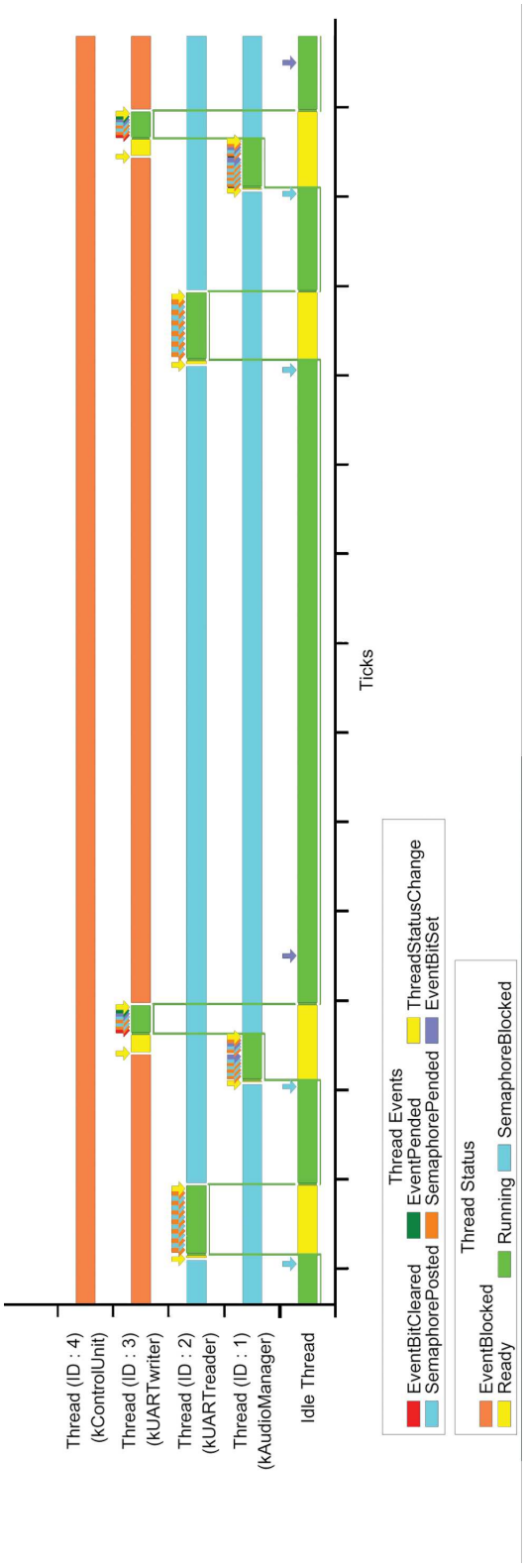

Figura 3.2: Funzionamento a regime con trasferimento audio

circa 1.3 μs[1]: ciò implica che non possono essere presenti sezioni di codice in cui gli interrupt siano disabilitati e che impieghino più di 1.3 μs per essere eseguite. Tra queste sezioni sono incluse alcune primitive di nucleo del VDK; allora è stato necessario abbandonare l'utilizzo del VDK in favore di una soluzione leggermente diversa.

In realtà, l'architettura dell'applicazione risulta invariata. E' stato però necessario scrivere un primitivo schedulatore, che si attivi ogni 10 ms e mandi in esecuzione le varie routine. Queste routine non sono altro che la trasposizione dei corpi dei vari thread in semplici funzioni C++. E' stato necessario scrivere anche un'apposita classe che sostituisse gli eventi del VDK: questo per limitare al minimo le modifiche al codice, che comunque funziona correttamente a parte l'anomalia hardware suddetta; questi nuovi eventi sono necessari, come quelli del VDK, per una corretta comunicazione tra i moduli dell'applicazione.

La nuova soluzione adottata non va quindi vista come uno stravolgimento di quanto descritto finora, dato che l'applicazione rimane sostanzialmente la stessa: le uniche modifiche sostanziali sono quelle che riguardano la gestione della SPORT e l'uso del `TIMER2` per sincronizzare lo schedulatore. In ogni caso, l'applicazione che fa uso del VDK rimane valida per una nuova release del DSP che elimini l'anomalia hardware.

E' importante notare un'ulteriore differenza tra le due versioni dell'applicazione. Nella prima infatti la gestione della SPORT occupava in maniera minima il core del processore grazie all'uso del controllore DMA; nella seconda versione vengono invece generati continuamente degli interrupt, impegnando il core nell'esecuzione delle ISR relative. Tenendo conto della lunghezza delle ISR e della frequenza con la quale vengono generati gli interrupt si trova che la gestione della SPORT occupa il core per circa il 28% del tempo. Ciò sarebbe inaccettabile in altri contesti ma non si può fare altrimenti a causa del problema hardware già citato. In ogni caso la poten-

[1]Inafatti la connessione con il codec ha una baud rate di 12.288 Mbit/s e vengono trasferiti 16 bit ad ogni interrupt: 16/12.288 Mbit/s $\cong$ 1.3 μs.

za di calcolo residua è sufficiente per svolgere tutti gli altri compiti richiesti all'applicazione.

3.3 Sviluppi futuri

L'applicazione descritta finora svolge in maniera corretta i compiti per la quale è stata creata. E' interessante a questo punto analizzare quali potrebbero essere gli sviluppi futuri e quali sarebbero le conseguenti modifiche da apportare a quanto già realizzato.

Dapprima si osservi che l'utilizzo di una futura release dell'ADSP-2191, in cui sia eliminata l'anomalia hardware, consentirebbe l'utilizzo della versione originale dell'applicazione.

Per quanto riguarda la versione attuale, il software è stato progettato in maniera modulare e ciò implica una facile modifica del codice. Infatti ciascun modulo può essere completamente sostituito a patto di rispettare le regole di interfacciamento con gli altri blocchi del sistema. La sostituzione dei moduli che gestiscono la comunicazione con l'esterno potrebbe essere necessaria in caso si decidesse di cambiare parte dell'hardware, ad esempio se si sostituisse il codec AD1885 con un altro modello.

Inoltre, è interessante considerare quali nuove funzionalità sarebbe possibile aggiungere al sistema modificando la parte dell'applicazione che si occupa dell'elaborazione dei dati. Il DSP è ottimizzato per le elaborazioni numeriche dei segnali e l'aggiunta di nuove operazioni sui campioni audio sarebbe possibile sia in termini di peso computazionale sia in termini di tempi di sviluppo. Infatti, essendo progettato per questo tipo di applicazioni, l'ambiente di sviluppo è dotato di funzioni di libreria apposite che permettono la gestione dell'audio.

Con poco sforzo sarebbe possibile aggiungere le funzionalità di filtraggio dell'audio e di cancellazione dell'eco. Ciò migliorerebbe la qualità del servizio offerto dal sistema senza bisogno di cambiare l'hardware.

Inoltre sarebbe interessante implementare il riconoscimento vocale per

permettere all'utente di dare comandi senza staccare le mani dal manubrio del motoveicolo. Ciò renderebbe il sistema più sicuro e renderebbe possibile l'eliminazione dell'interfaccia a pulsanti.

Conclusioni

In questo lavoro è stato descritto lo sviluppo di un'applicazione per l'ADSP-2191, un processore DSP a virgola fissa. L'obiettivo è stato quello di creare un sistema per la gestione di una rete wireless Bluetooth per un motoveicolo.

A partire dall'architettura del sistema complessivo, composto da dispositivi Bluetooth, sorgenti audio e DSP, sono state definite le specifiche richieste all'applicazione. Essa deve essere in grado di recepire i comandi che l'utente invia attraverso un'interfaccia a pulsanti posta sul cruscotto e deve agire di conseguenza; deve poter comunicare con il dispositivo Bluetooth posto sulla moto, che funge da master della piconet, e dirigerlo nella creazione e gestione della piconet, che include come slave i nodi sui caschi. Una volta creata la rete il DSP è incaricato di miscelare i flussi audio provenienti dai caschi e dalle sorgenti audio esterne e indirizzarli alle uscite opportune. Così il sistema complessivo permette la creazione di una rete wireless a bordo della moto e rende possibile la comunicazione tra i due motociclisti e tra ciascun motociclista e gli altri dispositivi, cioè il telefono cellulare, il navigatore satellitare e la radio.

Il secondo passo è stata l'implementazione del software per il DSP a partire dalle specifiche. Sono state analizzate le varie alternative offerte dall'hardware del dispositivo e sono state compiute una serie di scelte, che hanno portato alla struttura descritta nel capitolo 2.

Il software così ottenuto è stato testato e messo a punto: in questa fase abbiamo apportato le modifiche necessarie ad un corretto funzionamento del sistema.

E' stato finalmente ottenuto un adeguato controllo della rete Bluetooth descritta nel capitolo 1. L'applicazione presentata alla fine del lavoro soddisfa tutte le specifiche e può essere utilizzata nella realizzazione del prototipo finale.

Il sistema è stato inoltre progettato in modo che nuove funzionalità, come il filtraggio dei flussi audio e la cancellazione dell'eco, possano essere facilmente aggiunte modificando alcuni moduli del software. Per come è stata progettata l'applicazione, tali modifiche non richiederebbero un eccessivo tempo di sviluppo e cambiamenti hardware e potrebbero migliorare sensibilmente la qualità del servizio offerto dal sistema.

Bibliografia

[Mettala 99] R.Mettala, "Bluetooth Protocol Architecture," Bluetooth White Paper, Version 1.0, 1.C.120/1.0, 25 Agosto 1999

[Haartsen 98] J.Haartsen, M.Naghshineh, J.Inouye, O.J.Joeressen, W.Allen, "Bluetooth:Vision, Goals, and Architecture," Mobile Computing and Communications Review, Volume 1, Number 2, 1998, pag. 1-8

[Bisdikian 01] C.Bisdikian, "An overview of the Bluetooth Wireless Technology," IEEE Communications Magazine, Dicembre 2001, pp.93-94

[Bluetooth 1] Specification of the Bluetooth System Version 1.1, Volume 1: Core, February 2001

[Bluetooth 2] Specification of the Bluetooth System Version 1.1, Volume 2: Profiles, February 2001

[Profile] Basic Printing Profile - Interoperability Specification, Revision 0.95a, 10 Maggio 2001

[JTAG] IEEE Std 1149.1-2001. IEEE Standard Test Access Port and Boundary-Scan Architecture

[AC'97] Audio Codec '97, Component Specification, Revision 1.03, 1996, Intel Corporation, www.intel.com

[Haykin] S.Haykin, Communication Systems, 4th edition, New York, John Wiley & Sons, 2001