I registri dei dispositivi sono memory mapped. Tali registri sono di 32 bit come gli altri presenti in RAM tuttavia essi portano solo un byte di informazione. Per questo motivo i primi 4 bit del registro sono usati per programmare le interruzioni, l’8 bit serve a distinguere tra libero e occupato e quelli da 0 a 7 contengono le informazioni. Il bit 31 indica abilita le interruzioni ad alta priorità mentre il bit 30 vuol dire abilita le interruzioni normali (quindi il dispositivo può mandarle di entrambi i tipi, ma è il processore a sceglierlo scrivendo 0 o 1 nel primo o nel secondo bit (quindi questi bit sono impostati dal processore non dal dispositivo). Quindi i dispositivi che hanno entrambi in quei bit non mandano richieste di interruzione. Ovviamente non ha senso programmare un dispositivo per fargli mandare interruzioni di entrambe le priorità contemporaneamente. I bit 29 e 28 corrispondono allo stato del dispositivo rispetto alle interruzioni (essi infatti assumono il valore 1 nel momento in cui il dispositivo manda una richiesta di interruzione): il terzo bit (o meglio il 29) corrisponde alla interruzione abilitata dal bit 31 (FIRQ) mentre il quarto (28) corrisponde all’interruzione abilitata dal 30 (IRQ). Quindi solo i primi due bit sono inizializzati dal sistema durante l’accensione, i successivi due sono invece quelli letti dall’handler per capire se c’è stata richiesta di interruzione oppure no.

Nel caso del dispositivo di input, vogliamo che esso mandi le Fast Interrupt (quindi le veloci): servirà che durante l’accensione il bit 31 sia impostato a 1 e il 30 a 0. I bit dal 27 al 9 vengono ignorati, mentre negli ultimi 8, per quanto riguarda il dispositivo di input, seguono la convenzione che l’8 bit presenta 1 se non c’è alcun carattere digitato, 0 altrimenti, e i 7 successivi contengono la codifica ascii del pulsante premuto.

Non c’è un sistema di buffer nel sistema di input, quindi se la lettura del registro memory mapped non avviene in fretta c’è il rischio di perdere dati inseriti dall’utente (poiché vengono sovrascritti quando sono premuti nuovi tasti). L’acknowledgement da parte dell’handler provoca il passaggio dell’8 bit dal valore 0 al valore 1 (oltre che la copiatura del carattere dal registro al processore).

Per i dispositivi di output la richiesta di interruzione avviene al termine della stampa (quando il dispositivo è nuovamente libero). Inoltre in questo caso il dispositivo di output solleva le IRQ ma non le FIRQ. Per sapere se la stampa è terminata basta anche in questo caso guardare l’8 bit (che sarà 1 a stampa terminata, 0 se è ancora in corso).

Per migliorare il funzionamento dei dispositivi, però, introduciamo noi i buffer in RAM. Ci serve innanzitutto sapere quanta memoria vogliamo dedicare, nel nostro esempio utilizziamo buffer da 100 caratteri. Bisogna identificare due zone di memoria (poiché i dispositivi sono due) da 100 bytes ciascuna da usare come buffer. Il nostro handler si preoccuperà di leggere i caratteri in ingresso dal dispositivo di input e inserirlo nel buffer dedicato ad esso, se il buffer si riempie ignoriamo le interruzioni successive sollevate dal dispositivo di input (quindi possiamo tenere al massimo 99 caratteri, poiché il 10simo è lo 0, il terminatore di stringa).  
Le applicazioni per leggere dal buffer e permetterne lo svuotamento) useranno una system call di tipo READ (tramite l’istruzione SWI). La SWI legge e svuota il buffer fino a un certo punto, a seconda della richiesta dell’applicazione. Se l’applicazione vuole leggere più caratteri del contenuto del buffer allora lo legge e azzera tutto, altrimenti taglia e incolla soltanto la porzione da essa richiesta.

Il buffer non fa quindi parte del dispositivo IO, ma è alimentato dal sistema operativo.

Per la stampa funziona analogamente. L’unica differenza che è necessaria la system call WRITE anziché la READ (quindi i bit di motivazione della SWI dovranno essere diversi). Se il buffer è vuoto, però, ovviamente la stampa “fallisce”. Per le applicazioni, quindi, la stampa finisce nel momento in cui hanno inserito la loro merda nel buffer di sistema, è il sistema operativo che si deve poi occupare di assicurarsi che il risultato sia stampato a video. Se il buffer si riempie, invece, una possibilità è di sollevare un errore, mentre un alternativa è di mettere in pausa il programma e avviarne un altro che non richiede l’accesso al buffer di uscita e farlo ripartire nel momento in cui il buffer si è svuotato.   
Qualsiasi dispositivo POSIX utilizza l’alternativa “bloccante” (almeno di default, poi può essere modificata per sollevare invece un errore) perché, in genere, è migliore per l’esecuzione delle applicazioni.

Il nostro prof ha fatto sì che, nel simulatore amber, in un indirizzo predefinito della ram siano presenti i buffers e che abbiano tutti lo stesso tipo di struttura (anche se in realtà sono solo due buffer). I buffer sono definiti da elementi di descrizione, realizzati da quattro celle di memoria consecutive. Il primo elemento contiene l’indirizzo di partenza (è un puntatore a carattere), il secondo elemento è un numero che ci indica la dimensione del buffer, nel nostro esempio darebbe 100 (quindi se volessimo modificarne le dimensioni basterebbe cambiare la memoria allocata e questo numero), abbiamo poi un indice di inserzione e un indice di estrazione (questi due numeri possono variare da 0 a 99) che indicano la posizione in cui inserire il prossimo carattere e la posizione da cui estrarre il prossimo carattere.

L’indirizzo di partenza sarà quel che sarà (potrebbe anche essere 3600), la dimensione sarà 100 e i due indici vanno inizializzati a 0.

Ora proviamo ad abbozzare i gestori delle interruzioni (FIRQ e IRQ, che corrispondono al dispositivo di ingresso e di scita).

//Siamo costretti a partire dalla cella 28 perché lì si trova l’elemento del vettore di interruzioni che corrisponde alla firq, tuttavia poiché quello è l’ultimo elemento del vettore possiamo mettere le istruzioni direttamente dopo senza fare un branch, inoltre la firq ci permette di avere i registri tra l’8 e il 14 duplicati, quindi possiamo conservare i valori senza dover salvare nulla. Tutto questo risparmia cicli di clock.

FP (R11) <-rmm\_input //questi valori sono inizializzati nella modalità giusta durante lo start.

R10\_firq <-buffers

Firqh: 28 LDR R8, [FP]

ANDS R9, R8, #536 870 912 // questo numero corrisponde a 0x20000000, il risultato dell’AND potrà essere o 0 o il valore 200000000 in esadecimale (se il 31 e il 29 bit sono a 1), se il risultat è 0 c’è stato un errore.

MOVEQ PC, LP

ANDS R9, R8, #128 //questo ci permette di controllare l’8 bit del registro, se infatti è 0 vuol dire che c’è un carattere da leggere nel dispositivo di ingresso

MOVNE PC, LP

AND R8, R8, #127 //questo AND permette di estrarre il carattere nel registro, infatti tutti i bit oltre il 7mo vanno a zero

STM SP!, R0, R1, R2, LP //salvo i registri non duplicati, il default per ! è “increment after”

LDR R9, [R10] //si ricorda che nell’R10 delle FIRQ c’è l’indirizzo dei buffers

LDR R2, [R10,#+4] //in R2 si mette le dimensioni

LDR R0, [R10,#+8] //in R0 si mette l’indice di inserzione

ADD R1, R0, #1 //sommo all’indice di inserzione la costante 1

CMP R1, R2

MOVGE R1, #0 //se dopo l’incremento R1 diventa >= R2 allora bisogna ripartire da 0

LDRB R2, [R9 + R1] //la B fa sì che la lettura prenda un solo byte

CMP R2, #0 //mi aspetto di trovare 0 perché l’ultimo valore di una stringa deve sempre essere il terminatore

LDMDBNE SP!, R0, R1, R2, PC^ //LDM con Decrement Before se R2 Not Equals 0

STRB R8, [R9 + R0] //salvo il carattere nel buffer

STR R1, [R10, #+8] //salvo R1 come nuovo indice di inserzione

LDMDB SP!, R0, R1, R2, PC^

Per quanto riguarda l’IRQ, la prima istruzione dovrà senz’altro essere un BRANCH a un’altra zona di memoria, che chiameremo irq\_h.

Irq\_h: 104: //in questo caso non abbiamo registri duplicati, tranne il Link Pointer e lo Stack Pointer

STM SP!, R0, R1, R8, R9, R10, FP, LP

MOV FP, #rmm\_input //gli indirizzi sono uguali a sopra ma in teoria dovrebbero essere diversi

MOV R10, #buffers //ad esempio qui non è veramente buffers ma buffers+16

LDR R8, [FP, #+4] //infatti qua mettiamo FP+4

ANDS R9, R8, #268 435 456 // 0x10000000 //il bit 28 deve essere a 1, perché solo così ha sollevato l’interruzione

LDMDBEQ SP!, R0, R1, R8, R9, R10, FP, PC^

ANDS R9, R8, #128 //si guarda l’8avo bit, se questo è 0 la stampa non è ancora terminata

LDMDBEQ SP!, R0, R1, R8, R9, R10, FP, PC^

LDR R9, [R10, #+16] //ecco qui il +16 dopo buffers

LDR R0, [R10, #+28] //indice di estrazione del buffer, 28 = 16+12

LDRB R8, [R9, +R0] //metti in R8 il valore nella cella a indice del buffer + indice di estrazione

CMP R8, #0 //confronto R8 col terminatore di stringa, per capire se va stampato

LDMDBEQ SP!, R0, R1, R8, R9, R10, FP, PC^

MOV R1, #0

STRB R1, [R9, +R0] //il valore da stampare viene sostituito nel buffer con il valore 0, è uno svuotamento

ADD R0, R0, #1 //si aggiorna l’indice di estrazione (il buffer funziona come una queue)

LDR R1, [R10, #+20] //in R1 viene inserita la dimensione del buffer di uscita

CMP R0, R1

MOVGE R0, #0 //ci si è calcolati il nuovo indice di estrazione

STR R0, [R10, #+28] //lo si mette a posto

STRB R8, [FP, #+4] //si inserisce il byte nel posto giusto del registro memory mapped per stamparlo

LDMDB SP!, R0, R1, R8, R9, R10, FP, PC^