Questo mese niente programmi da parte dei lettori, ma una «passeggiata didattica» nei meandri del console.device di Amiga. In un certo senso si tratta solo di un antipasto: credo che il problema andrebbe comunque trattato in maniera più scientifica, visitando non solo qualche device di tanto in tanto (... e in generale) ma effettuando una trattazione globale del «devicing» di Amiga ponendosi in un livello d'astrazione ben maggiore. Vedremo...

Il Console.device

di Andrea Renzi - Roma

Nelle righe che seguono vi parlerò di come creare una comune console da utilizzare nei nostri programmi, prima di addentrarci all'interno dell'argomento però, qualche informazione generale.

Sicuramente ciascuno di voi avrà visto in funzione decine e decine di console e saprà di che si tratta ma si sa, nessuno è perfetto, per ciò ribadiamo il concetto: molto semplicemente, e sinteticamente, una console è un'interfaccia tra noi e la macchina che ci permette di comunicare con essa prevalentemente attraverso la tastiera, meglio ancora, con un esempio, è una console la finestra di un word processor come lo è la window del cli.

Vedremo come sia possibile, anche per noi comuni mortali, utilizzare le potenzialità che Amiga ci offre in questo senso e, cosa più importante, utilizzarle a nostro vantaggio.

In realtà ci sono due modi di servirci di una console, uno, piuttosto semplice, consiste nell'utilizzo dei file handle che ci permetteranno con una sola Open(); di crearci la console, tuttavia la sua semplicità va senza dubbio a discapito dell'efficienza, e per questo che vi propongo come valida alternativa un secondo metodo.

Vedremo infatti come, guadagnando in efficienza, ci sarà possibile fare ciò che desideriamo passando per il console.device, ma prima...

L'allegra banda dei device

Per device viene intesa una particolare struttura di controllo che ci permette una sorta di interfacciamento con entità di Input/Output, come ad esempio la porta seriale, una console, la traccia di un disco.

Possiamo aprire, chiudere, estrometterе e sopratttutto comunicare con i device con funzioni e particolari «strutture» create appositamente per lo scopo, ma vediamo una cosa per volta.

Innanzitutto affinché possiamo accedere al device è necessario che sia stato aperto con una OpenDevice, la sua sintassi e:

OpenDevice(DEVICENAME,unit,ioRequest, flags)

DEVICENAME rappresenta il nome del device che vogliamo aprire, ad esempio "console.device".

Unit si riferisce all'unità del device che vogliamo utilizzare, ad esempio UNIT_VBLANK o UNIT_MICROHZ nel caso del timer.device, e ovvio che se il nostro device non è composto da più unità questo campo debba essere impostato a 0.

IORequest, invece, è il puntatore ad un normale IOStdReq che avremo previdentemente inizializzato e riempito con informazioni utili, in questo modo per le future richieste di IO sarà necessario utilizzare il solo blocco di I/O, risparmiandoci un bel po' di lavoro. Infine il campo flag varia di volta in volta per cui non si può dargli un significato assoluto, in generale è un flag (letteralmente bandiera), settabile al bisogno, è bene per il momento settare anche lui a «0».

Anche per la comunicazione con i device utilizzeremo la struttura IOStdReq o richiesta standard di I/O; come potete vedere dalla figura contiene una quantità di dati piuttosto ampia, tuttavia non dovremo ogni volta inizializzare tutti i suoi campi, come già detto, inoltre exec da bravo riempirà i due campi IO_device ed IO_Unit che non dovranno essere neppure modificati, non dovremo far altro quindi che utilizzare il solo blocco IORequest ogni volta che sarà necessario dialogare con il device.

Alcuni device hanno bisogno di un numero maggiore di informazioni di quelle utilizzate nella struttura standard di IO, in questi casi sono stati ideate apposite strutture alternative con tutti i campi necessari; vedremo più avanti come il nostro sia uno di questi casi.

Come potete vedere osservando la struttura dell'IOStdReq o meglio del blocco di IO troverete un campo identificato come IO_Command commentato con un semplice «comando», in effetti si tratta del campo dedicato al comando da inviare scelto nel set di comandi appositamente studiato per la gestione dei device.

Con questo set di comandi, punto di forza della gestione dei device, potremo fare diverse cose, dalle semplici funzioni di lettura/scrittura a funzioni strettamente ideate per i singoli device.

Non mi dilungherò ulteriormente su questo argomento anche perché se ne è

```
struct IOStdtReq
{
        struct Message io_Message;   /* struttura per la comunicazione tra task */
        struct Device * io_Device;   /* Campi utilizzati da Exec */
        struct Unit * io_Unit;       /* */
        UWORD io_Command;            /* Comando */
        UBYTE io_Flags;              /* Flag variabile */
        BYTE io_Error;               /* Codice dell'errore */
        ULONG io_Actual;             /* Dati gia' letti */
        ULONG io_Length;             /* Lunghezza dei dati */
        APTR io_Data;                /* Dati veri e propri */
        ULONG io_Offset;             /* Offset */
}
                    Struttura dell'IOStdReq
```

Figura 1

parlato in modo piuttosto esaustivo nel numero 88 di MC che vi consiglio di andare a ripescare, è giunto il momento!

A questo punto non ci resta che entrare nel merito dell'argomento principale: il console.device.

Il nostro obiettivo: il console.device

Il console.device funziona in linea generale come un qualsiasi altro device, non dovrebbero quindi esserci problemi particolari per accedere ad esso tuttavia i più pigri possono andare a vedersi lo stralcio di codice in figura 2 che mostra come aprire correttamente il console.device.

Come accennato nelle righe precedenti il console.device non si accontenta del solito IOStdReq, usufruisce invece di una struttura appositamente fatta su misura per lui, la ContIOBlocks.

Al suo interno troviamo altre due strutture che sono due normali IOStdReq, uno per la lettura ed uno per la scrittura.

Inizializzeremo quindi questa struttura al momento dell'apertura dei device.

Ma non basta!

Abbiamo bisogno infatti di una finestra perché la console sia utilizzabile, una normalissima window intuition farà al caso nostro.

Inizializziamo quindi la struttura di una window e specifichiamo il suo puntatore nel IORequest (vedere la sintassi di OpenDevice) che chiameremo nella OpenDevice(); ossia:

```
.....
c→writeReq→io_Data = (APTR) window;
c→writeReq→io_Length = sizeof (struct Window);
OpenDevice("console.device", 0, c→writeReq, 0);
.....
```

dove writeReq sarà il puntatore di un IOStdReq.

In figura 3 è riportata la funzione NewCon che ci permetterà con una sola chiamata di assegnare ad una qualsiasi window intuition la nostra tanto sospirata console! Comodo, vero? Il source parla da solo per cui non spendo altre parole a riguardo.

Prima di passare all'utilizzo vero e proprio del nostro gioiellino però, un'ultima precisazione importante: se alleghiamo un IDCMP alla nostra window, esso avrà priorità assoluta sulla console per cui specificando ad esempio RAWKEYS o VANILLAKEYS, non riusciremo a far giungere al console.device alcun messaggio vanificandone la sua utilità!

Possiamo senz'altro dire a questo punto che abbiamo nelle nostre mani un'autentica console a tutti gli effetti, non ci resta che vedere come riusciremo a comunicare direttamente alla console il nostro volere.

Anche questa operazione non si discosta dalle analoghe utilizzate dai suoi «colleghi», inizializzeremo il solito blocco IORequest (letteralmente richiesta di I/O) e lo invieremo, unica sostanziale differenza sarà che dovremo riferirci all'IORequest contenuto nell'IOStdReq per la lettura, se vorremmo leggere dati, ed a quello per la scrittura se vorremmo scrivere dati.

Vi ricordo che entrambi gli IOStdReq sono contenuti a loro volta nel principale ConIOBlocks che abbiamo inizializzato all'apertura del device, se la cosa non è chiara il listato dimostrativo di figura 3 sarà senz'altro illuminante.

Anche su quest'ultima operazione va fatta un'altra considerazione: comunemente abbiamo due modi per inviare la richiesta di I/O il SendIO e il DoIO, essi differiscono per un unico particolare, il primo invia una richiesta asincrona mentre il secondo una richiesta sincrona. Mi spiego meglio: utilizzando il tipo di richiesta asincrona una volta inviato il block il nostro task porta ad occuparsi di

```
/* 'c'e' un puntatore alla struttura ConIoBlocks
/*  writeReq e' un puntatore all'IOStdReq per la lettura

errore = OpenDevice("console.device",0,c->writeReq,0)
if (errore != 0) exit(0);
```

Come aprire il console.device

Figura 2

```
/* struttura ConIOBlocks */

struct ConIOBlocks {
        struct IOStdReq *writeReq; /* per scrivere */
        struct IOStdReq *readReq;  /* per leggere  */
        struct MsgPort *tpr;

        };
extern struct IOStdReq *CreateStdIO( );
extern struct MsgPort *CreatePort( );

struct ConIOBlocks *
NewCon(w)
        struct Window *w; /* window da utilizzare */
{
        struct ConIOBlocks *c;

        struct MsgPort *tpw; /* porta di risposta scrittura */

        int error;

        /* allocazione memoria */
        c = (struct ConIOBlocks *)AllocMem(
            sizeof(struct ConIOBlocks), MEMF_CLEAR);

        if (c == 0)     /* non c'e' memoria sufficiente*/
           goto cleanup1;

        tpw = CreatePort(0,0);
        if (tpw == 0)
           goto cleanup2;

        c->tpr = CreatePort(0,0);
        if (c->tpr == 0)
           goto cleanup3;

        c->writeReq = CreateStdIO(tpw);
        if(c->writeReq == 0)
           goto cleanup4;

        c->readReq = CreateStdIO(c->tpr);
        if(c->readReq == 0)
           goto cleanup5;

        c->writeReq->io_Data = (APTR)window;
        c->writeReq->io_Length = sizeof(struct Window);

        error = OpenDevice("console.device",0,c->writeReq,0);
        if(error != 0)
           goto cleanup6;

        c->readReq->io_Device = c->writeReq->io_Device;
        c->readReq->io_Unit   = c->writeReq->io_Unit;


        return(c);

cleanup6:
        DeleteStdIO(c->readReq);
cleanup5:

        DeletePort(c->tpr);
cleanup4:
        DeleteStdIO(c->writeReq);
cleanup3:
        DeletePort(tpw);
cleanup2:
        FreeMem(c, sizeof(struct ConIOBlocks));
cleanup1:
        return(NULL);
}
```

Figura 3

CMD_RESET	Resetta il device ai valori di partenza
CMD_READ	Legge.
CMD_WRITE	Scrive.
CMD_UPDATE	Porta all'esterno i contenuti dei buffer interni.
CMD_CLEAR	Cancella tutto il contenuto del buffer.
CMD_STOP	Blocca il device che non puo' piu' fare nulla.
CMD_START	Riprende il normale funzionamento dopo uno stop.
CMD_FLUSH	Fa abortire tutte le richieste di i/o simulando sempre un errore.

Comandi standard per i devices

Figura 4

Spazio	08	
LineFeed	0a	
FormFeed(cancellazione console)	0c	
		Styles
Return	0d	
Shift In	0e	00 Plain
Shift Out	0f	01 Bold
<CSI>	9b	03 Italic
Insert di <N> spazi	9b <n> 40	04 Underlined
Muove in alto di <N> spazi	9b <n> 41	07 Reverse
// in basso	9b <n> 42	
// a sinistra	9b <n> 43	
// a destra	9b <n> 44	

ALCUNI CARATTERI DI CONTROLLO

Figura 5

qualcos'altro mentre nel modo sincrono il task rimarrà in stato di attesa, e quindi di inoperosità, finché non sarà esaudita la nostra richiesta.

La scelta è, ovviamente, effettuata di volta in volta dal programmatore e dalle sue esigenze, sta a voi quindi decidere in questo senso.

In realtà esiste una terza funzione, la BeginIO che non sarà trattata in questa sede in quanto presuppone una conoscenza del device molto approfondita.

Chiusa la parentesi ritorniamo ai blocchi IORequest, tramite il loro utilizzo, dicevamo, possiamo controllare la nostra console utilizzando il set di comandi per la gestione dei device riportato in figura 4.

Come potete vedere possiamo praticamente fare di tutto con questi semplici comandi, unico neo è che ogni volta sarà necessario inviare l'intero, e lungo, blocco di richiesta di I/O... un'ottima alternativa è crearci delle funzioni, da utilizzare per l'uso, sullo stile dell'ADPmttb, così che ci sia possibile solo con la chiamata, leggere, scrivere e fare tutto ciò ci sia necessario sulla nostra console.

Proviamo subito a stampare qualcosa, il solito «Hello World!» andrà benissimo, a differenza del vostro primo programmino analogo però questa volta avrete la soddisfazione di aver creato da voi stessi l'ambiente di lavoro, bel salto di qualità!

Oltre alla possibilità di stampare e leggere semplicemente, sulla nostra console ci si presentano altre interessanti peculiarità.

Grazie a particolari e prestabilite sequenze di Byte definibili come «sequenze di caratteri di controllo», che ci permettono appunto di controllare nel modo più assoluto la console, potremo fare di tutto, dal semplice spostamento del cursore alla cancellazione della window.

L'elenco di alcuni di questi caratteri di controllo e riportato in figura 5 e, come potete vedere, il loro impiego è piuttosto ampio.

Per farne uso basta semplicemente inviarli al console.device tramite il comando CMD_WRITE con una apposita IORequest.

Potrebbe essere utile, per farvi un esempio, associarli ad alcuni tasti in modo da poter usufruirne semplicemente con la pressione del corrispondente tasto.

Sempre attraverso questo particolare sistema oltre a poter controllare la console nel modo descritto potremo cambiare il colore di primo piano e dello sfondo selezionabili tra i primi otto della tavolozza in uso con l'invio dei numeri esadecimali da 30 a 37 per il testo e da 40 a 47 per lo sfondo nonché selezionare lo stile voluto mediante i codici riportati sempre in figura 5.

E non è tutto! Non è necessario inviare volta per volta i singoli caratteri di controllo, basterà inviare un'unica sequenza di byte per ottenere tutti gli effetti voluti.

Inutile dire che con i clasici OpenFont e SetFont potremo sceglierci il font che più ci aggrada senza dover far altro visto che autonomamente la console si adeguerà ricostruendosi, in parole povere, con colonne e righe adeguate alla grandezza dei font.

Porte di risposta ed eventi complessi di input

Ancora un altro paio di considerazioni.

Utilizzare una porta di risposta nella gestione dei device non è indispensabile, tuttavia potete notare nell'esempio che si fa uso di questa particolarità.

In effetti quando si lavora con i device può essere utile implementare l'utilizzo di una porta di risposta, soprattutto visto che utilizziamo una macchina come l'Amiga che fa larghissimo uso del multitasking, infatti è possibile che il nostro task, a causa del non verificarsi di alcuni eventi rimanga in stato di inattività, per cui una porta di risposta può far si che giunga un messaggio al task in uso, inoltre in alcuni casi questa porta e indispensabile affinché si possa dialogare correttamente con il device.

Tramite il console.device è possibile anche gestire eventi complessi di input, come la selezione di un menu o di un gadget, tuttavia in questi casi è preferibile l'utilizzo degli strumenti che Intuition ci mette a disposizione più del device stesso, visto che con IDCMP (Intuition Direct Communication Message Port per intenderci...) è semplicissimo controllare questo tipo di eventi.

Conclusioni

Siamo giunti alla fine di questo «viaggio» nel console.device che spero vi abbia interessato, lo spero soprattutto perché a mio parere è una caratteristica che non può essere trascurata nello studio della struttura di Amiga.

Indubbiamente non ho detto tutto ciò che potevo dire ma del resto quando si parla di Amiga è praticamente impossibile....

Prima di lanciarvi verso i vostri rispettivi «gioielli», vi consiglio se volete saperne di più, di andarvi a guardare anche il sesto capitolo dell'Amiga Rom Kernal Manual dove senza dubbio l'argomento è trattato in modo molto più approfondito.

Buono studio.

MC

COMPUTER HSP COMPUTER

COMPUTER HSP COMPUTER

COMPUTER HSP COMPUTER

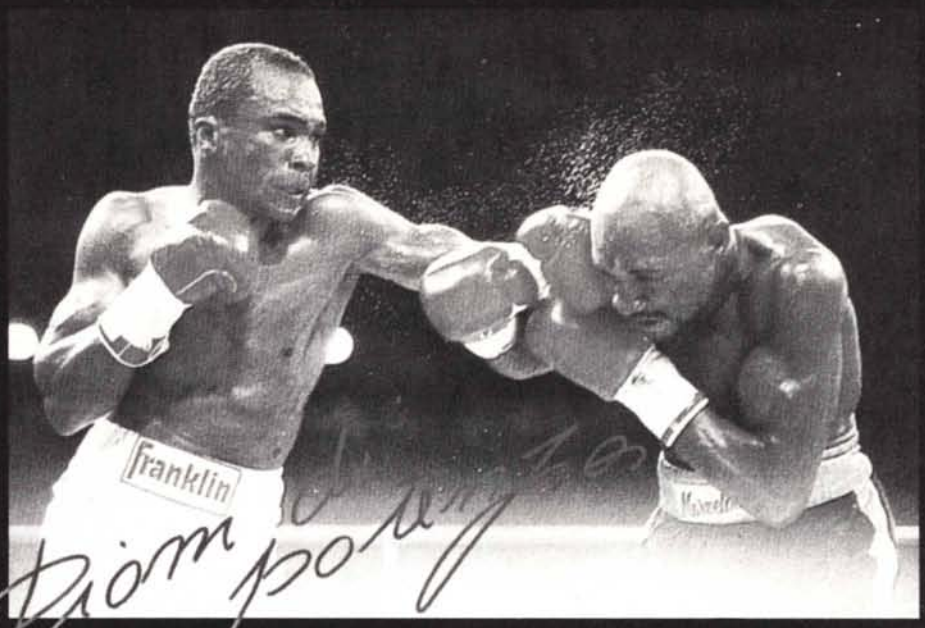
COMPUTER HSP COMPUTER

COMPUTER HSP COMPUTER

COMPUTER HSP COMPUTER

COMPUTER HSP COMPUTER

THE BIG APPLE

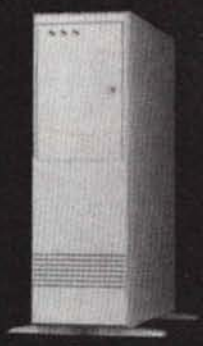


i campioni

386 33MHz
64 K CACHE

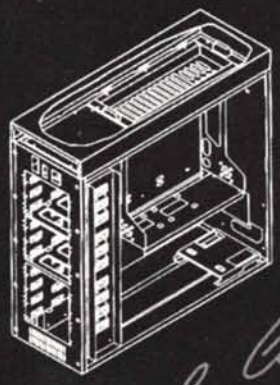
ANNO 1989

486 25 MHz



ANNO 1990

.....



ANNO 1991

la storia continua

Uff. Comm.: Via P. Fumaroli 12/A Tel. 06 – 2251517 – ROMA
Conc. Centro Italia Info.Sist.: Via Malta 8 – Tel. 06–8842378/8411987 – 00198 ROM
Centro ass. PC Service Via Malta 8 – Tel. 06 – 8411987 – 00100 ROMA