

ADPmttb 2.0

quarta parte

Quarta e, per il momento, ultima puntata dell'ADP multitasking toolbox per Amiga. Prima di passare alla descrizione dell'ultima manciata di funzioni è d'obbligo (mica tanto) una precisazione. Ho notato, con dispiacere, che qualche lettore ha fatto un po' di confusione tra ADPmttb e ADPnetwork. Qualcuno (anzi, più d'uno) s'è anche complimentato col sottoscritto per il bell'ADPnetwork Toolbox, mischiando pericolosamente le due cose. L'mttb è praticamente un linguaggio di programmazione che permette di scrivere facilmente applicazioni multitask. Ovvero invece di scrivere un unico programma «grande» che fa molte cose, si scrivono un insieme di processi che singolarmente svolgono poche funzioni, ma globalmente (e grazie al multitask di Amiga) equivalgono, ottimizzandone le prestazioni, all'unico programmine iniziale. Il software di rete di ADPnetwork è stato scritto interamente utilizzando l'mttb (come linguaggio di programmazione) ma le funzioni svolte sono ben diverse. L'mttb serve per far comunicare più processi su una stessa macchina, ADPnetwork serve per far comunicare processi in esecuzione su macchine diverse collegate in rete. È chiaro?

Non vi nascondo però che l'attuale mttb si sta già espandendo verso la rete, permettendo, in futuro, molte delle funzionalità attualmente offerte «in locale» anche «in remoto», ma è un discorso che semmai riprenderemo tra qualche mese dopo aver completato la serie di articoli su ADPnetwork.

Caratteri, puntatori e figli

I comandi presentati questo mese riguardano la spedizione inter-processi di caratteri e puntatori, la creazione e il controllo di processi figli. Come potete vedere dal listato, le due nuove coppie di send e receive fanno al loro interno comunque uso delle note SendBlock e ReceiveBlock e quindi servono solo per semplificare operazioni comunque già possibili. Infatti SendBlock e ReceiveBlock sono in grado di spedire e ricevere da un processo ad un altro qualsiasi cosa mantenuta in memoria.

Cominciamo dalla spedizione caratteri. Il processo interessato all'operazione esegue semplicemente un:

```
SendChar(mode, porta, carattere)
```

dove «mode» è come al solito MODE_SYNC o MODE_ASYNC per comunicazione sincrona o asincrona, «porta» è il nome della porta sulla quale effettuare la spedizione e «carattere» è, naturalmente, il carattere da spedire sotto forma di costante o variabile. Per fare un esempio, spedizioni «valide» di caratteri possono essere:

```
SendChar(MODE_SYNC, "yourport", 'a')
SendChar(MODE_ASYNC, "destport", v[35])
SendChar(MODE_SYNC, "yourport", p)
```

dove naturalmente «p» è una variabile di tipo char e «v» un array di caratteri.

Nel processo partner, ovvero il processo destinatario, oltre alla definizione di porta mttb:

```
NewPort("nome della porta")
```

troviamo la chiamata:

```
L = ReceiveChar(mode, porta, &var)
```

dove «mode» è il modo di ricezione MODE_WAIT o MODE_NOWAIT per receive bloccante e non bloccante, «porta» è la porta precedentemente definita (nello stesso processo!) e «&var» è il puntatore ad una variabile di tipo carattere (o ad un elemento di un array

```

.....
*
*      A D P m ' t t b   2 . 0
*
*      MultiTasking ToolBox
*      (quarta parte)
*
*      (c) 1989 ADPsoftware
*
.....
/

int  SendPointer(UBYTE,char *, VOID *);
int  SendChar(UBYTE,char *,char);
int  ReceivePointer(UBYTE,char *, VOID **);
int  ReceiveChar(UBYTE,char *,char *);
BPTR StartProcess(char *,long);
VOID RunProcess(int, char *,char *,char *,char *,char *,char *,char *,char *);
int  WaitEndProcess(BPTR);
BOOL CheckEndProcess(BPTR);
VOID AllocFree(int, char *, char *, char *,struct WBStartup *);

/.....
*
*      S E N D   P O I N T E R
*
.....
/

SendPointer(mode,porta,pointer)
UBYTE mode;
VOID *pointer;
char *porta;
|
char pp[12];
sprintf (pp,"%d", (int)pointer);
return(SendBlock(mode,porta,pp,strlen(pp)+1));
|

/.....
*
*      R E C E I V E   P O I N T E R
*
.....
/

ReceivePointer(mode,porta,pointer)
UBYTE mode;
VOID **pointer;
char *porta;
|

```

```

char pp[12];
int i;
l=ReceiveBlock(mode,porta,pp);
*pointer = (VOID *)atoi(pp);
return(l);
}

.....
*   S E N D   C H A R   *
.....

SendChar(mode,porta,car)
UBYTE mode;
char car,*porta;
{
char pp[12];
sprintf(pp,"%c",car);
return(SendBlock(mode,porta,pp,2));
}

.....
*   R E C E I V E   C H A R   *
.....

ReceiveChar(mode,porta,car)
UBYTE mode;
char *porta, *car;
{
char pp[12];
int i;
l=ReceiveBlock(mode,porta,pp);
if (l) *car = pp[0];
else *car = '\0';
return(l);
}

.....
*   S t a r t   P r o c e s s   *
.....

BPTR StartProcess(program,pri)
char *program;
long pri;
{
int i,len,slash=0,j=0;
char *porta,*msg,process[30];
struct MsgPort *rport,*wbat;
struct WBStartup *wbst;
BPTR NewSeg;

len = strlen(program);
for (i=len;i>0 && slash==0;i--)
if (program[i] == '/' || program[i] == ':') slash = i;
for (i=slash;i<=len;i++) process[j++] = program[i];

porta = (char *)AllocMem(30,MEMF_CLEAR);
msg = (char *)AllocMem(10,MEMF_CLEAR);
wbst = (struct WBStartup *)AllocMem(sizeof(struct WBStartup),MEMF_CLEAR);

if ((NewSeg = LoadSeg(program)) == 0)
{
FreeMem(porta,30);
FreeMem(msg,10);
FreeMem(wbat,sizeof(struct WBStartup));
return(OP_FAIL);
}

if ((rport = (struct MsgPort *)CreateProc(program,pri,NewSeg,4000)) == NULL)
{
FreeMem(porta,30);
FreeMem(msg,10);
FreeMem(wbat,sizeof(struct WBStartup));
UnloadSeg(NewSeg);
return(OP_FAIL);
}

sprintf(porta,"RPOf%d",NewSeg);
AllocFree(ACTION_ALLOC,porta,porta,msg,wbat);

if ((rport = (struct MsgPort *)CreatePort(porta,0))!=NULL)
{
AllocFree(ACTION_FREE,porta,NULL,NULL,NULL);
return(OP_FAIL);
}

strcpy(msg,"startup");
wbst->sm_Message.mn_ReplyPort = rport;
wbst->sm_Message.mn_Length = sizeof(struct WBStartup);
wbst->sm_Message.mn_Node.ln_Name = msg;
wbst->sm_ArgList = NULL;
wbst->sm_ToolWindow = NULL;

PutMsg(rport, (struct Message *)wbst);
return(NewSeg);
}

.....
*   R u n   P r o c e s s   *
.....

VOID RunProcess(count,p0,p1,p2,p3,p4,p5,p6,p7)
char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
{
BPTR seg0,seg1,seg2,seg3,seg4,seg5,seg6,seg7;

```

(continua a pag. 160)

di caratteri) nella quale riceveremo il carattere. Se la receive era non bloccante (MODE_NOWAIT) e il mittente non aveva ancora effettuato la spedizione, viene restituito il carattere nullo e ritornato uno 0 come risultato della ReceiveBlock (posto nel nostro caso nella variabile intera «L»).

Discorso del tutto analogo per spedire e ricevere i puntatori; le due funzioni sono SendPointer e ReceivePointer. Anche in questo caso, alla ReceivePointer sarà necessario passare l'indirizzo della variabile dichiarata come «puntatore a qualcosa». Facciamo un esempio chiarificatore. Immaginiamo che il processo mittente spedisca in modo sincrono il puntatore al suo array di char Pippo sulla porta «ics» del processo ricevente. La chiamata sarà:

```
SendPointer(MODE_SYNC, "ics", Pippo)
```

essendo Pippo effettivamente il puntatore al primo elemento dell'array. Nel processo ricevente troveremo:

```
char *Pluto;
NewPort("ics");
ReceivePointer(MODE_WAIT, "ics", &Pluto);
```

e al ritorno della ReceivePointer avremo in «Pluto» il puntatore all'array «Pippo» del mittente. Inutile ricordarvi che gli altrui puntatori vanno manipolati con molta cura e che non bisognerebbe mai modificare zone di memoria il cui puntatore è stato ceduto a qualcun altro. Come dire che è meglio lasciare perdere queste due funzioni prima di averci preso la mano coi problemi multitask.

Per quanto riguarda la creazione di processi figli, il discorso si complica un po' ma non eccessivamente. Innanzitutto sono state utilizzate all'interno delle funzioni mttb, chiamate a funzioni del Workbench. Si presuppone che i processi da lanciare come figli siano ovviamente opportunamente compilati e presenti in un qualsiasi device sottoforma di file eseguibile. Partiamo con la funzione RunProcess che permette di lanciare un numero di processi compreso tra 1 e 8. Tale funzione restituisce il controllo al processo chiamante (il padre) appena sono terminati tutti i processi lanciati (i figli). I parametri da utilizzare sono semplicemente il numero di processi e, l'uno dopo l'altro, i nomi dei processi da lanciare. I nomi dei processi devono naturalmente coincidere coi nomi dei file eseguibili, e gli stessi sono passati al sistema come nomi dei processi in esecuzione sulla macchina (path di indi-

rizzamento escluso). Se ad esempio da un processo invochiamo:

```
RunProcess(3, "pippo", "df0:pr/pluto",
«c:list»)
```

creeremo tre processi di nome «pippo», «pluto» e «list» caricati rispettivamente dalla directory corrente, dalla directory «pr» del disco posto in df0 e dal device logico «c:». L'esecuzione del processo chiamante continuerà appena i tre processi lanciati saranno tutt'e tre terminati.

Se invece siamo interessati a lanciare processi in maniera asincrona, ovvero a creare figli continuando però la nostra elaborazione, esiste la funzione StartProcess che lancia un sottoprocesso e restituisce subito il controllo al chiamante. Per fare questo dobbiamo farci carico di una piccola incombenza, ovvero tenere traccia del puntatore al segmento di memoria utilizzato per caricare il processo da eseguire. Tale puntatore, restituito appunto dalla StartProcess, sarà utilizzato sempre da noi per testare (eventualmente) lo stato del processo lanciato e/o per attendere la sua terminazione (deallocando così il segmento utilizzato). Oltre a questo la StartProcess permette di lanciare processi a priorità variabile e non a priorità zero come fa di default la RunProcess. Proviamo a lanciare i tre di sopra in maniera asincrona:

```
BPTR seg1, seg2, seg3;
```

è la dichiarazione dei tre puntatori BCPL che utilizzeremo. Segue:

```
seg1 = StartProcess("pippo", 0);
seg2 = StartProcess("df0:pr/pluto", 0);
seg3 = StartProcess("c:list", 0);
```

a questo punto l'elaborazione del processo chiamante continua e possiamo ad esempio chiederci se un determinato processo è terminato oppure no. Esiste la funzione booleana CheckEndProcess alla quale passiamo il BPTR corrispondente al processo da testare:

```
if [CheckEndProcess(seg1)]
printf ("processo terminato");
```

stamperà la stringa indicata se il processo «pippo» (associato a seg1, ricorda-te?) è terminato. Analogamente possiamo attendere la fine di un processo da noi creato, con la funzione WaitEndProcess passando nuovamente il BPTR iniziale. In questo caso si torna dalla funzione solo quando il processo relativo al BPTR passato come parametro è effettivamente terminato. **MC**

(segue da pag. 159)

```
switch (count)
{
case 8: seg7 = StartProcess(p7,0);
case 7: seg6 = StartProcess(p6,0);
case 6: seg5 = StartProcess(p5,0);
case 5: seg4 = StartProcess(p4,0);
case 4: seg3 = StartProcess(p3,0);
case 3: seg2 = StartProcess(p2,0);
case 2: seg1 = StartProcess(p1,0);
case 1: seg0 = StartProcess(p0,0);
}

switch (count)
{
case 8: WaitEndProcess(seg7);
case 7: WaitEndProcess(seg6);
case 6: WaitEndProcess(seg5);
case 5: WaitEndProcess(seg4);
case 4: WaitEndProcess(seg3);
case 3: WaitEndProcess(seg2);
case 2: WaitEndProcess(seg1);
case 1: WaitEndProcess(seg0);
}

/*****
 * WaitEndProcess
 *****/

WaitEndProcess(segment)
{
BPTR segment;
{
char p[30];
struct MsgPort *rp;

sprintf(p,"RPOf%d",segment);
if ((rp = (struct MsgPort *)FindPort(p))==NULL) return(NO_PORT);

WaitPort(rp);
GetMsg(rp);
DeletePort(rp);
UnloadSeg(segment);
AllocFree(ACTION_FREE,p,NULL,NULL);
}

/*****
 * CheckEndProcess
 *****/

BOOL CheckEndProcess(segment)
BPTR segment;
{
char p[30];
struct MsgPort *rp;
BOOL bool;

sprintf(p,"RPOf%d",segment);
Forbid();
rp = (struct MsgPort *)FindPort(p);
bool = (BOOL)(rp->mp_MsgList.lh_Head->ln_Succ!=NULL);
Permit();
return(bool);
}

/*****
 * Routine utilizzata da StartProcess & WaitEndProcess
 *****/

VOID AllocFree(Action, PortName, memport, memmag, memwbst)
int Action;
char *PortName, *memport, *memmag;
struct WBStartup *memwbst;
{
struct mem_alloc
{
char id[20];
char *porta;
char *mag;
struct WBStartup *wbst;
};

static struct mem_alloc ToFree[50];
static int count = 0;
int i=0,j;
if (Action == ACTION_ALLOC)
{
strcpy(ToFree[count].id, PortName);
ToFree[count].porta = memport;
ToFree[count].mag = memmag;
ToFree[count].wbst = memwbst;
}
if (Action == ACTION_FREE)
{
while (strcmp(ToFree[i].id, PortName) != 0) i++;
FreeMem(ToFree[i].porta,30);
FreeMem(ToFree[i].mag,10);
FreeMem((char*)ToFree[i].wbst,sizeof(struct WBStartup));
count--;
for (j=i;j<count;j++)
{
strcpy(ToFree[j].id,ToFree[j+1].id);
ToFree[j].porta = ToFree[j+1].porta;
ToFree[j].mag = ToFree[j+1].mag;
ToFree[j].wbst = ToFree[j+1].wbst;
}
}
}
}
```


La nuova cartuccia per C64

Mk 6°

Mk 6, manuale in italiano, garanzia 5 anni 99.000
 Cavo Centronics per Mk 6 39.000
 Enhancement Disk - utilities e parametri speciali 19.000
 Graphic Disk, nuovo disco di utility per Mk 6 con SlideShow di immagini, Sprite Editor Deluxe, Message Maker ad altro ancora 19.000

STAMPANTI

Star LC-10 II, nuova 180 cps, multifont 490.000
 Star LC-10 II, versione a colori 590.000
 Nec P-2200, 24 aghi, 165 cps, moltissimi font residenti per una superba qualità di stampa 899.000
 Nec P6 Plus, 24 aghi, 265 cps, pilotata da microprocessore 8088, trattore a spinta, estremamente sofisticata 1.699.000
 Xerox 4020, stampante a colori a getto d'inchiostro, risoluzione 240 dpi (qualità fotografica), silenziosissima .. 3.490.000

LBP-4, stampante Canon laser HP compatibile, 300 dpi 2.850.000
 OmniLaser 2106, stampante Laser Texas Instruments, adotta il linguaggio PostScript, emulazione HP, Diablo 630, plotter HP7475, 300 dpi, 6 pagg./min. 6.990.000

INCREDIBILE!! Fax MacRom 88, 16 toni di grigio solo 999.500

AT 12 MHz, 1 MB Ram, alimentatore 200 W, Bios Award, controller 16 bit floppy disk e hard disk, 1 disk drive a scelta da 1.2 MB o 1.44 MB, scheda video Hercules+Cga, tastiera 101 tasti **1.390.000**

NASTRI DI RICAMBIO PER TUTTE LE STAMPANTI

HARD DISK

Hard card 21 MB 649.000
 Hard card 33 MB 749.000
 Hard card 44 MB 949.000
 Seagate 20 MB 40 ms 550.000
 Seagate 40 MB 23 ms 850.000
 Amigos 20 MB per A-500 850.000

Dischi Fish di pubblico dominio aggiornati al n. 240

ESPANSIONI DI MEMORIA

Per A-500 da 512 K 239.000
 Per A-590 da 2 MB 649.000
 Per A-500 da 1.8 MB int. 999.000
 Per A-2000 da 2 MB 899.000
 Per Amiga 1000 da 390.000
 Per XT a 640 K da 245.000
 Per AT da 2 MB da 990.000

386-SX 16 MHz, 1 MB Ram espandibili a 8 MB su piastra (standard EMS 4.0), alimentatore 200 W, Bios Award, controller 16 bit floppy disk e hard disk, 1 disk drive a scelta da 1.2 MB o 1.44 MB, scheda video Hercules+Cga, seriale, parallela, tastiera 101 tasti **IN OFFERTA 1.990.000**

Assistenza, consulenza e riparazioni per tutti i computer

Amiga 500
512 K Ram
799.000

Amiga 2000
1 MB Chip Ram,
1 disk drive 3"1/2
1.799.000

XT 640 K
espandibile a
1 MB, 12 MHz,
1 disk drive da
360 K o 720 K,
scheda video
Hercules+Cga,
tastiera 101 tasti
779.000

Philips TC-100 899.000

Amiga 500
Ultimo modello con
1 MB Chip Ram
sulla piastra madre
999.000

Amiga 2000
1 MB Chip Ram,
2 disk drive 3"1/2
1.999.000

Philips NMS 9111 1.349.000
8 MHz, 768 K Ram, 2 disk drive da 360 K o 720 K a scelta, schede Hercules+Cga, seriale, parallela; tastiera, MS-Dos 3.21

Philips NMS 9115 1.799.000
8 MHz, 768 K Ram, hard disk 20 MB, 1 disk drive da 360 K o 720 K a scelta, schede Hercules+Cga, seriale, parallela; tastiera, MS-Dos 3.21

Prezzi IVA compresa

Atari ST-1040
con monitor Atari
mono SM-124
1.299.000

Commodore 64 + registratore dedicato + joystick 349.000

Viale Monte Nero 31
20135 Milano

Tel. (02) 55.18.04.84

(4 linee ric. aut.)

Fax (02) 55.18.81.05 (24 ore)

Negoziato aperto al pubblico tutti i giorni dalle 10 alle 13 e dalle 15 alle 19.

Vendita per corrispondenza.

Sconti per quantità ai sigg. Rivenditori.

Atari ST-1040
con monitor Atari
SM-1224 colori
1.599.000

Commodore 64 + disk drive OC-118 + joystick 499.000

