Mark Smiddy

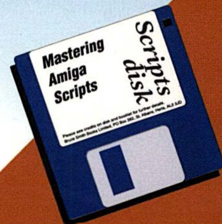
Mastering Amiga Scripts

Over 100 DOS Scripts

Covers Versions 1.x, 2.x and 3.x

FREE Scripts Disk Offer

BSB
Bruce Smith Books



Mastering AmigaDOS Scripts

Mark Smiddy

Mastering AmigaDOS Scripts

© Mark Smiddy 1994
ISBN: 1-873308-36-1 First Edition: December 1994.

Editors: Mark Webb
Typesetting: Bruce Smith Books Ltd

Workbench, Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc. UNIX is a trademark of AT&T. MS-DOS is a trademark of Microsoft Corporation. Designer Mouseware is a trademark of Mark Smiddy. All other Trademarks and Registered Trademarks used are hereby acknowledged.

All rights reserved. No part of this publication may be reproduced or translated in any form, by any means, mechanical, electronic or otherwise, without the prior written consent of the copyright holder(s).

Disclaimer: While every effort has been made to ensure that the information in this publication (and any programs and software) is correct and accurate, the Publisher can accept no liability for any consequential loss or damage, however caused, arising as a result of using the information printed in this book.

E. & O.E.

The rights of Mark Smiddy to be identified as the Author of the Work has been asserted by him in accordance with the *Copyright, Designs and Patents Act 1988.*

Bruce Smith Books is an imprint of Bruce Smith Books Limited.

Published by:

Bruce Smith Books Limited. PO Box 382, St. Albans, Herts, AL2 3JD.
Telephone: (01923) 894355, Fax: (01923) 894366

Registered in England No. 2695164.

Registered Office: Worplesdon Chase, Worplesdon, Guildford, Surrey, GU1 3UA.

Printed and bound in the UK by Ashford Colour Press, Gosport.

"Experts are not born.
They are hewn from the bedrock of endeavour,
and the granite of experience."

The Author

MARK SMIDDY is a founding Consulant Editor on Future Publishing's acclaimed *Amiga Shopper* and the world's best known AmigaDOS author. He has worked with and written about, a variety of computers over the last 12 years and thinks intellectual bores should be lined up against a wall and shot: not necessarily in that order. Mark currently lives in a sleepy Cleveland backwater and remains convinced that life is a fatal disease.

The Scripts Disk

Mastering AmigaDOS Scripts – The Disk contains a large number of programs. If you have an aversion to typing these in – or find that you can't get them to run correctly – then you may be interested to know that they are available on a companion disk we have compiled to go along with this book. In addition, the MAD3 Scripts Disk also contains a good selection of interesting and useful PD/Shareware software. The disk may also contain information that has come to light since this book was published.

The disk is available at a nominal charge of £2.00 to cover the cost of P&P and to obtain it simply fill in and return the tear-out form you will find towards the end of this book.

When you get your disk you will find that it contains a ReadMe file when the disk window is opened. Simply double-click on this for a full description of the files on the disk.

Contents

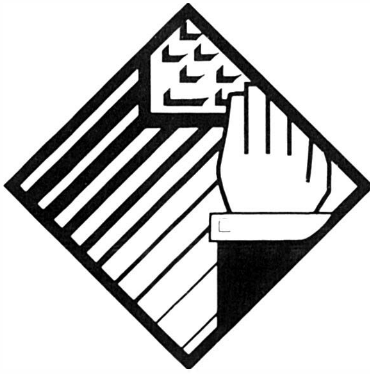
Introduction 9
Alarm Clock: Alarmset 11
AlarmSnooze 13
Alarm Clock WB Alarmset 15
AlarmClock 17
AskEm 20
AddData 79
AutoHelp 22
AutoStart 1.3 23
AutoStart 2 25
Back (Alias) 26
Barclock 27
BarGraph 30
Booty 40
CALC 1.3 43
Calendar 49
Monthprint 2 57
Monthprint 45
Calendar 2 62
CCOPY (Alias) 66
CCOPY 64
Chatter 67
CHATTY 70
Clock 1.3 71
Clock 2 73
Clock 3 75
Clock 77
Delf (Alias) 86
Del (Alias) 87
DiskDoc (Script/Alias) 111
Doctor (Alias 88
DRS (Alias) 89
DVS (Alias) 90
DelBlock 81

Database 94
DCopy 106
DEL 108
EDS (Alias) 115
EDU (Alias) 116
ENABLE 124
EX (Alias) 125
FFIND (Alias) 135
FindData 91
FRED (Alias) 136
FREUD (Alias) 137
FTEXT 138
EggTimer 117
EMove 122
FACTOR 126
FancyList 130
FCD 131
GetEm 139
GetEm 2 141
Halt 143
Host-Chat 145
HostRead 147
InterDel 149
IntelliRes 152
ListALL 158
List D 161
ListDel 163
LD (Alias) 147
Mail-2-Host 167
Mail-2-Remote 168
MD (Alias) 164
MID 174
MID1 194
MRun 196
MemBar 175
MemFreeP 178

MemG ... 185
MemInk ... 187
Mem6 ... 188
MD ... 164
NOT ... 199
PathFind ... 203
Pest (AmigaDos 1.3) ... 205
Pest (AmigaDos 2) ... 209
Pest2 (AmigaDos 1.3) ... 212
Pest3 (AmigaDos2) ... 215
Pest3 (AmigaDos3) ... 216
Pest: AddPestEvent ... 220
Pest3 ChangPestMessage ... 232
Pest3: DeletePestEvent ... 235
Pest3: GetArgs ... 238
Pest3: KillPestEvent ... 240
Pest3: ListPestEvent ... 244
Pest3: SetPestEvent ... 247
Pest3: SetWaitEvent ... 253
Pest3: StartPest ... 256
PrintData ... 99
PFind (Alias) ... 258
QFF (Alias) ... 259
QF (Alias) ... 260
RCD2 ... 261
RCD ... 253
RecDemo ... 268
RemAlias ... 270
Remt-Chat ... 273
RemoteRead ... 274
REN ... 275
ResCalc ... 276
SAFE ... 281
SlideshowWB ... 282
Slideshow ... 284
STOP ... 286

SortData....101
SubDemo....287
SX....289
TD (Alias)....292
TreeStart....293
Tree....294
UNSAFE (Alias)....299
ViewBlock....103
VLS (Alias)....300
VOLS....301
WD (Alias)....302
WHO (Alias....303
WX....304
X (Alias)....306
XCD....307

Appendix
A: Books for your Amiga....311



Introduction

"Alas poor Yorick, I knew him..."

Poor Shakespeare would turn in his grave if he knew how many times people have misquoted that famous line from Hamlet. The problem is our brains hear the first part of the sentence and fill in the rest – in modern English we wouldn't think of ending such a sentence with "Horatio" (the listener). It just doesn't sound right. That's the thing with scripts: they're a sequence of instructions written down for actors to follow, and a director to alter at will.

Amiga scripts are like that. No more than a sequence of AmigaDOS commands which define some action or actions. When starting out with this fascinating area, you are the copyist entering the script for the actors to follow. Later, you will direct the course of events by changing the scripts to suit your needs. Eventually you will be the scriptwriter creating scripts to solve your own, distinct problems.

Looked at from another angle, scripts are AmigaDOS programs. No different from the programs that Charles Babbage first conceived of when he first thought up the idea of his Analytical Engine. Babbage's idea was to create a machine that could follow a set of pre-written instructions that could be changed to fit the job at hand. Of course, in those days, programs only solved mathematical problems. Even Babbage, were he alive today, might find it difficult to conceive of how much effect his idea could have had on our everyday lives.

All this talk of computers and computer programs may have you thinking, "Is all this for me?".

Programming is an art and a science all rolled into one. The scientific part is being able to think of a problem logically and break it into easily achievable steps (commands, if you like) whereas the artistic part is the ability to add flare and polish to a program.

Nothing in this book is beyond you. Even the fact you have managed to read the words here proves you are intelligent enough to learn a language: one of the most complex in the world come to that. To get the most from this book, you should have a basic understanding of the workings of AmigaDOS, perhaps by following on from Mastering AmigaDOS Tutorial; or The AmigaDOS Insider Guide. In any case a copy of Mastering AmigaDOS 3 Reference would be very helpful.

Just remember as I have said many times, "Experts are not born. They are hewn from the bedrock of endeavour and the granite of experience". Here then are the three stages to becoming an expert.

1. Your can enter the programs as they appear at first and use them as described in the text. Copying is the first stage of learning: very few things we do are instinctive; almost everything is learnt by copying other people.
2. You follow the flow of the program as described in the detailed text and modify it to see what happens. Later on you will be able to predict what effect your actions have.
3. Having had some experience with the first two stages, you will begin to see places in your everyday use of the Amiga where a script can streamline work: and where nothing supplied here can fit the bill. You will become the scriptwriter.

AlarmClock: AlarmSet

Synopsis: [EXECUTE] <[Time=]time> [[Message=]"Text"]

Template: Time/a,message/f

Path: S:

Requires: V3+

See also: AlarmClock, AlarmSnooze

Type: Script

Brief: Alarm setting module for the Alarmclock

Description

This module is used to set the AlarmClock's Alarm from AmigaDOS. The script has a built-in message (that you can change) but it is more normal to supply one. The choice is yours. Typically, you'll use AlarmSet like this:

```
1>ALARMSET 15:00
1>ALARMSET 13:22 It's time for a cuppa!
```

Line-By-Line

1-3. Defines a simple header. Note the use of a "final" argument to ensure the whole message is collected by the command line.

4. Sets the default message. You can change this to suit yourself.

5. Creates the message variable. Multiple calls to this command will change the message: be wary of this.

6. Clears the Alarm variable.

7. Starts a new resident process which will continue until the required time is reached. Note input and output re-direction to NIL:. This ensures you can close the Shell that called AlarmSet.

8. When the time is reached and the WAIT times-out, this activates the alarm.

```
1.  .key time/a,message/f
2.  .bra {
3.  .ket }
4.  .def message "You rang, sir?"
5.  setenv AlarmMsg "{message}"
6.  setenv AlarmOn "ON"
```

```
7.  setenv Alarm ""
8.  run <NIL: >NIL: wait until {time}
9.  setenv Alarm "NOW"
```

AlarmSnooze

Synopsis: Internal to AlarmClock

Template: ...

Path: S:

Requires: V3+

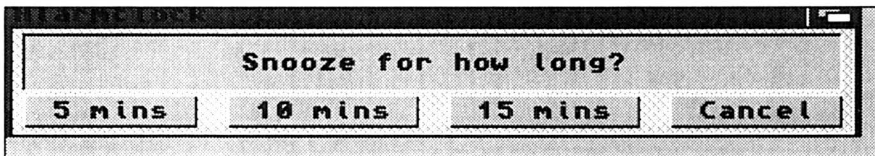
See also: AlarmClock, AlarmSet

Type: Script

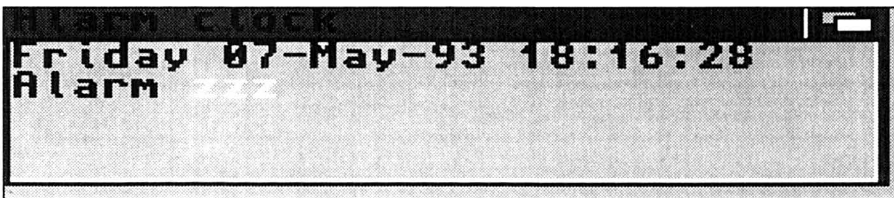
Brief: Snooze timer module for the alarm clock

Description

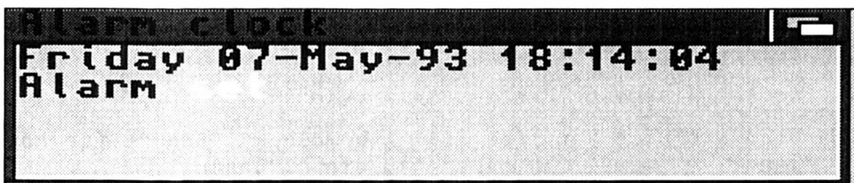
This module uses some clever features of AmigaDOS 2 and 3 to present a requester and calculate a snooze period for the AlarmClock. It is not normally executed on its own.



Snooze Request



Alarm Clock Snoozing



Alarm Clock

Line-By-Line

1-3: Construct a standard header.

4. Clears the global alarm variable.

5. Sets the Alarm message variable to itself plus "Snooze". This is to indicate that the clock has already been snoozed from its original time. On successive runs, the word Snooze is added again and again, so you can see how many times you have snoozed!

6. Displays the requester inquiring how long the user wants to snooze for. Four returns are possible here:

   1: 5 minutes.

   2: 10 minutes.

   3: 15 minutes.

   4. Cancel

7. This line does several things in one go.

   • Calculates the actual delay time required by multiplying the return from Step 6 by 3. This is done by expanding the return variable inside an inserted EVAL command. The result from: `eval $ret{$$} *5` is inserted at that point. For a return of 3, AmigaDOS sees the line as:

   ```
   run >nil: Wait 15 mins +
   ```

   • Starts a new process that will wait for the specified time.

   • Does nothing until the next line has been added to the process.

8. Is part of the process started by Step 7. When the WAIT times out, the Alarm variable is set to NOW and the Alarmclock triggers.

Listing

```
1.  .key dummy
2.  .bra {
3.  .ket }
4.  setenv Alarm ""
5.  setenv AlarmMsg "Snooze: $AlarmMsg"
6.  requestchoice >env:ret{$$} "Alarmclock" "Snooze for how
    long?" "5 mins" "10 mins" "15 mins" "Cancel"
7.  run >nil: Wait `eval $ret{$$} *5` mins +
8.  setenv Alarm "NOW"
```

AlarmClock: WBAlarmSet

Synopsis: Only run from Workbench

Template: Time,message/f

Path: na

Requires: V3+

See also: AlarmClock, AlarmSnooze

Type: Script

Brief: Alarm setting module for the Alarmclock

Description

This module is used to set the AlarmClock's Alarm from Workbench – it's really for weanies who can't be bothered using the AmigaDOS one which is a lot faster in the long run...

Line-By-Line

1-3. Defines a simple header. Note the use of a "final" argument to ensure the whole message is collected by the command line. Also, note that the "time" argument is not required by the Workbench version of this script.

4. Sets the default message. You can change this to suit yourself.

5. Checks if a time has been entered. If not, control continues at Step 6; otherwise it jumps to Step 9.

6. Displays the prompt if a time has not being supplied- as will usually be the case from Workbench.

7. The script now calls itself recursively with interactive mode triggered. Note that the output is re-directed to NIL: to prevent the command line options from being shown. If you change the name of the script you must also change its name here too.

8. When the script unwinds its recursion this jumps to the bail-out for speed.

9. Terminates the IF...ENDIF construct from Step 5.

10. Creates the message variable. Multiple calls to this command will change the message- be wary of this.

11. Sets a variable to indicate the alarm is active.

12. Clears the Alarm variable.

13. Starts a new resident process which will continue until the required time is reached. Note input and output re-direction

to NIL:. This ensures you can close the Shell that called AlarmSet.

14. When the time is reached and the WAIT times-out, this activates the alarm.
15. Is the bail-out point for the recursive calls.
16. Is some information for WX to use if the script is called from AmigaDOS.

Listing

```
1.  .key time,message/f
2.  .bra {
3.  .ket }
4.  .def message "You rang, sir?"
5.  if "{time}" EQ ""
6.   echo "Enter a time (and optional message)"
7.   execute >NIL: WBAlarmSet ?
8.   skip out
9.  endif
10. setenv AlarmMsg "{message}"
11. setenv AlarmOn "ON"
12. setenv Alarm ""
13. run <NIL: >NIL: wait until {time} +
14. setenv alarm "NOW"
15. lab out
16. ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory
    Gauge/SMART/NOSIZE
```

AlarmClock

Synopsis: Run from Workbench

Template: ...

Path: na

Requires: V3+

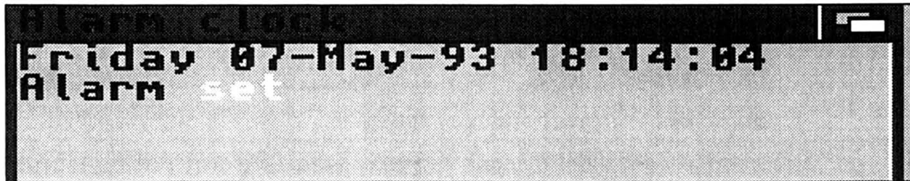
See also: Snooze, AlarmSet, Clock 2, Clock 3

Type: Script

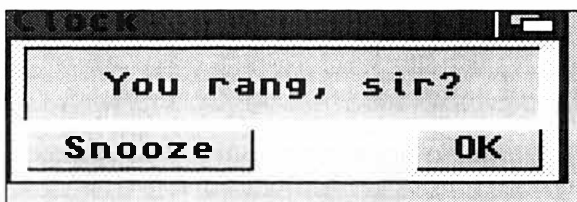
Brief: Digital alarm clock with snooze facility!

Description

This is a script to amaze your friends with. In the current incarnation it requires Workbench 3 (an A1200, for instance) because it makes use of requesters to provide instant responses. It's a bit like Pest3 in some respects, but unlike Pest it's a lot simpler and only supports a single Alarm time. A snooze mode is provided with a programmable Snooze period from 5...15 minutes. A status indicator shows if the Alarm is clear, set or asleep.



Alarm Clock Snooziing



Clock Alarmed

Line-By-Line

1-2. Makes some essential commands resident.

3-5. Checks for the global variable "AlarmON". AlarmClock needs this (even if it contains nothing) to work. If the variable does not exist it is created.

6. Switches the cursor off and positions the print position at the top, left hand corner of the window.
7. Marks the start of a loop.
8. Displays the day, date and time.
9. Checks if AlarmON contains anything. If it does, control continues at Step 10; otherwise it jumps to Step 23.
10. Tests if ClockRtn (set later in the script) was 1: Snooze mode. If it was, control continues at Step 11; otherwise it jumps to Step 12.
11. Displays the snooze "Zzz" message in a highlight colour and re-positions the cursor.
12. If control gets here from Step 11, it jumps to Step 14; otherwise it continues...
13. ...here where it displays the alarm Set message in a highlight colour.
14. Terminates the IF...ELSE...ENDIF construct opened at Step 10.
15. Tests if the alarm has "timed out" indicated by the variable. Alarm. If Alarm is set to trigger, control continues at Step 15; otherwise it jumps to Step 22.
16. Displays a requester with the alarm message (determined by "AlarmSet"). Two possible returns are possible:

```
   0:  OK. Cancel the alarm
s  1:  Snooze. Trigger snooze mode.
```

17. If the return from the alarm request was "1", control continues at Step 18; otherwise it branches to Step 19.
18. Silently creates a new process running the Snooze timer. Actually, AlarmSnooze will create its own sub-process, as you'll see described there.
19. If control reaches here from Step 18, it jumps to Step 21; otherwise it continues...
20. ...here and re-sets the alarm variable.
21. Closes the IF...ELSE...ENDIF construct opened at Step 17.
22. Closes the IF...ELSE...ENDIF construct opened at Step 15.
23. If control reaches here from Step 22, it continues at Step 25, otherwise it carries on...
24. ...here, and prints the "Alarm off" message before re-positioning the cursor ready to display the time.
22. Closes the IF...ELSE...ENDIF construct opened at Step 9.
26. Halts the script for about a second. You can increase this period if you want to give more time over to other processes.

27. Re-starts the loop and keeps the clock ticking.
28. Is some information for the WX script.

```
1.  resident c:wait
2.  resident c:date
3.  if not exists ENV:AlarmON
4.   setenv AlarmON ""
5.  endif
6.  echo "*e[0 p*e[0;0H" noline
7.  lab start
8.  date
9.  if "$AlarmOn" NOT EQ ""
10.  if $ClockRtn EQ "1"
11.  echo "Alarm *e[32mzzz*e[31m*e[0;0H" noline
12.  else
13.  echo "Alarm *e[32mset*e[31m*e[0;0H" noline
14.  endif
15.  if "$alarm" EQ "NOW"
16.   Requestchoice >env:ClockRtn "Clock" "$AlarmMsg"
    "Snooze" "OK"
17.   if $ClockRtn EQ "1"
18.    run >NIL: execute s:AlarmSnooze
19.   else
20.    setenv alarmON ""
21.   endif
22.  endif
23. else
24.  echo "Alarm off*e[0;0H" noline
25. endif
26. wait 1 secs
27. skip start back
28. ;WX:WINDOW=WINDOW=con:0/0/240/30/CLI_Clock
```

AskEm

Synopsis: EXECUTE >NIL: Askem <[file=]Answerfile> ?

Template: file/a,a,b,c,d,e,f,g,h,i,j

Path: S:

Requires: V1.2+

See also: Pest 3: GetArg

Type: Script

Brief: To interactively read input from the user in a script

Description

A programmer once commented: "ASK is fine for simple questions but what happens if I need to get a text string inside a program? There's no way to interactively ask the user a question like '*what's your name?*' once the script has started. From the outset this looks a deceptively simple command. However, this is not a stand alone program – it's a script designed to be executed from another script – possibly called by ICONX.

Line by line

1. This line is the command's key – which gives some hint as to how this script works. Apart from the file argument the rest of the options look meaningless. In fact, the less meaningful these are, the better! They never actually appear on screen and serve to pick up the user's input. I've provided 10 here which should suffice for questions such as "What is your name" and so on.

2-3. Set bra and ket to { and }.

4. This merely echoes the user's input back to the file defined in line 1.

This script is not obvious until you see it in use. So here's a very simple script to show how it works:

```
1    .key dummy
     .bra {
     .ket }
2    echo "What is your name?" noline
3    execute >nil: AskEm ram:Answer{SS} ?
     echo "Nice to 'eat you' " noline
     type ram:Answer{SS}
```

Line by line

1. This is a dummy parameter.
2. ECHO is being used here to ask the question. This has to be done here for reasons which will become clear below.
3. This is the crucial part. AskEm is executed with output redirection sunk to NIL:. This makes sure that it can't generate any output of its own to the screen. The ? at the end puts the script's key into interactive mode so it is ready to accept some input. Remember, nothing actually appears because the output is going to NIL: although, what you type belongs to the current console so it does appear.

   More important, AskEm has one required argument – the file it will send its output to. A little known feature of interactive mode is that you can send partial command lines – in this case the filename – before the interactive mode starts. When it does, the user's input is passed to each argument letter in turn, thus allowing about ten words for this example.

Listing

```
1.  .key file/a,a,b,c,d,e,f,g,h,i,j
2.  .bra {
3.  .ket }
4.  echo >{file} "{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}"
```

AutoHelp

Synopsis: [EXECUTE] AutoHelp

Template: none

Path: S:

Requires: V1.3+

See also: ...

Type: Script

Brief: Make all disk-loaded commands produce help templates

Description

Here's a little script for beginners and experts alike who cannot remember how each command behaves. It uses LIST to create a special alias for all commands so they always present their command line templates:

The first line creates a script file in RAM: called "helpme" formatted like this for every file in the C: assignment:

```
;<Path>
ALIAS <command> <path and command> ?
```

For instance if C: contained just CD and DIR, "helpme" would look like this:

```
;Workbench1.3:C
ALIAS DIR Workbench1.3:C/DIR ?
;Workbench1.3:C
ALIAS CD Workbench1.3:C/CD ?
```

although, in real terms, the list will be much longer – two lines for every command in C:! When this file is executed, it is no longer necessary (or possible) to supply an argument to each command. Instead you just give the command without parameters and it presents the list of parameters it requires:

```
1>DIR
NAME,OPT/K,ALL/S,DIRS/S,INTER/S,FILES/S:
```

All you have to do is enter the parameters as usual and press the <Return> key to activate the command. This is useful if you only have a single disk drive because transient commands are pre-loaded so you can swap disks without having the hassle of getting the wrong directory etc.

Listing

```
1.  LIST >RAM:HELPME C:#? LFORMAT ";%S*nALIAS %S %S%S ?"
2.  EXECUTE RAM:HELPME
```

AutoStart 1.3

Synopsis: none

Template: none

Path: S:

Requires: V1.3 - 1.3.3

See also: AutoStart 2

Type: Script

Brief: Auto start multiple application (like WB20+'s WBStartup)

Description

The most logical way to create a boot disk is to custom build a disk that will automatically start applications from a special Workbench drawer. This example applies to all releases of Workbench from 1.3 to 1.3.3 and provides functionality similar to that in Workbench 2.

Installing AutoStart

1. Boot your Workbench disk, make a copy of the Empty drawer and rename it Auto.
2. Open a Shell and enter:

```
1>ED S:Startup-sequence
```

3. Move the cursor to the line where EndCLI >NIL: appears, press <Return> to open a blank line and move the cursor into it. Now enter the lines shown in the listing below.
4. Now drag one or more applications (tools) to the Auto drawer and reboot the machine. Typical examples are Clock and NotePad (on 1.3). This patch only works on "tools". If you are unsure what an icon is, select it and choose Info from the menu. The icon's type must be a tool otherwise it will not work. (The Shell's icon for instance is a Project.)

Line-By-Line

1. The first line checks for the Auto drawer required by the patch. If the drawer is missing execution passes to step 6 and allows the startup to continue as normal. This allows you to modify one Startup-sequence and copy it to lots of different disks without having to create an Auto drawer on every one.
2. This creates a script (T:AutoTemp) using LIST's LFORMAT argument. Typically it will look something like this if the Clock and NotePad tools were placed in the Auto drawer:

```
RUN <NIL: >NIL: .info
RUN <NIL: >NIL: Clock.info
RUN <NIL: >NIL: NotePad.info
RUN <NIL: >NIL: Clock
RUN <NIL: >NIL: NotePad
```

3. Creates a macro (T:Strip) for the EDIT command. There isn't room here to describe EDIT in detail, but this macro will force EDIT to search for and delete, any lines containing the string ".info". (See Mastering AmigaDos 3 Reference).
4. Creates the final script (T:RunIt) by removing any lines containing the substring ".info". The new script will typically look something like this:

```
RUN <NIL: >NIL: Clock
RUN <NIL: >NIL: NotePad
```

As you can see, this script only attempts to RUN tools. The original MakeAuto program tried to run everything, icons and all and this slowed things down. Redirection to and from NIL: (<NIL: >NIL:) is used to stop any tools getting a "lock" on the CLI window, thus allowing it to close.

5. Executes the script. The reason for using RUN might not be clear, but in some releases EXECUTE complains about the lack of a .KEY statement. This fixes that problem at the expense of an extra [CLI 2] message during startup.
6. This is just the tag for the IF statement at step.

Listing

```
1.  IF exists SYS:Auto
2.  LIST >T:AutoTemp SYS:Auto LFORMAT "RUN <NIL: >NIL: %S%S"
3.  ECHO >T:Strip "O(F/.info/;d;)"
4.  EDIT T:AutoTemp TO T:RunIt WITH T:Strip
5.  RUN EXECUTE T:RunIt
6.  EndIF
```

AutoStart 2

Synopsis: none

Template: none

Path: S:

Requires: V2+

See also: AutoStart 1.3

Type: Script

Brief: Auto start multiple application (like WBStartup)

Description

The most logical way to create a boot disk is to custom build a disk that will automatically start applications from a special Workbench drawer.

This example applies to all releases of Workbench from 1.3 onwards. Although the Workbench already has an auto start drawer (WBStartup) it does not work correctly for applications that don't exit quickly. This makes it unsuitable for things like EMacs and IconEd for instance.

- It's worth noting, that Workbench does have a ToolType: DONOTWAIT that accomplishes this task instantly, but you have to add this manually using Icons...Information.

1. Boot your Workbench disk, and use the New Drawer function to create a drawer called Auto.
2. Append the listing to the end of User-Startup.
3. Now drag one or more applications (tools) to the Auto drawer and reboot the machine. Typical examples are Clock and NotePad (on 1.3). This patch only works on "tools". If you are unsure what an icon is, select it and choose Info from the menu. The icon's type must be a tool otherwise it will not work. (The Shell's icon for instance is a Project.)

Line-By-Line

It works just like AutoStart 1.3, with the exception that the script is created without the .info files. This is afforded by the new "~" (NOT) wildcard modifier which stops the dot-info files being included here.

Listing

```
1.  IF exists SYS:Auto
2.  LIST >T:AutoTemp SYS:Auto/~(#?.info) LFORMAT "RUN <NIL:
    >NIL: %S%S"
3.  RUN EXECUTE T:AutoTemp
4.  EndIF
```

BACK

Synopsis: BACK <command>

Requires: V1.3+

See also: ...

Type: Alias

Brief: Run a command in the background

Definition: ALIAS BACK RUN <NIL: >NIL:

Description:

This little tip is for AmigaDOS/ARP versions 1.3 and above. Partly because it uses ALIAS and partly because the NIL: device did not work correctly in earlier versions. The idea for this one came from Charlie (ARP) Heath's PD utility, RUNBACK; a patch that allows processes to run completely in the background. RUNBACK is not required for AmigaDOS 1.3 and above because the facility is already there. You use BACK like this:

```
BACK [command] [options]
```

For instance, to start the PD file viewer ZAP:

```
1>BACK ZAP
```

Beginners are probably wondering what all the fuss is about. Indeed, if you try to BACK DIR or something similar, nothing seems to happen. BACK was devised so you can start programs from the Shell then close it down. If you try this, many programs will prevent the Shell window from closing until they exit. Of course, BACK is useless for most AmigaDOS commands, it is only intended for Intuition based applications.

BarClock

Synopsis: Run from Workbench

Template: none

Path: ...

Requires: V2.0+

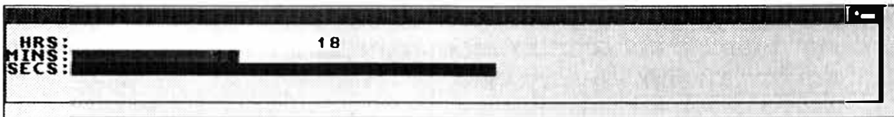
See also: ...

Type: Script

Brief: Analogue AmigaDOS clock using bar graphics

Description

This script is a demonstration of "bar" graphics – as a real-time clock. Three bars are used to represent hours, minutes and seconds elapsed. It isn't really practical to add an alarm to this script: but it is possible. Special EDIT macros are used to separate the hours, minutes and seconds from the date string: and the numbers produced determine the length of the bars.



Bar Clock

Line-By-Line

1-3. Form a standard header.

4-8. Make some essential commands resident for speed.

9-11. Create the EDIT macros used to extract the time components. Note these macros are not the most obvious solution to the problem, but were written this way to avoid a bug in EDIT's DTA command.

12. Creates the "bar" using spaces.

13. Marks the start of the clock loop.

14. Positions the cursor and switches the cursor off.

15. Sends the date to a file.

16-18. Extract the hours, minutes and seconds to three global variables.

19. Tests if the current hours variable has increased since the last loop. If not, control passes to Step 23. This block stops the bar "flashing" every loop.

20. Positions the cursor at the start of the HOURS line, clears the

entire line and sets the new background colour.

21. Displays the bar in the current background colour and appends the number of hours at the end for clarity. Note that the LENgth of the second string "$hrs" is always displayed correctly, even though its length is trimmed by LEN.
22. Sets the local HRS to the current value of the global hours#.
23. Terminates the IF...ENDIF construct opened at Step 19.
24. Tests if the current minutes variable has increased since the last loop. If not, control passes to Step 28. This block stops the bar "flashing" every loop.
25. Positions the cursor at the start of the MINS line, clears the entire line and sets the new background colour.
26. Displays the bar in the current background colour and appends the number of minutes at the end for clarity. Note that the length of the second string "$mins" is always displayed correctly, even though its length is trimmed by LEN.
27. Sets the local "mins" to the current value of the global "mins#".
28. Terminates the IF...ENDIF construct opened at Step 24.
29. Tests if the current seconds variable has increased since the last loop. If not, control passes to Step 32. This block stops the bar "flashing" every loop.
30. Positions the cursor at the start of the SECS line, clears the entire line and sets the new background colour.
31. Displays the bar in the current background colour. Seconds are not displayed.
32. If control reaches here from Step 31, it is transferred to Step 35; otherwise it continues...
33. ...here. Positions the cursor at the start of the SECS line, *does not clear* the line and sets the new background colour.
34. Displays the bar in the current background colour. Seconds are not displayed.
35. Terminates the IF...ELSE...ENDIF construct opened at Step 29.
36. Updates the local secs counter.
37. Waits a second. This probably isn't necessary given the average speed of the DOS language, but it gives other processes a look in!
38. Re-starts the loop.
39. Is some information for the WX script.

```
1.  .key dummy
2.  .bra {
3.  .ket }
```

```
4.  resident c:avail
5.  resident c:wait
6.  resident c:eval
7.  resident c:date
8.  resident c:edit
9.  echo >t:ed1{$$} "2pa/:/;sb//*np;d" ; extract SECS
10. echo >t:ed2{$$} "sa/:/*np;d;2>;3#" ; extract MINS
11. echo >t:ed3{$$} "2dta/ /*n2>;6#" ; extract HRS
12. echo >env:bar{$$} "
"
13. lab start
14. echo "*e[2;0H*e[0 p" noline
15. date >t:t{$$}
16. edit t:t{$$} to ENV:secs{$$} with t:ed1{$$}
17. edit t:t{$$} to ENV:mins{$$} with t:ed2{$$}
18. edit t:t{$$} to ENV:hrs{$$} with t:ed3{$$}
19. if $hrs{$$} NOT EQ $hrs
20.  echo "*e[40m*e[2;7H*e[K*e[2;0H HRS:*e[42m" noline
21.  echo "$bar{$$}" "$hrs{$$}" len=$hrs{$$}
22.  set hrs $hrs{$$}
23. endif
24. if $mins{$$} NOT EQ $mins
25.  echo "*e[40m*e[3;7H*e[K*e[3;0HMINS:*e[43m" noline
26.  echo "$bar{$$}" "$mins{$$}" len=$mins{$$}
27.  set mins $mins{$$}
28. endif
29. if $secs{$$} NOT GT $secs
30.  echo "*e[40m*e[4;7H*e[K*e[4;0HSECS:*e[41m" noline
31.  echo "$bar{$$}" len=$secs{$$}
32. else
33.  echo "*e[40m*e[4;7H*e[4;0HSECS:*e[41m" noline
34.  echo "$bar{$$}" len=$secs{$$}
35. endif
36. set secs $secs{$$}
37. wait 1 secs
38. skip start back
39. ;WX:WINDOW=con:0/0/550/60/Bar Clock/SMART/NOSIZE
```

BarGraph

Synopsis: [EXECUTE] Bargraph <[Data]> [Min=<min>] [Max=<max>] [Xaxis=<xais>] [Header=<text>] [Sub=<text>]

Template: data/a,min/k,max/k,xaxis/k,header/k,sub/k

Path: S:

Requires: V2+

See also:

Type: Script

Brief: Use AmigaDOS to create a bar graph

Description

Why would anyone in their right mind want to use a DOS command language to produce graphics? Back in the days when computers didn't have proper graphics, everything was done with fixed width characters. As a respectful salute to those pioneering machines I present an AmigaDOS screen-based charting program. The example given here produces automatically or manually scaled, horizontally aligned, bar charts for integer data values between -10000 and +10000.

This might appear something of an esoteric problem, but the solution addresses some interesting areas: not least how to handle fixed point arithmetic. FORTH programmers have being doing such things for years, but most of us take floating point for granted in other high-level languages such as BASIC. The theory behind such things is quite involved so the discussion is featured elsewhere in these pages. The program requires AmigaDOS 2 (sorry about that) since it makes extensive use of the new environment handler.

The BarGraph script reads data (and labels) from a text file. Typically a data file will look like this:

```
D1  200
L1  Jan
D2  325
L2  Feb
```

and so on. Data Values are prefixed by Dn and labels by Ln where "n" is the number of the data item or label (1-11 characters) attached to it. Every data item must have a label and all the items must be separated by two spaces. This is slightly more complex than spreadsheet-based graphics, but is necessary for speed; and AmigaDOS isn't exactly fast at the best of times. Once the data file

has been created, the script is called like this:

```
1>Bargraph T:Data
```

Using the default settings like this, the script determines the maximum and minimum values for the X axis by taking the highest and lowest values from the data set. This may produce unwanted results, so either or both of these can be set at run-time, for example:

```
1>Bargraph T:Data Min=50
1>Bargraph T:Data Max=3000
1>Bargraph T:Data Min=-200 Max=200
```

The number of data items that can be plotted depends on the height of the current CLI window – although the width of the plot assumes a full-width, hi-res screen. Using an interlaced screen with a larger window will afford better results.

Finally, a simple XAxis label, header and sub-header can be defined like this:

```
1>Bargraph T:Data Min=0 Max=5000 Header="Accounts"
    sub="3/1/93"  XAxis="Pounds"
```

Fixed Point Theory

It is well known that AmigaDOS does not handle numeric data particularly well. Specifically, even the simplest calculation must be carried out by a special command: EVAL. But EVAL is only capable of very simple arithmetic and does not handle decimal fractions at all. For instance: 7/2 gives 3 remainder 1 and even this must be performed in two distinct steps.

(The details following apply to any language, not just AmigaDOS and can get machine code programmers out of some tenuous situations. Unless shown, most of the results are truncated integers as would be returned by EVAL. This should be considered when checking the arithmetic with a calculator.)

Consider the sum "7/2". Using traditional methods: "two goes into seven twice, with one left over (the remainder). Pop the remainder (1) over the divisor (2) and you are left with the vulgar fraction 1/2. This is fine for dividing up a cake, but not much use in computer maths. The decimal fraction version of this is:

$$\frac{\text{Dividend}}{\text{Divisor}} = \frac{7}{2} = 3.50$$

Of course, most of us can do that in our heads, but AmigaDOS cannot. Now suppose we change the scale of the figures somewhat by multiplying just the dividend by 10.

$$\frac{(Dividend*Scaler)}{Divisor} = \frac{(7*10)}{2} = \frac{70}{2} = 35.0$$

The result is to move the (purely imaginary) decimal point one place to the right. However we have also retained the fractional part – this is the essence of fixed point arithmetic. To show this in more detail let's take a slightly more complex problem: 2/5 for instance. Using our integer AmigaDOS calculation, we get:

$$\frac{2}{5} = 0$$

Even thought the answer is 0.4. Now use a constant (K) 1000 to get something more realistic:

$$\frac{(Dividend*K)}{Divisor} = \frac{(2*1000)}{5} = \frac{2000}{5} = 400$$

$$\frac{Result}{K} = \frac{400}{1000} = 0.4$$

So far so good— but what is the point to all of this? Consider you have a set of values of between 0 and 1000 which must be scaled down to fit on an axis with 70 plottable points. The scaling factor can be calculated thus:

$$Factor = \frac{70}{1000} = 0.07$$

And any value can be plotted by multiplying it by the scaling factor 0.07. Take a value of 500 which is half-way up the scale:

Data*Factor

= 500*0.07

= 35 points

Since AmigaDOS would lose the fractional part in the original calculation – 0.07 becomes 0 – the scaled value would be useless. By using a constant of 5000, we can calculate the scaling factor thus:

$$Factor = \frac{Max\ Data * K}{Plot\ Width} = \frac{1000*5000}{70} = 71428$$

To arrive at a final result we must now divide the data by the scale factor and multiply the result by the constant:

$$Points = \frac{Data * K}{Factor} = \frac{500*5000}{71428} = 35$$

Since no fractions are involved in this calculation, AmigaDOS can cope. Provided the scaler is large enough to cope, fairly complex

arithmetic can be performed. In some cases, part of what would have been the decimal fraction is discarded – but this is common in all maths – so it is nothing to be concerned with. You can see this in action by dividing 7 by 6 – a calculation which always results in a recurring fraction 1.16666666. (or rounded up 1.167):

$$\frac{(7*10000)}{6} = \frac{(70000)}{6} = 11666\bullet$$

As a guideline, the size of the scaler determines the accuracy of the calculations: a scaler of magnitude 10000 sets an internal accuracy of one ten thousandth (the last digit is dropped due to rounding errors). Unfortunately, the size of the scaler is finite: an error will occur if, in any calculation, the scaler multiplied by the scaled data exceeds the operational limit of EVAL (in this case).

Line-By-Line

1. Defines the key as described above. Note the data file name is a required argument, all other parameters are keywords and must be supplied with the data.

2-4. Re-define the "bra", "ket" and "dollar" characters.

5. Copies the data file to the T: assignment (in RAM) with the filename "DATA".

6-8. Add EVAL, SEARCH and JOIN to the resident list. While you are entering the program the ADD switch should be omitted to save memory in case the script terminates abnormally.

9. Sets the constant "K" to 100000. This is the scaler described in the detailed description of fixed point arithmetic.

10. Sets "width" to 56 – the usable window width. This value is used to determine the scaling factor for the data. You may experiment with this value, but it must be an even number.

11. Sets scan ON. The chart is plotted in two phases, the first phase scans the data for the upper and lower boundaries – the second plots the chart.

12-14. These three local variables are set to the contents of the axis labels. If any are missing, a single space is set instead.

15-17. These set the default scan values for the lowest, middle and maximum points on the chart. Global variables are used because they can be written directly by AmigaDOS commands.

18. Directly writes the global variable "Middle" to half the value of Width.

19. As 19, but stores an adjusted value to centre the XAxis – held in "MidPoint".

20. This line is not usually used, but is provided here as an

alternative graph style. Using the listing as shown, the bars will be black. By replacing line 21 with this one, the bars appear with a hatched pattern. You can use either of these or design your own – but do not use asterisks (*) or dollars ($) since these have a special meaning to AmigaDOS.

21-22. Set the bar and XAxis styles to spaces.

23-24. Concatenate the strings defined at 21 and 22 and stores them in global variables.

25. The script will have been running for a few seconds now, so this prints a progress message to indicate the start of the scan phase. During this time the script is looking for values outside those determined by Max and Min.

26. Marks the start of the main loop.

27. Sets the global variable "Loop" to 0. This value is used as a data index as you will see later.

28. Marks the start of the scanning loop.

29. Increments "Loop" by 1.

30. Writes the global variable "Number" prefixing the value with "D" and suffixing it with a space. On the first loop therefore, Loop=D1.

31. As 30 but storing Ln in "Labels".

32. Searches the data file for the current data item (for the contents of "Number") and stores it in the global "data". The nonum switch suppresses SEARCH's unwanted line numbering facility. Using the example data file supplied here, if "Number=D1" then "Data" receives:

```
D1   2200
```

Note the entire line is read from the data file – this is corrected later on.

33-35. If the numbered data item could not be found this test forces the script to exit either the scan or display phases. Under normal circumstances this will only happen when all the data has been read and displayed.

36. String slices the numeric data from the string generated at Step 32 and stores the result in the global "NData". Using the FIRST keyword on its own forces ECHO to retrieve the whole string starting from position four and moving right. Using the current example:

```
D1   2200
```

-translates to-

```
2200
```

37-39. Tests if the current data value is greater than the current maximum chart value, and resets maximum to the data value if it is. Since this test is in a loop, all data points are tested (the scan phase) so the highest point on the X-Axis is always at least equal to the highest value in the data.

40-42. As above, but sets the lowest data point. The function "NOT GE" is IF's version of "less than".

43. If "Scan" is ON (the scan phase) control skips to Step 56 otherwise it continues at Step 44.

44. Prints the right side of the string made up from a lot of space ($XAxis) plus the X-Axis ($Axis) label. This centres it roughly over the X-Axis.

45. Calculates the scaling value described in the detailed description of fixed point. Assuming Max is 500 and Min is −500, the calculation works like this:

46. Calculates the value of the mid-point and it stores in the global "Mid".

47-50. Print the X-Axis labels. This is completed in several stages.

47. Prints some blank space characters above where the labels will appear.

48. Prints the Minimum scale value followed by some padding space – the amount of which is determined by the value of Midpoint. This is crude, but it works.

49. Does the same as above with the Middle scale value.

50. ...and this displays the highest scale value, "Max".

51-53. Display the X-Axis graticule. Not posh, but functional.

54. Changes the "scan" variable to "SHOW" so the program will enter the display phase (at Step 57).

55. Alters the display colours to white text on a black background. The text foreground colour is changed primarily to show hatched bars (an option described above) but it also makes the labels etc stand out.

56. Closes the IF...ENDIF construct opened at Step 43.

57. Tests if the "scan" variable has been set to "SHOW" (the display phase). It is important to note, since this is an iterative script, this variable is not changed until the scanning phase is completed.

58. Extracts the current label (Ln) from the data file and stores it in the global "Lab".

59. Prints the current label with the correct amount of padding spaces to position the cursor ready to print the bar. The

length of the string printed is always 10 characters regardless of the length of any particular label – longer ones are truncated.

60. Calculates the number of characters required to print the bar. Let's assume "Ndata" is 350 and "Min" is -500, the calculation works like this

    So the length of the bar representing 350 on a scale of 500 to –500 is 47 characters of 56 possible (as defined in "Width" – see above). Work that out on a calculator and you will find the result is out by a fraction – but such things are outside the limits of the display. Using a proportional font (such as Times) could improve the resolution ten-fold, but the console device cannot cope in Release 2.

61. Changes the background colour to default (slate-grey). This, like many of the other colour changes *must* be done separately otherwise the string slicing will get in the way with unpredictable consequences.

62. Prints the current label with one extra padding space.

63. Changes the background colour to black.

64. Prints a bar according to the magnitude of the current data. See Step 60 for details of how the variable "Dlen" is calculated.

65. Changes the background to slate-grey...

66. ...and prints the actual value of the data displayed. This is an optional feature but has been included here to help overcome the deficiencies with the display resolution.

67. Terminates the IF...ENDIF construct opened at Step 57.

68. Jumps back to Step 28 and does it all again for the next data point!

69. Marks the escape point. Control jumps here from Step 34 when the last data point has been read *or* charted and the data table is exhausted.

70. Checks if the scan phase is still active. If the scan has been completed control jumps to Step 74 and exits, otherwise...

71. ..."scan" is set to OFF to mark the end of the scan phase and the start of the charting phase and...

72. Control jumps right back to the start and does the whole lot again, this time it's for real.

73. Terminates the IF...ENDIF block opened at Step 71.

74-76. Puts everything back to normal, prints the header and sub-headers...

77-80. ...and finally removes SEARCH, JOIN and EVAL from the resident list.

Listing

```
1.  .key data/a,min/k,max/k,xaxis/k,header/k,sub/k
2.  .bra {
3.  .ket }
4.  .dollar |
5.  copy {data} T:data
6.  resident c:eval add
7.  resident c:search add
8.  resident c:join add
9.  set K 100000
10. set Width 56
11. set scan ON
12. set Header {header|" "}
13. set Subhead {sub|" "}
14. set Axis {Xaxis|" "}
15. setenv max {max|1}
16. setenv min {min|0}
17. setenv mid 1
18. eval $Width/2 to env:Middle
19. eval $Middle -2 to env:MidPoint
20. ;echo >T:G "/\/\/\/\/\/\/\" noline
21. echo >T:G "                " noline
22. echo >T:H "          " noline
23. join T:G T:G T:G T:G T:G T:G T:G T:G T:G as ENV:Bar
24. join T:H T:H T:H T:H T:H T:H T:H T:H T:H as ENV:XAxis
25. echo "Scanning..."
26. lab again
27. setenv loop 0
28. lab Read_loop
29. eval $loop + 1 to env:loop
30. eval $loop to env:number lformat "D%n "
31. eval $loop to env:labels lformat "L%n "
32. search >env:data T:data $number nonum
33. if warn
```

```
34.  skip done
35. endif
36. echo "$data" first=4 to env:ndata
37. if val $ndata GT $Max
38.  setenv Max $ndata
39. endif
40. if val $ndata NOT GE $Min
41.  setenv Min $ndata
42. endif
43. if $scan EQ OFF
44.  echo "$XAxis $XAxis $Axis" len=$Middle
45.  eval (($Max-$Min) * $K)/$Width to env:Scale
46.  eval (($Max-$Min)/2)+$Min to env:Mid
47.  echo "          " noline
48.  echo "$Min$XAxis" first=1 len=$MidPoint noline
49.  echo "$Mid$XAxis" first=1 len=$MidPoint noline
50.  echo "$max"
51.  echo "..........!" noline
52.  echo "........................!" noline
53.  echo "...........................!"
54.  set scan SHOW
55.  echo "*e[41m*e[32m" noline
56. endif
57. if $scan EQ SHOW
58.  search >env:lab T:data "$labels" nonum
59.  echo >env:labels "$lab$XAxis" first=4 len=10 noline
60.  eval ( ($ndata - $min) * $K ) /$Scale to env:dlen
61.  echo "*e[44m" noline
62.  echo "$labels " len=10 noline
63.  echo "*e[41m" noline
64.  echo "$Bar" first=1 len=$dlen noline
65.  echo "*e[44m" noline
66.  echo "$ndata" len=12
67. endif
68. skip Read_loop back
69. lab done
70. if $scan EQ ON
```

```
71.   set scan OFF
72.   skip again back
73. endif
74. echo "*e[44m*e[31m"
75. echo "*e[I$Header"
76. echo "*e[I$Subhead"
77. resident c:eval add
78. resident c:search add
79. resident c:join add
```

Listing

Sample data file (only five items shown).

```
D1  2200
D2  2300
D3  2400
D4  2100
D5  2700
L1  January
L2  February
L3  March
L4  April
L5  May
```

Booty

Synopsis: [EXECUTE] Booty <Drive> Name

Template: DRIVE/A,NAME

Path: S:

Requires: V1.3+

See also: ...

Type: Script

Brief: Make a boot disk

Description

This is a simple script to interactively create boot disks. It is designed primarily for AmigaDOS 1.3 and 2. Calling the script is a simple matter of supplying the drive number and (optionally) a name for the new disk. The disk is formatted, initialised and all major directories are created and "stuffed". Booty works best with at least two drives or a hard disk. Single drive users especially will benefit by using IntelliRes (detailed later) on this script.

This script is an example only - you should examine it and modify it to suit your own wants and requirements. For instance, Workbench 3 users could use REQUESTCHOICE in place of the ASK commands. Experienced users can modify the script to select the correct handlers, devices and so on. It's up to you.

Example

```
1>Booty 0 Works
```

Line-by-Line

1. Defines the key as having one required argument for the drive number and one optional argument for the volume name.
2. If a volume name is not supplied, this sets the default one.
3. Prints a totally pointless progress message. It's there because it gives people a sense of achievement.
4. Formats the disk using the normal system command. This line assumes you have the System directory defined in your path setting and will produce some interactive output. It is possible to start the format straight away but that was considered much too volatile a technique to use. You may wish to add DCFS FFS or INTL switches here. See MAD 3 Reference
5. Installs (adds boot code) to the disk thus making it a blank

boot disk. You could end the script at this point and copy your own information to it if you prefer.

6. Checks to see if you want the fonts copying from your current system disk. (This test was added because fonts are not required for all applications and they do take up a lot of room.) Capital N serves as a reminder that the default action is NO.
7. The WARN flag is set at Step 6 if you answer Y. In this case control continues at Step 8, otherwise it branches to Step 9.
8. Copies all the fonts from the current boot disk onto the new boot disk.
9. Control reaches here if you answered "N" to the question at Step 7 and continues at Step 10. If you answered "Y" it jumps to Step 11.
10. Creates an empty fonts drawer on the new boot disk. This isn't absolutely necessary but it should be retained for the sake of keeping things clean.
11. Terminates the IF...ELSE...ENDIF construct opened at 7.
12. Tests if you need a complete System directory (FORMAT, FIXFONTS etc).
13. If you answer Y at Step 12, the WARN flag is set and execution continues at Step 14, otherwise it jumps to Step 16.
14. Copies the entire System directory from the current system disk onto the new boot disk...
15. ...and this copies that all-important Shell icon.
16. If execution arrives here from Step 15 it jumps to Step 20, otherwise it continues at Step 17.
17. As Step 15.
18. Creates the system directory and copies the system directory commands: FastMemFirst and SetMap. These are required by the Startup-sequence/Note changes for 2.1 and higher.
19. Closes the IF...ELSE...ENDIF construct opened at Step 13.
20. Makes the Utilities directory (nothing is placed here and no Workbench icon is created).

21-25. Copies the handler library, libraries, Devices, script and command directories to the new boot disk – generally speaking these lines should be left as they are.

26. Copies the Preferences directory over. This line is really only required for Workbench 2.
27. Guess what!

Listing

```
1.   .KEY DRIVE/A,NAME
2.   .DEF NAME Lazy_Bones
3.   Echo "Making a simple boot disk - please wait"
4.   FORMAT DRIVE DF<DRIVE>: NAME <NAME>
5.   INSTALL DF<DRIVE>:
6.   Ask "Do you require fonts y/N?"
7.   IF WARN
8.    COPY Fonts: DF<DRIVE>:Fonts ALL
9.   ELSE
10.   MAKEDIR DF<DRIVE>:Fonts
11.  ENDIF
12.  Ask "Do you require a complete system y/N?"
13.  IF WARN
14.   COPY SYS:System DF<DRIVE>:System ALL
15.   COPY SYS:Shell.info DF<DRIVE>:
16.  ELSE
17.   COPY SYS:Shell.info DF<DRIVE>:
18.   COPY SYS:System/(FastMemFirst|SetMap) DF<DRIVE>:System
19.  ENDIF
20.  MAKEDIR DF<DRIVE>:Utilities
21.  COPY L: DF<DRIVE>:L ALL
22.  COPY Libs: DF<DRIVE>:LIBS ALL
23.  COPY Devs: DF<DRIVE>:DEVS ALL
24.  COPY S: DF<DRIVE>:S ALL
25.  COPY C: DF<DRIVE>:C ALL
26.  COPY SYS:Prefs DF<DRIVE>:Prefs ALL
27.  Echo "Operation complete..."
```

CALC1.3

Synopsis: [EXECUTE] CALC [[val=]value1] [[op=]operator] [[val2=]value2]

Template: val1,op,val2

Path: S:

Requires: V1.3

See also: ...

Type: Script

Brief: Attempt to patch one of the bugs in EVAL

Description

AmigaDOS 1.3's EVAL command has a problem – if you get the spaces in the wrong place, it doesn't work. This script attempts to solve the problem by analysing the input line and deciding if it received enough parameters. This is a highly modified version of an example to be found in *Mastering AmigaDOS Reference* which relies on /a required arguments to generate an error. This version gives the users some polite help when they stumble...

Line-by-Line

1. The argument key has three arguments. These would normally be required but I'm assuming someone using this is going to need help. The last thing they're going to need is EXECUTE screaming "args no good for key..." which is what would happen if one of those arguments was required.

2-3. Set the bra and ket characters to { and }.

4-9. If one of the required arguments is missing this code generates some help. Required arguments could have been used in the template (at Step 1) but one feature of this script is to show how interactive help can be used.

10. If control reaches this point when the correct arguments have been supplied, it continues at Step 11; otherwise it jumps to Step 12.

11. The reason EVAL failed in the first place haunts execute too. That is: if two (or more) optional arguments run into each other, they appear just like one argument. And we're going to use that very feature to catch the bug. Consider a command line which reads:

```
CALC 1 +2
```

What actually happens is this: The "1" is assigned to "Val1";

"+2" is not separated by a space so that gets assigned to "op". This leaves "val2" empty – so we check for a null value for "val2" and, if found, give some help. In practice you can give as much or as little help as you think is required. Be brief: you can always refer to the documentation files if need be. This is only mentioned in passing because you may note the use of "**" in the help string. This is because this example needs to print a literal "*" (multiply).

12 This isn't strictly necessary at the end of a script, but it's good practice.

Listing

```
1   .key val1,op,val2
2   .bra {
3   .ket }
4   if {val2} EQ ""
5   echo "Argument missing - help is:"
6   echo "Usage: CALC <value1> <operator> <value2>"
7   echo "All arguments are required."
8   echo "Value1 and value2 use 0x for hex"
9   echo "Operator is one of - +,-,**,/,etc..."
10  else
11  eval {val1} {op} {val2}
12  endif
```

Monthprint

Synopsis: [EXECUTE] MonthPrint

Template: none

Path: S:

Requires: V2+

See also: Calendar

Type: Script

Brief: Month printing module for the Calendar script

Description

One of the most irritating features with AmigaDOS is its low speed – and the calendar script is no exception. However, any operation always seems quicker if you can see something happening. For instance, when Workbench is busy, it displays a sprite (clock or Z's bubble) to show it's working; hacks such as Sleepy 3 go further by animating the sprite.

Long operations should have some form of progress indicator and this is the method chosen for Calendar's display module, Monthprint. The script's progress is shown in the form of a bar traversing the screen from 0 to 100 per cent completed.

Progress indicators can be implemented in several ways – the choice of which method to use depends on how the script works. In linear scripts you update the indicator at strategic points – after a long copying operation for instance. Looping scripts are easier, you update once every loop. In such cases it is also much easier to determine the length of the progress bar because you can usually determine how many loops will be performed in advance. This is the method used by Monthprint – the length of the progress indicator is calculated from the number of days in the month.

Line-by-Line

1. Prints a simple message to let you know what's going on.
2. Displays the fixed part of the progress indicator bar using string slicing. For the sake of illustration, let's imagine the program was displaying a (purely hypothetical) seven day month. The variable "DiM" contains eight loops so the printed result from this step looks like this. (The extra space is picked out with a period and the cursor position with an asterisk.):

   ```
   0%.——-*
   ```
3. Adds the second part of the progress indicator. Note how this

line looks a little strange at first glance. It sends a line feed, adds some spaces and then stops ECHO from printing a line feed. The screen display now looks like this:

```
0%.——-.100%
...*
```

As you can see, the cursor has been moved to the first position in the progress indicator, just below the first bar. This explains the strange use of "*n" and NOLINE switch.

4. Defines a loop label "loop" which is accessed by the backward jump at 20. The loop is defined as early as is practical in the script to help speed things up. When jumping backwards, the SKIP command starts at the beginning of the program and works its way down. If you must jump backwards, keep the labels early on.

5. In BASIC this line reads:

   ```
   IF DiM > daynum
   ```

   This tests if the value in "DiM" (Days in Month) has exceeded the value in "daynum" and branches accordingly. The variable daynum is initialised to "1" in the Calendar script.

6. This calculates the value held in the global environment variable "wrap" which is used later to determine when to wrap the display. The variable "wrap" is calculated each loop to contain a value between 0 and 6 – the offset of the current date in the week. The calculation uses a technique which is not available in AmigaDOS 1.3 since it writes directly to the variable being used. In early versions, EVAL opens a file to the variable and keeps it open until the command has completed. Since you cannot write to a file which is open for reading this was not possible. Assuming "wrap" contains 5, AmigaDOS treats this line thus:

   ```
   EVAL (5 +1) mod 7 to ENV:wrap
   ```

   The file containing the variable is opened, read and closed, while the line is being parsed. When EVAL gets round to executing it sees the variable just as if it had been typed.

7. In BASIC this line could be written:

   ```
   IF daynum <= 9
   ```

   The test determines if the value stored in "daynum" is less than or equal to 9. If the test is positive, control branches to Step 9, otherwise it continues...

8. ...here where a single space is added to the print file, "MFile". (Mfile was created by Calendar.) This handles the character alignment by making sure all the numbers line up neatly.

Since all the numbers appear a+ regular tab stops (accomplished with the string *e[I), single digits line up over the tens column, viz:

This	becomes	this
1	⟶	1
8	⟶	8
15	⟶	15

You may exclude Steps 7 to 9 if you wish.

9. Closes the IF...ENDIF construct opened at Step 7.

10. Tests if the value of "wrap" is less than or equal to 1. (These values occur when the day is a Saturday or Sunday.) If it is, control continues at Step 11 otherwise it branches to Step 12.

11. Control reaches here if the date being displayed falls on a weekend. This special exception is highlighted by adding control characters to the output string: "*e[32m" turns the printed output white and "*e[31m" puts it back to normal. See Step 13 for more information.

12. If control reaches here from Step 11 it branches to Step 14 otherwise it continues...

13. ...here, where it prints the next day number. A couple of things are worth noting here. First the output is sent to file for later display, but second two separate items are being printed. There's nothing unusual in that, but look at how this is achieved:

```
echo >>T:MFile $daynum "*e[I" noline
```

ECHO is receiving two print arguments ($daynum and "*e[I") instead of the more usual one. This is a unique feature of AmigaDOS 2 and cannot be used in earlier versions. In fact, you can send as many arguments as you like, switches such as NOLINE should be added to the end for clarity.

14. Closes the IF...ELSE...ENDIF construct opened at Step 10.

15. Increments the variable "daynum" using the direct write technique described at Step 6.

16. Checks if the value held in "wrap" is zero. If it is, control resumes at Step 17 otherwise it jumps to Step 18.

17. Adds a new line to the print file "MFile".

18. Closes the IF...ELSE...ENDIF construct opened at Step 16.

19. This displays the progress meter block for the current loop. Note this is echoed directly to the current console screen and not sent to file.

20. Jumps back to Step 4 for another bite at the cherry.
21. Closes the IF...ELSE...ENDIF construct opened at Step 5. Control reaches here when the entire month has been sent to the print file.
22. Appends the bottom "ruler" to the print file...
23. ...which is finally displayed here using MORE. Note that since MORE is not RUN-launched it uses the current Shell window for display. This also clears the progress indicator.
24. Prints a black line.
25. Removes EVAL from the resident list since it is no longer required...
26. ...and closes the Shell process opened by Calendar.

Listing

```
1.  echo "Calendar Working... wait"
2.  echo "0% ——————————————" first=1 len=$DiM noline
3.  echo "– 100%*n    " noline
4.  lab loop
5.  if val $DIM GT $daynum
6.  eval ( $wrap +1) mod 7 to env:wrap
7.  if val $daynum NOT GT 9
8.  echo >>T:MFile " " NOLINE
9.  endif
10. if val $wrap NOT GT 1
11. echo >>T:MFile "*e[32m$daynum*e[31m" "*e[I" noline
12. else
13. echo >>T:MFile $daynum "*e[I" noline
14. endif
15. eval $daynum + 1 to env:daynum
16. if $wrap eq 0
17. echo >>T:Mfile ""
18. endif
19. echo "*e[41m *e[40m" noline
20. skip loop back
21. endif
22. echo >>T:Mfile "*n======================================="
23. more T:Mfile
24. echo ""
25. resident eval remove
26. endcli
```

Calendar

Synopsis: [EXECUTE] Calender <[YEAR=] year> [month]

Template: year/a, month

Path: S:

Requires: V2+

See also: MonthPrint

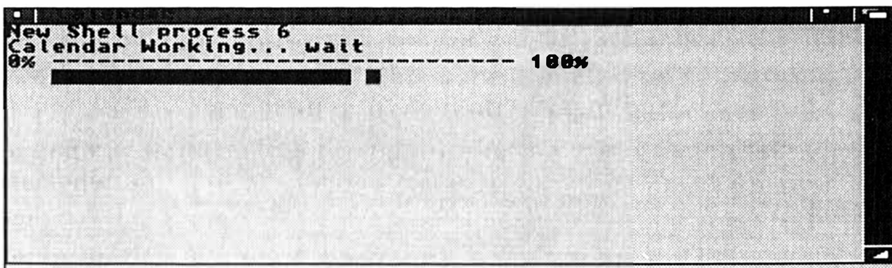
Type: Script

Brief: Main module for the AmigaDOS calendar

Description

Zeller's congruence is something of a mouthful after, say, a few pints; anything mathematical brings tears to my eyes. Zeller's congruence is a complex integer-based formula to calculate the day number of the first day in any year from the start of the Georgian calendar (1582) to well into the next millennium; including leap years. It's just as complex to express as a mathematical formula too. Nevertheless, Zeller's mathematical prediction is widely used in applications such as perpetual digital calendars.

The first day numbers (there's seven of them from zero to six) are fixed and it is possible to program say, a watch, with a hundred or so in packed binary (two values per byte) and use them to fix the calendar. However, that approach is a bit feeble so here's how to program the congruence in AmigaDOS with a complete calendar program to boot. As you will see the maths are quite easy, the hard part is making use of the figures!



Calendar 1

First though, here is one way to express Zeller's congruence in most versions of BASIC:

```
10 INPUT "Year", Year
20 Century=INT( (Year-1)/100)
30 Decade=Year-1-100*Century
40 Day=(799+Decade+(Decade/4)+(Century/4)-(2*Century)) MOD 7
```

```
50 PRINT "Day number is: ";Day
```

Looks pretty hair-raising at first glance doesn't it – but it breaks down quite well. Lines 20 and 30 split the year into two parts – the century number (1800, 1900, 2000 etc) divided by 100; and the decade number minus one. Therefore, 1992 breaks down thus:

```
Century=19

Decade = 91
```

Line 30 uses these values to calculate the number of the first day in January of any particular year. In 1992 for instance the first day is Wednesday, so the result is 3 (where Sunday=0 and Saturday=6). Essentially this is just a piece of simple arithmetic and even AmigaDOS 2 can handle that without too many problems.

The script programs presented here are not suitable for earlier versions of AmigaDOS because of the advanced maths and variable handling, but if enough of you make a fuss, I will attempt to re-program this example for AmigaDOS 1.3.2. This sort of problem is not suitable for AmigaDOS 1.3 because the EVAL command did not support multiple arguments. Enthusiastic owners might like to try this as an exercise.

Calendar is divided into two separate scripts for speed. The first is a linear script which does all the necessary calculations, the second displays an entire month. It is quite possible to write this program as a single script, but since the printing side performs a lot of backward loops, it is faster to do it this way. Let's take a close look at how the main part works.

Line-By-Line

1. Defines the arguments. Calendar only requires a year to work, but you can supply a month number too. The month argument could have been a month name, but this just adds complexity and means you have to type more. .

2-4. Re-defines the bra, ket and dollar symbols. Dollar is changed here to make the script easier to read – you'll see why later on.

5. Preloads EVAL into memory for speed. Note the ADD argument is supplied here so the command can be safely removed without affecting any other scripts.

6-7. Creates local environmental variables "Y" and "M" containing the year and month (if any) specified from the command line.

8. Subtracts 1 from the year number and stores the result in the global environmental variable, "Date". (You should note here, the dollar symbol is used to signify an environmental variable – it is not affected by the .DOLLAR command used earlier.)

9. Subtracts 1 from the month number and stores the result in the variable, "Month".

10. This is a natty little trick to remove the century number from the date variable. Assuming the value held in Date was 1991, it works like this:

```
ECHO "$Date"
```

is read by AmigaDOS thus:

```
ECHO "1991"
```

because the local variable is expanded as the command executes. This is then affected by the FIRST and LEN keywords - FIRST=1, tells ECHO to display the leftmost character on the string. LEN=2, makes ECHO display just two characters – in other words the century number. In fact, this value is not displayed, instead it is sent to a new global environmental variable, "Cent".

11. Like step 10, this removes two characters from the "Date" variable. However, since the FIRST keyword is not supplied, ECHO reads the rightmost two characters – the Decade in other words. As before, this value is used to create an environmental variable (Decade).

12. This looks a lot worse than it really is! It uses the BASIC translation of the Zeller's congruence method described above to calculate the day number of the first day in the required year. A point worth noting here is there *must* be a space before the dollar symbol used to signify an environmental variable. The result is stored in (yet another) global variable, "Day".

13. It is an interesting fact that you can determine if a year is a leap year (29 days in February) by performing modulo 4 on it. Leap years always return a value of 0. This calculation performs MOD 4 on the year number (supplied at the command line) and stores the result in the global environmental variable, "Leap". Just to aggravate matters though, most centuries are *not* leap years. A century must be divisible by 4 (1600, 2000, 2400 etc) for it to be leap year.

14. Tests the value of "Leap" and determines what to do next. If the year is not a leap year, execution continues at Step 15; if it is, execution branches to Step 19.

15-16. This two-part step does some string slicing to obtain a value from an array of numbers. Each of the twelve month's in a year has a particular number of days, you knew that much of course – but the computer does not. In BASIC for instance, you would set up an array like this:

```
FOR N=1 TO 12
```

```
READ DaysInMonth(N)
NEXT N
DATA 31,28,31,30,31,30,31,31,30,31,30,31
```

and read the array thus:

```
Days=DaysInMonth(Month)
```

where the variable "Month" selects the correct element from the array. AmigaDOS cannot handle arrays in this way – but by careful use of string slicing (and some careful typing) this can be achieved quite simply. I'll explain this step in detail because it occurs several times in this script.

The first job is to construct the array of numbers. This is just the number of days in each month as demonstrated in the BASIC example above. To keep the script easy to read (and debug) the list is constructed with full stops between each value – although this is not strictly necessary. This leaves something like this:

```
".31.28.31.30.31.30.31.31.30.31.30.31"
```

Each number is three characters long, therefore you can pick any value by multiplying the offset (the month number) by three. A feature of AmigaDOS means the first character in the string is numbered one. Also, since the months start from zero (determined earlier) we must add two to get the correct offset. If that makes your brain itch, consider this:

Take June – month number five. In the script, the variable "Month" will be holding four. Therefore:

```
Offset = (4*3)+2 = 14
```

The 14th and 15th characters in from the start of the data are "31", the fifth number in the data. Taking this offset as a start value and reading two characters, you can create an environmental variable. Here's how:

15. Calculates the starting position using the environmental variable Month and sending the offset result to global environemtal variable, "Slice".
16. Starting from the position determined by "Slice" this takes two characters from the data string and saves the result in "DiM" (Days In Month).
17. Creates another offset variable, which is used to read the data at Step 18...
18. ...here. This data is the number of days in the year that have elapsed at the start of the current month. Note this table is almost identical to the first one except the numbers are two

or three characters long. To read a data table in this way, it is vital all strings are the same length. Therefore, if a number is composed of just two digits, it must be preceded by a padding space.

19. If control reaches here from Step 18, it branches to Step 24 otherwise it continues at Step 20.

20-23. These lines are essentially the same as 15-18, however these data strings are used for leap year exceptions. The data changes after February which has 29 days in this case.

20. Calculates the offset variable used at Step 21...

21. ...which is used to determine the number of days in the selected month. This value is then sent to the variable, "DiM".

22. Calculates the offset variable used at Step 23...

23. ...which determines how may days have elapsed up to the current month. It is important to note when you enter this program, *all* but three of the values change in this data set!

24. Closes the IF...ELSE...ENDIF construct opened at Step 14.

25. Prepares another string slice offset. This one is used at Step 27 to grab the month name.

26. Creates a text (MFile) file in T: with an initial string. Note here, the NOLINE switch is used to suppress the extra line feed. At this stage MFile contains:

```
Calendar for:
```

27. Uses ECHO's string slicing facilities plus the append to file operator (>>) to attach the current month name to the message string, MFile. If month 4 had been requested, MFile now contains

```
Calendar for: Apr
```

28. Next, the year is added. This is taken from the local environmental variable (Y) created at Step 7. MFile now looks like this:

```
Calendar for: Apr 1992
```

29. This appends a "ruler" to the message file. (Equals signs are used here, but you can used any convenient character.) Note the line feed at the start of the line:

```
Calendar for: Apr 1992
 ====================
```

30. This appends the "day names" heading to the message file. Note how "*e[I" (TAB) escape sequences are used to tabulate the text correctly.

31. Like Step 31, this adds rules to the day names. You can use any characters you prefer here, but you should keep the tab sequences.

32-34. Calculate the initial print position of the first date under the day name rules. Since this is quite an involved topic, I'll look at it in a bit more detail. The idea is quite simple, the day names appear across the top from Sunday to Saturday like this:

```
Sun Mon Tue Wed Thu Fri Sat
=== === === === === === ===
```

Now, let's take January 1992 (January is the simplest month). The 1st is a Wednesday (Day=3) so the program has to start printing 24 characters (1 TAB=8 characters) in from the start, like this:

```
Sun Mon Tue Wed Thu Fri Sat
=== === === === === === ===
                  1   2   3   4
  5   6   7   8   9  10  11
```

This is quite simple to produce using the formula:

```
Space = Day * 8
```

But what happens later in the year? Take May for instance. The 1st of May 1992 is a Friday so how can we calculate that from the day number returned from Zeller's congruence? This is where the "Elapsed" variables determined at Steps 21 and 23 come into effect. These determine the number of days elapsed up to the start of the current month. At the first of January, no days have elapsed, but by the first of May, 121 days have passed. By adding this to the initial day number and dividing by seven, the remainder is the offset to the first day in the week. The formula is therefore:

```
Space = ((Day+Elapsed) MOD 7) * 8
```

32. This is the AmigaDOS version of the above calculation. "Elapsed" and "Day" are summed first. Then the modulo (remainder after division) is taken and stored in "Day". The calculation is split in two like this because the value of "Day" is required elsewhere.

33. The new value of Day is multiplied by 8 and stored in the new global environmental variable, "Space".

34. Uses ECHO's string slicing function to produce an effect similar to the STRING$() function found in most modern BASICs. Note in the listing these are shown as periods (.) but

they should be entered as spaces.

35. Sets the global environmental variable, "daynum" to 1. "Daynum" is used by the display script.

36. Copies the current value of "Day" to a new global environmental variable, "wrap" (used by the display script). It is interesting to note, this operation could be accomplished by COPY. However, EVAL has been used because that command is made resident for the script.

37. Increments the value held in "DiM" by 1.

38. Starts the display script, MonthPrint although it is important to note how this has been achieved. In normal circumstances, the script would be called using either EXECUTE or RUN EXECUTE; the latter being the closest approximation to the final solution. Using NEWSHELL allows you to effectively RUN launch EXECUTE and specify a window size at the same time.

Listing

```
1.  .key year/a, month
2.  .bra {
3.  .ket }
4.  .dollar !
5.  resident c:eval add
6.  set M {month}
7.  set Y {year}
8.  eval $Y-1 to env:Date
9.  eval $M-1 to env:Month
10. echo "$Date" first=1 len=2 to env:Cent
11. echo "$Date" len=2 to env:Decade
12. eval (799+ $Decade+($Decade/4)+($Cent/4)-(2* $Cent)) mod
    7 to env:Day
13. eval (($Cent+1) + $Y) mod 4 to env:leap
14. if val $leap NOT EQ 0
15.  eval $month  * 3 +2 to env:slice
16.  echo ".31.28.31.30.31.30.31.31.30.31.30.31" first=$slice
    len=2 to env:DiM
17.  eval $month  * 4 +2 to env:slice
18.  echo "..00..31..59..90.120.151.181.212.243.273.304.334"
    first=$slice len=3 to env:Elapsed
19. else
20.  eval $month  * 3 +2 to env:slice
```

```
21.  echo ".31.29.31.30.31.30.31.31.30.31.30.31" first=$slice
    len=2 to env:DiM
22.  eval $month  * 4 +2 to env:slice
23.  echo "..00..31..60..91.121.152.182.213.244.274.305.335"
    first=$slice len=3 to env:Elapsed
24. endif
25. eval $month  * 4 +2 to env:slice
26. echo  >T:MFile "Calendar for: " noline
27. echo >>T:MFile " Jan Feb Mar Apr May Jun Jul Aug Sep Oct
    Nov Dec" first=$slice len=3 noline
28. echo >>T:MFile " $Y"
29. echo >>T:MFile "*n======================================"
30. echo >>T:Mfile
    "Sun*e[IMon*e[ITue*e[IWed*e[IThu*e[IFri*e[ISat"
31. echo >>T:MFile
    "===*e[I===*e[I===*e[I===*e[I===*e[I===*e[I==="
32. eval ( $Elapsed + $Day ) mod 7 to env:Day
33. eval $Day * 8 to env:Space
34. echo >>T:MFile ".............." first=1 len=$space noline
35. setenv daynum 1
36. eval $day to env:wrap
37. eval $DiM + 1 to env:DiM
38. newshell from s:MonthPrint  con:0/0/480/140/Calendar/Auto
```

Monthprint2

Synopsis: [EXECUTE] MonthPrint2

Template: none

Path: S:

Requires: V2+

See also: Calendar2, SuperCal

Type: Script

Brief: Month printing module for the Calendar2 script

Description

Although given here as a complete listing, there are only a few differences between this module and the one listed elsewhere in the book. The changes are noted below.

Line-by-Line

The following changes have been made to Monthprint:

1. Cosmetic change to be more informative

11 and 13. Tabs replaced by spaces.

23. Signals to Calendar2 it is safe to continue by breaking the wait state.

Listing

```
1.  echo "Calendar Working on $MName $Y. Please wait"
2.  echo "0% ——————————————" first=1 len=$DiM noline
3.  echo "— 100%*n   " noline
4.  lab loop
5.  if val $DIM GT $daynum
6.  eval ( $wrap +1) mod 7 to env:wrap
7.  if val $daynum NOT GT 9
8.  echo >>T:MFile " " noline
9.  endif
10. if val $wrap NOT GT 1
11. echo >>T:MFile "*e[32m$daynum*e[31m" "   " noline
12. else
13. echo >>T:MFile $daynum "   " noline
14. endif
```

```
15. eval $daynum + 1 to env:daynum
16. if $wrap eq 0
17. echo >>T:Mfile ""
18. endif
19. echo "*e[41m *e[40m" noline
20. skip loop back
21. endif
22. echo >>T:Mfile "*n================================="
23. break $BreakMe C
24. endcli
```

SuperCal

Synopsis: [EXECUTE] Supercal <[YEAR=] year> [Month]

Template: Year/A,Month

Path: S:

Requires: V2+

See also: Calendar2, Monthprint2

Type: Script

Brief: Full year version of the AmigaDOS calendar program

Description

Supercal is the main module for Calendar2, the whole year calendar. For the sake of speed, it operates as a separate module which calls modified versions of Calendar and Monthprint.

Line-By-Line

1. Define the key for this command. You must specify a year for this script and optionally a starting month. For instance you might only want the calendar from August 1992. Note however, both these arguments are numeric.
2. Just in case you don't supply a month, Supercal assumes you mean January – this will normally be the case since Supercal is designed to display whole year calendars.
3. Checks if the Calendar variable has been set – this is used to check for certain once only configuration. If Calendar exists control branches to Step 7, otherwise execution continues...
4. ...here, where the Calendar variable is defined.
5. Adds the EVAL command to the resident list – this used to be done in Calendar, but since that is now a subroutine of this script, it is done here.
6. Creates the print file and defines its heading with the current year.
7. Closes the IF...ENDIF construct opened at Step 3.
8. Defines a global environmental variable MN and gives it the value of the current month.
9. Prints a simple progress message in the current console window. This is the working window that Supercal was launched from.
10. Executes Calendar2 (listed below) with the correct parameters.

11. Increments the month number.
12. Tests if the whole year (up to December has been done). If it has, execution branches to Step 14 otherwise it continues...
13. ...here, which calls Supercal itself recursively. This has been done in preference to using RUN because that would cause more than one occurrence of Calendar2 to execute at once and that cannot happen. Using EXECUTE on its own does not work because this script contains a backward loop and the temporary command file required by the SKIP command is trashed by the second EXECUTE running from the same process. (Phew!) Don't worry, it just works that way.
14. Closes the IF...ENDIF construct opened at 12.
15. Displays the completed Calendar.
16. Removes EVAL from the resident list since we've now finished with it.
17. Makes a copy of the print file in your S: assignment...
18. ...and lets you know for future reference. This is the print file. You can make your own calendar by copying this to your printer thus:

    ```
    COPY S:Calendar1992 to PRT:
    ```

19. Finally, makes sure no recursive copies of the script are left to execute.

Listing

```
1.  .key Year/A,Month
2.  .def Month 1
3.  if *$Calendar EQ $Calendar
4.   setenv Calendar ON
5.   resident c:eval add
6.   echo >T:MFile "Calendar <Year>"
7.  endif
8.  setenv MN <Month>
9.  echo "Working on $MN/<Year>"
10. execute s:Calendar2 <Year> $MN
11. eval $MN + 1 to ENV:MN
12. if val $MN NOT GT 12
13.  execute s:Supercal <Year> $MN
14. endif
15. more T:MFile
```

```
16. resident eval remove

17. copy T:MFile to S:Calendar<Year>

18. echo "Calendar saved to disk as Calendar<Year>…*nSee text
    for info on how to print this."

19. quit
```

Calendar2

Synopsis: [EXECUTE] Calendar2

Template: none

Path: S:

Requires: V2+

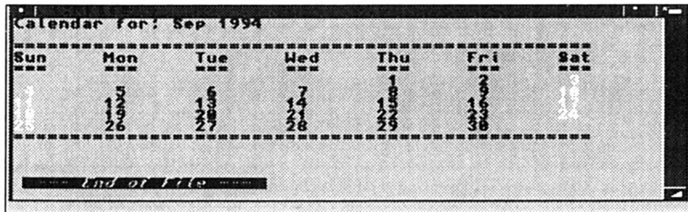
See also: MonthPrint2, SuperCal

Type: Script

Brief: Improved calendar script

Description

The only changes between this version and the one listed elsewhere are as follows (line numbers refer to this version):



Calendar 2

Line-By-Line

25. The month string is now written to a global environmental variable.

26-29. Tabs removed and other cosmetic improvements.

32. $Day is now multiplied by five to account for new spacing.

37. A new variable has been added here to aid the multi-tasking. "Breakme" is set to the current process number running Calendar2.

38. NEWSHELL calls a different file and the window has been made smaller.

39. An extra line forces the script to wait until the Monthprint2 process is completed. Redirection to NIL: is used to stop the "***Break" message appearing.

Listing

```
1.   .key year/a,month
2.   .bra {
3.   .ket }
4.   .dollar !
```

```
5.  set M {month}
6.  set Y {year}
7.  eval $Y-1 to env:Date
8.  eval $M-1 to env:month
9.  echo "$Date" first=1 len=2 to env:Cent
10. echo "$Date"         len=2 to env:decade
11. eval (799+ $decade+($decade/4)+($cent/4)-(2* $cent)) mod
    7 to env:day
12. eval (($Cent+1)+$Y) mod 4 to env:leap
13. if val $leap NOT EQ 0
14. eval $month  * 3 +2 to env:slice
15. echo " 31 28 31 30 31 30 31 31 30 31 30 31" first=$slice
    len=2 to env:DIM
16. eval $month  * 4 +2 to env:slice
17. echo "  00  31  59  90 120 151 181 212 243 273 304 334"
    first=$slice len=3 to env:Elapsed
18. else
19. eval $month  * 3 +2 to env:slice
20. echo " 31 29 31 30 31 30 31 31 30 31 30 31" first=$slice
    len=2 to env:DIM
21. eval $month  * 4 +2 to env:slice
22. echo "  00  31  60  91 121 152 182 213 244 274 305 335"
    first=$slice len=3 to env:Elapsed
23. endif
24. eval $month  * 4 +2 to env:slice
25. echo  >ENV:MName " Jan Feb Mar Apr May Jun Jul Aug Sep
    Oct Nov Dec" first=$slice len=3 noline
26. echo >>T:MFile "*n=========== $MName $Y ------------"
27. echo >>T:MFile "================================="
28. echo >>T:Mfile "Sun  Mon  Tue  Wed  Thu  Fri  Sat"
29. echo >>T:MFile "===  ===  ===  ===  ===  ===  ==="
30. eval $day to env:wrap
31. eval ( $Elapsed + $Day ) mod 7 to env:Day
32. eval $day * 5 to env:space
33. echo >>T:MFile "              " first=1 len=$space noline
34. setenv daynum 1
35. eval $DiM + 1 to env:DiM
36. eval $day to env:wrap
37. setenv BreakMe $process
38. newshell from s:MonthPrint2  con:0/0/360/50/Calendar/Auto
39. wait >NIL: 20 mins
```

CCOPY

Synopsis: [EXECUTE]CCOPY<[FROM]=Source>[TO=]Destination [BUF=<buffers] [CLONE] [DATES] [NOPRO] [COM]

Template: FROM/A, TO, ALL/S, QUIET/S, BUF/K, CLONE/S, DATES/S, NOPRO/S, COM/S

Path: S:

Requires: V1.3+

See also: CCOPY Alias

Type: Script

Brief: Intelligently select COPY or COPY "" dependant on arg chain

Description

The CCOPY Alias is all very well, but you have to remember which version of COPY to use depending on the situation. To get around this, it is necessary to write a small script to make COPY intelligent. If a destination is supplied it works like AmigaDOS; if not it behaves like MS-DOS.

This script mirrors the original COPY command very closely although a few embellishments have been added – displaying the source and destination directories for instance. Also, the destination is no longer a required argument. To use this, simply type it into your favourite editor and save it in S:. Now set the "S" protection flag and it works like the real thing. It relies on an undocumented feature of AmigaDOS 1.3 in that switch (/s) arguments are supported.

Example

```
1>CD  RAM:
1>CCOPY S:Startup-sequence
```

Line-By-Line

1. Defines the argument template to match that of the existing COPY command.

2-3. Redefine bra and ket to my favourite settings.

4. Sets the default value for "BUF" to 200 (200K in disk buffers for this script. You may alter this as you see fit (or whatever fits your machine).

5. Sets the default value for the TO (destination) parameter to something silly. You could replace this with some white space or punctuation if you prefer – this value was chosen for clarity.

6. Checks if a value was sent via the TO argument. If an argument was supplied, control transfers to Step 10; otherwise it continues at Step 7.
7. Prints the first part of a progress message to keep you informed as to what's going on – note use of the NOLINE switch to keep everything on the same line. (You may like to add another command line option to suppress these messages or incorporate the existing QUIET switch here.) Assuming the example above, the script will output something like this

   ```
   Copying from: S:Startup-sequence TO:
   ```

8. Displays the current directory setting thus completing the example above – remember, CD gives the volume name – not the device name:

   ```
   Copying from: S:Startup-sequence TO Ram Disk
   ```

9. Performs the copy operation using the options specified at the command line and copying to the current directory.
10. If execution gets here from Step 9 it jumps to Step 13; otherwise it continues...
11. ...here and gives out a pretty obvious progress message based upon the command line arguments.
12. Actually does the copy proper.
13. Terminates the IF...ELSE...ENDIF construct opened at Step 6.

Listing

```
1.  .key FROM/A,TO,ALL/s,QUIET/s,BUF/K,CLONE/s,DATES/s,
    NOPRO/s,COM/s
2.  .bra {
3.  .ket }
4.  .def BUF 200
5.  .def TO NOTHING
6.  IF {TO} EQ "NOTHING"
7.   ECHO "Copying from: {FROM} TO " NOLINE
8.   CD
9.   COPY {FROM} "" {ALL/s} {QUIET/s} {CLONE/s} {DATES/s}
    {NOPRO/s} {COM/s} BUF={BUF}
10. ELSE
11.  ECHO "Copying from: {FROM} TO {TO}"
12.  COPY {FROM} {TO} {ALL/s} {QUIET/s} {CLONE/s} {DATES/s}
    {NOPRO/s} {COM/s} BUF={BUF}
13. ENDIF
```

CCOPY

Synopsis: <source file> [options]
Template: see text
Path: na
Requires: V1.3+
See also: CCOPY Script
Type: Alias
Brief: Simple version of MS-DOS COPY
Definition: ALIAS CCOPY COPY [] ""

Description:

This potboiler started life on CIX late one evening - someone wanted COPY to act like a PC. That is: if a source directory is not specified, COPY duplicates the file in the current directory. For instance:

```
1>CD RAM:
1>COPY S:SPAT
```

is not possible in AmigaDOS because COPY requires two arguments.

Either argument can be replaced with "" but this is messy. The solution therefore is to use an alias. I've called this one CCOPY - Current Copy; the name is not important. Add this to your Shell-startup script so it will be available at any time; all the normal COPY options are available too.

Chatter

Synopsis: [EXECUTE] Chatter <[NAME1=]Name1> <[NAME2]Name2>

Template: NAME1/A, NAME2/A

Path: S:

Requires: V1.3

See also: Chatty

Type: Script

Brief: Start the pipe messaging system with names

Description

Although executed from the remote terminal, CHATTER and CHATTY handle all the communication between the two machines. Unlike MS-DOS and UNIX, AmigaDOS does not support unnamed pipes where the output stream of one command can be connected to the input stream of another. Hence the MS-DOS construct:

```
TYPE READ.ME | MORE
```

is not valid in AmigaDOS and has no direct equivalent. (The bar "|" symbol is used in MS-DOS to signify a pipe.) The nearest alternative is to do the command in two steps thus:

```
1>COPY READ.ME PIPE:A
1>MORE PIPE:A
```

or, alternatively:

```
1>TYPE >PIPE:A READ.ME
1>MORE <PIPE:A
```

Note: the ability to use unnamed pipes is documented as part of the ARP 1.3 release and is claimed to work with the Shareware Shell replacement, Conman. ARP 1.3 users may want to try this out for themselves.The MORE program exhibits different behaviour depending on whether it is launched directly or as a separate process via RUN. Try this:

```
1>MORE S:SPAT
1>RUN MORE S:SPAT
```

Notice in the first instance how MORE uses the current Shell window, but when RUN it opens a window of its own. This feature may seem pointless, but it allows MORE to be used over the serial port. Moreover, it also gives rise to a variation on the CHAT theme.

The "Chat" system is designed to be launched by the remote system and initialise all the pipes. It cannot set up the CHATTO alias – this can be done in Shell-startup, by adding the following lines:

```
ALIAS ChatTo COPY * TO PIPE:[]
ALIAS ChatNow RUN EXECUTE CHATTY
```

To start the chat system, the remote operator enters (for example):

```
1>CHATNOW MARK BRUCE
```

The host terminal then receives a message (via MORE) like this:

```
Chat system opened as: Host=MARK Remote=BRUCE
```

Mark (using the host Amiga) can now start chatting to Bruce:

```
1>CHATTO BRUCE
Hello Bruce!
```

and send the message by pressing CTRL+\ as described earlier. Similarly, Bruce can chat back to Mark like this:

```
1>CHATTO MARK
Hello Mark! wonderful system, huh?
```

Line-By-Line

1. This defines the script's argument template. Two inputs are required from the user: the name of the host and remote machines. Since the /A template option has been used, both arguments must be supplied or the script will fail to run.
2. Creates a file in the temporary files assignment T: containing the startup message which will be displayed on the host terminal. The arguments surrounded with angle brackets will be replaced by the user's input. Therefore if the command line is:

   ```
   1>RUN CHATTER DAVE PAT
   ```

   the file will contain the message: Chat system opened as: Host=DAVE Remote=PAT. The filename is determined by adding the process number to "QWE". Therefore if CHATTER was running as process 3, the filename would be "QWE3".
3. This is a trick which relies on the ability of MORE to recognise when it has been RUN-launched. Normally, MORE would display T:QWE on the remote terminal, however since EXECUTE has be RUN-launched, the script is also running as a process and any commands it contains are also running asychronously. The end result is MORE pops up as a window on that the HOST machine (much to the surprise of unwary operators).

4. Not a lot of people know this, but it is quite legal to RUN-launch scripts from within scripts – even those which have been launched with RUN in the first place. That's what happens here, CHATTY is RUN-launched from CHATTER. It must be started in this way because, as you will see later, it never returns. CHATTY is passed one parameter, the name of the remote terminal. (The parameter was missing in the original listing.)
5. This label defines the start of a loop which is called endlessly. Like CHATTER, this script never finishes.
6. Copies the contents of the host pipe (if any) to a temporary file. This forces the script to pause until some data appears at the pipe and prevents the script from needlessly looping.
7. Immediately displays the contents of the temporary file. If more was used like this to display the contents of the pipe directly the script would not pause correctly.
8. Forces the script to jump back to the label defined at step 5 completing the endless loop. The result is the program waits until a message appears on the pipe, displays it, and waits for the next one.

Listing

```
1.  .key NAME1/A,NAME2/A
2.  Echo >T:qwe<$$> "Chat system opened as: Host=<NAME1>
    Remote=<NAME2>"
3.  More T:qwe<$$>
4.  Run Execute S:Chatty <NAME2>
5.  Lab Start
6.  Copy Pipe:<NAME1> T:msg<$$>
7.  More T:Msg<$$>
8.  Skip Start Back
```

CHATTY

Synopsis: [EXECUTE] Chatty [Name=]

Template: Name

Path: S:

Requires: V1.3+

See also: Chatter

Type: Script

Brief: Read piped messages from either terminal

Description

This script is never executed directly, it works as a support script that is called by CHATTY. Typically, scripts of this size can be created by the script that calls them. However, that was not thought necessary for this program.

Line-By-Line

1. Defines the argument template for the script. Although the argument would normally be required, it is not necessary to do that here since the correct syntax is assured by the calling script. The argument received by CHATTER is the name allocated to the remote terminal.
2. Defines a label which will be jumped to when the script loops.
3. Waits for data to be sent to the pipe on the remote terminal and prints it. Like COPY, TYPE waits for information to appear on the pipe before doing anything.
4. Loops the script back to Step 1, causing it to execute again. This script never stops, but because it's attached to an internal Shell (via RUN) it does not affect the machine's operation.

Listing

```
1.  .key NAME
2.  Lab start
3.  Type pipe:<NAME>
4.  Skip start back
```

Clock 1.3

Synopsis: Run from Workbench

Template: none

Path: ...

Requires: V1.3-1.3.3

See also: Clock 2, Clock 3

Type: Script

Brief: Simple AmigaDOS clock

Description

The idea of having a real-time clock from AmigaDOS must seem a little preposterous: especially if you consider the same thing is already supplied with Workbench. The difference between this one and the one you get with Workbench is size: this clock program is less than 512 bytes long and fits easily on your Workbench disk even if you're short of space! This version should be run from Workbench via IconX.

Line-By-Line

1. Sets a dummy key for IconX.
2-6. Makes some commands resident for the script. This is necessary or the clock will spend most of its time loading commands from disk.
7. Sends some special control codes to the console to switch the cursor off (*e[0 p) and position the cursor at the start of the line (*e[;0H). Note: the punctuation is necessary!
8. Marks the start of the endless clock loop.
9. Displays the current date and time.
10. Re-positions the cursor. This stops the clock wrapping on a line.
11. Waits for a second. This controls the speed of update – you can set this wait for a longer period if you prefer: say five seconds.
12. Re-starts the loop again.
13. Is WX information. Not used by this script.

```
1.  .key dummy
2.  resident c:wait
3.  resident c:date
```

```
4.  resident c:echo
5.  resident c:lab
6.  resident c:skip
7.  echo "*e[0 p*e[;0H"
8.  lab start
9.  date
10. echo "*e[;0H" noline
11. wait 1 secs
12. skip start back
13. ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory_Gauge
```

Clock 2

Synopsis: Run from Workbench

Template: none

Path: ...

Requires: V2+

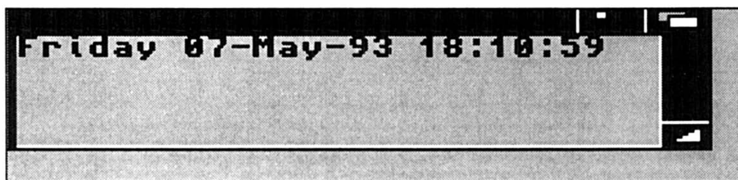
See also: Clock 1.3, Clock 2

Type: Script

Brief: Simple AmigaDOS clock

Description

This clock program is less than 512 bytes long and fits easily on your Workbench disk even if you're short of space! This one is written for AmigaDOS 2+ and should be run from Workbench via IconX or placed in WBStartup. The CON: options set for IconX determine how and where the clock appears. The BACKDROP option should only be used if the Workbench is normally operated as a window: with the BACKDROP option OFF.



Clock 2

Line-By-Line

1-2. Make WAIT and DATE resident. These commands are ADDed to the resident list and never removed, so this script should only be run once from WBStartup. The ADD option can be removed but this can cause problems if some other program has control of the resident list.

3. Positions and switch the cursor off.

4. Marks the start of the clock loop.

5. Displays the current date and time and positions the cursor back at the start of the line; thus avoiding line wrap and scrolling.

6. Waits for a second. This controls the update time of the loop (and the clock) and may be increased if preferred.

7. Re-starts the loop ready for the next display run.
8. Is some information for WX to use. You don't need this for the script if it's run from Workbench.

Listing

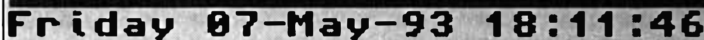
```
1.  resident c:wait add
2.  resident c:date add
3.  echo "*e[0 p*e[;0H" noline
4.  lab start
5.  echo "`date`*e[;0H" noline
6.  wait 1 secs
7.  skip start back
8.  ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory
    Gauge/SMART/NOSIZE
```

Clock 3

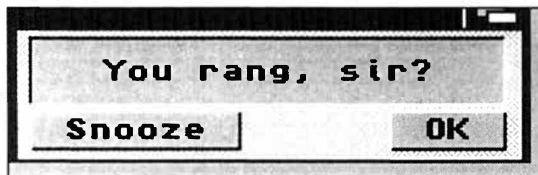
Synopsis:	[EXECUTE] Clock3 [[Ticks=]Seconds]
Template:	ticks
Path:	S:
Requires:	V2+
See also:	Clock 1.3, Clock 2
Type:	Script
Brief:	imple digital clock with auto window

Description

This script was created just to prove a big point in a short space. That is: you don't have to use Workbench if you want to have a program working in its own window. This script creates a completely new script and executes it using NEWSHELL. In this way it creates its own window and starts the program as a process!



Clock 3



Clock Alarmed

Typically you can use this script without arguments, but you can supply the number of "ticks": that is the number of seconds it leaves other processes before re-executing. Examples:

```
1>Clock3
1>Clock3 ticks=5
```

Line-By-Line

1-3. Define a simple, standard header.

4. Sets the default for seconds to wait. One second is used here since this is a simple script, but you could use a longer time and send a shorter delay from the Shell.

5-6. Make the required external commands resident.

7. Creates the program file in T: as "xlclock#". (Remember that # is the current Shell process number.) This line will appear

after translation as:

```
echo "*e[0 p *e[;0H" noline
```

8. Adds the second line to "Xclock#". This is simply a label.

9. Adds the next line to the "Xclock#" file. Note that this is only the first half of the third line! The reason is we don't want the "`" couplet expanding the date just yet! The NOLINE switch for ECHO ensures the line is not fully terminated just yet. At this stage the second line looks like this:

```
echo *"`date
```

10. Writes the second part of the third line completing it thus:

```
echo *"`date`*e[;0H"
```

11. Adds the third program line: inserting the wait limit directly into the program. If no arguments were supplied, ticks = 1, so the line looks like this:

```
wait 1 secs
```

12. Completes the program by adding the loop back.

13. Starts the clock program by executing the xclock# script with NEWSHELL. This isn't exactly the same as using IconX but it will suffice, thank you. The values shown here place the clock on the Workbench screen. You should remove the NOBORDER and BACKDROP switches if you want to be able to move the clock around.

14. This information is only used by WX – which is really not suitable for this script.

Listing

```
1.  .key ticks
2.  .bra {
3.  .ket }
4.  .def ticks 1
5.  resident c:wait
6.  resident c:date
7.  echo  >T:xclock{$$} "echo *"**e[0 p **e[;0H*" noline"
8.  echo >>T:xclock{$$} "lab start" noline
9.  echo >>T:xclock{$$} "echo *"`date" noline
10. echo >>T:xclock{$$} "`**e[;0H*" noline"
11. echo >>T:xclock{$$} "wait {ticks} secs"
12. echo >>T:xclock{$$} "skip start back"
13. newshell from t:xclock{$$}
    window=con:0/5/240/30/Clock/NOBORDER/BACKDROP/SMART
14. ;WX:WINDOW=WINDOW=con:0/0/240/30/CLI_Clock
```

Clock

Synopsis: Started from Workbench

Template: none

Path: ...

Requires: V2.0+

See also: ...

Type: Script

Brief: Simple digital AmigaDOS clock

Description

They don't get much simpler than this one. This script forms the basis of many of the clock scripts featured here without to0 many clever tricks. The great thing about this one is it's short. It must be run from Workbench via IconX to ensure it runs in its own window though.

Line-By-Line

1. Gives IconX something to chew on. It isn't absolutely necessary for scripts as simple as this one.

2-3. Makes some essential commands resident.

4. Switches the cursor off.

5. Marks the start of a loop.

6. Displays the current date and time.

7. Moves the cursor up a line.

8. Waits for a second...

9 ...and starts the whole thing again!

Listing

```
1.  .key dummy
2.  resident c:wait
3.  resident c:date
4.  echo "*e[0 p"
5.  lab start
6.  date
7.  echo "*e[A" noline
8.  wait 1 secs
9.  skip start back
```

AddData

Synopsis: [EXECUTE] AddData [a1...ac] [Data=<data>]

Template: a1,a2,a3,a4,a5,a6,a7,a8,a9,aa,ab,ac,data/k

Path: S:

Requires: V1.3+

See also: Sortdata, DataBase, PrintData etc

Type: Script

Brief: The add module for the Database

Description

You can add and delete records using the ED command. While this is fine for the most part, it is a little prone to error. This is the AddData module which will allow you to add one or more records directly from the command prompt. AddData works rather like the FindData module in that you can execute the command followed by a data string:

```
Command: A Mark Smiddy, Mastering AmigaDOS 3, BSB
Add another y/N?
```

or just execute it on its own, like this.

```
Command: A
Add another y/N? Y
Mark Smiddy, Mastering AmigaDOS 3, BSB
Add another y/N
```

Note in the second case, AddData asks if you want to add something before you actually do; this is quite normal.

Line-By-Line

1. Defines a long key with lots of parameters – as you may recall, this allows for long interactive command lines. In this case, it allows you to enter 12 "words" without having to resort to quotes.
2. Collects the contents of <a1>...<ac> into a variable called data.
3. Checks to see if any data has been entered – it can come from the initial command line or interactively from this segment. If data has been supplied control jumps to Step 5; otherwise it continues at Step 4.

4. Transfers control immediately to the end of the script (Step 20 actually). Since no data has been supplied, the script must be executed again in order to gather some.
5. Closes the IF...ENDIF construct opened at Step 3.
6. Tests for the presence of a temporary file in T: (the temporary files assignment). It's called "Temp" in this example, but any name would do – provided you stick to the same one throughout the script. If the file exists, control passes to Step 7; otherwise it jumps to Step 8.
7. The new record (the contents of the variable "Data") are appended (added) to the temporary file here. By appending each new entry, the file contains the new records – before they are added to the main database. It is important to note, the temporary file is created by this module and removed by the main Database script.
8. If control reaches here from Step 7 it branches to Step 10; otherwise it continues at Step 9.
9. Creates the temporary data file "T:temp" and adds the current record to it. T:Temp is only valid while this program is running, see the description of Step 7 for more details.
10. Closes the IF...ELSE...ENDIF construct opened at Step 6.
11. Prints the prompt "Add another y/N" and pauses for user interaction. (It waits for you to enter something.) Note the ASK command sets the WARN flag if you enter Y and clears it otherwise.
12. Tests for the WARN condition. If you enter, Y at Step 8, control resumes at Step 13; anything else causes a branch to Step 14.
13. Transfers control to Step 20 where the script will start again. You may notice, this label is also used by Step 4 to achieve the same effect.
14. Control can only arrive here from Step 12 (Step 13 is an unconditional branch) and it continues directly at Step 15 because Y was *not* entered.
15. Displays a short message (split over two lines by the use of *n) just to keep the user informed.
16. Adds the new records to the start of the data file using the JOIN command. The AS keyword is used here to keep the syntax clear. Note the new file is also stored in T: because it isn't legal to write to a file you are reading from. You may wonder at this stage why the records are kept in a temporary file. The reason is twofold. First, since T: is in RAM, a temporary list is faster to update than it would be if the write went directly to disk every time. Second, this also offers the

chance to insert a "Get out" clause; such as "Are you sure?". This is demonstrated in the DelBlock module described below but not incorporated here.

17. Replaces the existing database with the one just created at Step 16. This command can fail if the disk is full and in this case, you should copy the complete database from RAM to a disk with enough room. As an exercise you might like to devise a fix for this eventuality.
18. The module completes here and calls the main program again.
19. Closes the IF...ELSE...ENDIF construct opened at Step 12.
20. This marks the restart point called at Steps 4 and 13.
21. Displays the entry prompt without a line feed – to give the illusion of an interactive prompt.
22. Calls the AddData module recursively in its interactive mode. Redirection to NIL: (>NIL) is used to suppress the messy command line.

Listing

```
1.  .key a1,a2,a3,a4,a5,a6,a7,a8,a9,aa,ab,ac,data/k
2.  .def data "<a1> <a2> <a3> <a4> <a5> <a6> <a7> <a8> <a9>
    <aa> <ab> <ac>"
3.  if "<data>" EQ ""
4.   skip AddOne
5.  endif
6.  if exists T:temp
7.   echo >>T:temp "<data>"
8.  else
9.   echo > T:temp "<data>"
10. endif
11. ask "*nAdd another y/N"
12. if warn
13.  skip AddOne
14. else
15.   echo "Adding new records*nPlease wait..."
16.   join T:Temp S:Data AS T:tempdata
17.   copy T:TempData S:Data
18.   execute S:database
19. endif
20. LAB AddOne
21. echo "data: " noline
22. execute >NIL: s:AddData ?
```

DelBlock

Synopsis:	[EXECUTE] DelBlock [Start=] [Number=]
Template:	start,number
Path:	S:
Requires:	V1.3+
See also:	Database, Findata, AddData etc.
Type:	Script
Brief:	The delete data module for the Database

Description

The DelBlock module is used to remove specific records from the database and can be accessed from the command line in one of two ways. For instance, to delete record number three you would enter:

```
Command: D 3
3   Fred Bloggs, 1 The Marketplace, Newton Abbott
Delete records?
```

However, you can delete a range of records by specifying the start and ending record numbers, viz:

```
Command: D 2 4
2   Amiga Shopper, Future Publishing, BATH, Avon
3   Fred Bloggs, 1 The Marketplace, Newton Abbott
4   Dave Smith, Behind the Bike Sheds.
Delete records?
```

If you do not supply either value, DelBlock will prompt you for them automatically.

Line-By-Line

1. This module only takes two parameters: START, the record number to be deleted; and NUMBER and optional parameter which will be the last record to be deleted.
2. This defaults the NUMBER variable to 1 if no value is supplied.
3. Similarly, if a starting record number is not supplied, this variable is initialised to the string SKIP.

4-5. Redefine < and > as { and }.

6. Test the start variable to see if some number was supplied. (A test is made for a string supplied as a default value at Step 3

to make the meaning clearer.) If the test passes, control continues at Step 7; otherwise it jumps to Step 8.

7. Control arrives here if no values were supplied and is sent directly to the re-start code, beginning at Step 39.
8. Close the IF...ENDIF construct opened at 6. (Control arrives here if a START value was supplied.)
9. Calculates how many records (lines actually) are going to be removed from the database and stores the result in the environmental variable "ThisMany". Note a LFORMAT string is used to suppress the LF character.
10. Adds 1 to the value ThisMany and stores it in a temporary variable. Note that interactive mode must be used for this example because one of the values is being retrieved from a file. This must be done to retain downward compatibility with AmigaDOS 1.3
11. Copies the temporary variable back to the proper variable.
12. Test if the value "ThisMany" is not less than or equal to zero. ("VAL NOT GT" is akin to BASIC's <=.) If it is less than 1, control continues at Step 13. Note: Interactive mode is used again here because the value is being retrieved from a file.
13. Negative or zero values are not allowed for this value, so this line resets ThisMany to its lowest possible value.
14. Closes the IF...ENDIF construct opened at Step 12.
15. Raises the fail level slightly to prevent minor complaints from EDIT stopping the script in its tracks.
16. This is where it starts to get a little hirsute - so to make things a little simpler, let's assume some values. Set START as 5 and THISMANY as 4. Using these values, this line creates a file containing:

    ```
    5n;p;
    ```

    and saves that as DelMac1
17. Similarly, this creates DelMac2 which looks like this:

    ```
    (?;n;)
    STOP
    ```

18. Now the module joins the two macro segments to the environmental variable and the macro takes shape like this:

    ```
    5n;p;
    4(?;n;)
    STOP
    ```

Translation

5n	Go down five lines
p	Move back one line
4(?;n;)	Display this line then move to the next one (four times)
STOP	Quit and return to the caller.

This example also illustrates why it was necessary to suppress the line feed in "ThisMany". Otherwise, the macro would read:

```
5n;p;
4
(?;n;)
STOP
```

which is a subtle error and not easy to trace; pretty hairy stuff too, as I said.

19. Uses the EDIT macro on the database and creates a temporary file (T:DelRec) which contains a list of the records about to be deleted. The story does not end there though...
20. ...because, if the record(s) were not found, EDIT returns with an ERROR condition – remember Step 15. This is tested for here, and if found control resumes at Step 21; otherwise it branches to Step 23.
21. Displays the error message...
22. ...and jumps to the code at Step 37.
23. Control only gets here from Step 20 when the requested records are found. It continues at Step 24...
24. ...which changes the text colour to the highlight...
25. ...displays the list of records marked for the chop...
26. ...and switches the highlight off again.
27. Closes the IF...ELSE...ENDIF opened at Step 20.
28. Pauses the script and gives the user a last chance to decide whether or not to delete the records.
29. Tests for the WARN condition – returned by ASK if "Y" is entered. If WARN is found control continues at Step 30; otherwise it jumps to Step 36.
30. Displays an information message. The start and end numbers are filled in when the script runs of course.
31. We've already met something like this before – at Step 16. This creates the same edit macro once more (although the original would probably do just as well). Assuming the previous values, DelMac1 reads:

```
5n;p;
```

32. DelMac2 is simpler though, this just adds a single letter.

```
d
```

33. Creates a new version of DelMac which now looks like this:

```
5n;p;
4d
```

Which finds the first line (records) we are interested in and deletes that and the next three...

34. ...here as temporary file...
35. ...which replaces the original database here.
36. Closes the IF...ENDIF construct opened at Step 29.
37. Marks the skip point which is used by the error handler. Note, when EXECUTE is called, the failure level is reset back to 10 (crash on ERROR). This line is ignored if execution reaches here normally from Step 36.
38. Calls the main database module.
39. Marks the skip point called when the user has not entered any values – see Steps 6-7.
40. Displays the single line prompt...
41. ...and executes the module in interactive mode so the values can be retrieved correctly.

Listing

```
1.  .key start,number
2.  .def number 1
3.  .def start SKIP
4.  .bra {
5.  .ket }
6.  if "{start}" EQ "SKIP"
7.   skip restart
8.  endif
9.  eval {number} - {start} to ENV:ThisMany lformat "%n"
10. eval >NIL: <env:ThisMany op=+ value2=1 to T:Tmp lformat
    "%n" ?
11. copy T:Tmp to ENV:ThisMany
12. if >NIL: <ENV:ThisMany VAL NOT GT 1 ?
13.  setenv ThisMany 1
```

```
14. endif
15. failat 11
16. echo >T:DelMac1 "{start}n;p;"
17. echo >T:DelMac2 "(?;n;)*nSTOP"
18. join T:DelMac1 ENV:ThisMany T:DelMac2 AS T:DelMac
19. edit s:Data with T:DelMac ver=T:DelRec
20. if error
21.  echo "Record(s) not found"
22.  SKIP ReRun
23. else
24.  echo "*e[33m"
25.  type T:DelRec
26.  echo "*e[31m"
27. endif
28. Ask "*nDelete record(s) y/N?"
29. if warn
30.  echo "Removing Records from: {start} to {number}
    *nPlease wait..."
31.  echo >T:DelMac1 "{start}n;p;"
32.  echo >T:DelMac2 "d"
33.  join T:DelMac1 ENV:ThisMany T:DelMac2 AS T:DelMac
34.  edit s:data to T:DelData with T:DelMac
35.  copy T:DelData to S:Data
36. endif
37. LAB ReRun
38. execute s:database
39. LAB restart
40. echo "Delete record #: " noline
41. execute >NIL: s:DelBlock ?
```

DELF

Synopsis: DELF <file|pattern>

Template: As DELETE

Path: na

Requires: V2+

See also: DELQ

Type: Alias

Brief: Short form for DELETE

Definition: ALIAS DEL DELETE >NIL: [] FORCE

Description:

This alias deletes a list of files like DELETE, but does not report anything back to the console and will also delete protected files. Use this with extreme caution! Example:

```
1>DELF #? ALL
```

DEL

Synopsis:	DEL <name or pattern>
Template:	na
Path:	na
Requires:	V1.3+
See also:	DELQ, DELF
Type:	Alias
Brief:	Short name for DELETE
Definition:	ALIAS DEL DELETE

Description:

This alias is not included for padding (as it might seem to be) it has a very serious us. DEL is the MS-DOS command for DELETE and MS-DOS users will feel much more at home with AmigaDOS if the command works like this.

DOCTOR

Synopsis: DOCTOR

Requires: V1.3

See also: DISKDOC script

Type: Alias

Brief: Open a new CLI window with DiskDoctor

Definition: NEWCLI WINDOW CON:0/3/500/100/DiskDoc FROM S:DiskDoc

Description:

This command is listed here for the sake of completeness. It does nothing on its own apart from launching the DISKDOC script with a window. The clever bit is this:

```
NEWCLI WINDOW CON:0/3/500/100/DiskDoc FROM S:DiskDoc
```

This command performs several functions at once.

• It opens a new CLI independent of the current Shell so DISKDOCTOR can be run from here.

• It defines a new window for the CLI. In practice this is tucked away in the top left of the screen with enough room for most messages to be displayed. The idea is to stop it getting in the way – but you can position it to your own liking. For the sake of beginners only, here's a brief explanation of what it means:

```
WINDOW  Device:X/Y/Width/Height/Name

Device: CON: or NEWCON:

X:    X position. Range 0 to 639 (Topaz 80)

Y:    Y position. Range 0 to 255 (PAL) or 0 to 199 (NTSC)

Width:  Width of window in pixels - practical range 50 to 639

Height: Height of window in pixels - practical range 50 to
     255
```

• It starts DISKDOCTOR. The command is run from the script explained under DISKDOC using the FROM argument.

DRS

Synopsis: DRS
Template: ...
Path: ...
Requires: V2.0+
See also: VLS, DVS
Type: Alias
Brief: Check an assignment without removing it
Definition: ALIAS DRS ASSIGN DIRS

Description:

DRS is a quick way to list all the current directory assignments. This alias uses ASSIGN in such a way that the device and volumes lists are suppressed.

DVS

Synopsis:	DVS
Template:	na
Path:	na
Requires:	V2+
See also:	VLS, DLS
Type:	Alias
Brief:	Check an assignment without removing it
Definition:	ALIAS DVS ASSIGN DEVS

Description:

DVS displays the names of the current devices attached to the system using ASSIGN. The volumes and directory listings are suppressed.

FindData

Synopsis: [EXECUTE] FindData [a1]...[ad] [data=search string]

Template: a1,a2,a3,a4,a5,a6,a7,a8,a9,aa,ab,ac,ad,data/k

Path: S:

Requires: V1.3+

See also: Database

Type: Script

Brief: Find data module for the Database suite

Line-By-Line

1. This key can collect up to 13 items of data, although it will usually only pick one or two. The first item found is dumped in <a1>, the next in <a2> and so on.
2. Collects the data together in a single variable – <data>.
3. Tests for the presence of an argument string. You may wonder why the collected variable, <data> wasn't used here and true enough the reason is not immediately apparent. In fact, <data> is a collection of variables separated by spaces. If no data was supplied, <data> will still contain spaces, thus fooling IF into thinking some data has in fact been supplied. This is not true of <a1> because all the excess spaces are removed by EXECUTE's command line parser. Or in other words, don't worry, it just works that way.
4. Control reaches here if a blank command line was specified and immediately jumps to Step 19.
5. Closes the IF...ENDIF construct opened at Step 3. Control only reaches here when a command line search string was supplied.
6. This is a bit of extra redundancy. It prevents the script from crashing if the data file is missing for some reason. In this case control jumps to Step 11; ordinarily it proceeds to Step 7.
7. Using the search command, this attempts to locate the search string within the data file. (Note: the search string is surrounded by quotes to prevent a blank string confusing the parser.) If matching data is found, it is displayed with a corresponding record number and the WARN flag is cleared. If no matches are found, the WARN flag is set, which is tested...
8. ...here. If the search string was found control jumps to Step 10. Otherwise it continues at Step 9 and...

9. ...prints a short message to the effect the search string could not be found. The search string is enclosed in escaped quotes to show the exact contents of the search.
10. Closes the IF...ENDIF construct opened at Step 8.
11. Closes the IF...ENDIF construct opened at Step 6.
12. Is the jump point used when an empty command string is found at Step 3. If control reaches here from Step 11, the command is ignored.
13. Inserts a blank line (*n) and waits for the user to enter Y or press Return. Entering Y <Return> sets the WARN flag; it is cleared otherwise.
14. If the user entered "Y", control resumes at Step 15, otherwise it jumps to Step 16.
15. The user wants to execute another search, so control is passed to Step 19.
16. If control gets here from Step 15, it branches to Step 18; otherwise it continues at Step 17.
17. Control only reaches here if the user did not enter Y at step 14. In other words, they want to return back to the main program.
18. Closes the IF...ELSE...ENDIF construct opened at Step 14.
19. Marks the jump from Step 15.
20. Displays the command prompt. This is only displayed when a search string was not specified OR a successive search has been requested.
21. Calls the FindData script again recursively and places it in interactive mode. The command line argument string is sunk to NIL and not displayed.

Listing

```
1.  .key a1,a2,a3,a4,a5,a6,a7,a8,a9,aa,ab,ac,ad,data/k
2.  .def data "<a1> <a2> <a3> <a4> <a5> <a6> <a7> <a8> <a9>
    <aa> <ab> <ac> <ad>"
3.  if "<a1>" EQ ""
4.  skip FindOne
5.  endif
6.  if exists S:Data
7.  search S:Data <data>
8.   if warn
9.   echo "*"<data>*" not found"
10.  endif
```

```
11. endif
12. LAB again
13. ask "*nSearch again y/N"
14. if warn
15. skip FindOne
16. else
17. execute S:database
18. endif
19. LAB FindOne
20. echo "search string: " noline
21. execute >NIL: s:FindData ?
```

Database

Synopsis: [EXECUTE] Database

Template:

Path: S:

Requires: V

See also: Findata, AddData, SortData etc

Type: Script

Brief: The main menu module for the Database

Description

In the last few years digital diaries have sold in ever-increasing numbers (the idea for this script hit me late one night when my diary's battery expired). Since you already have a computer you may as well use it, but "real" database programs are rarely cheap and usually too powerful for simple applications – such as a telephone book. The flat-file database described here is crude by commercial standards but boasts the following features:

- Completely menu driven
- Sort records on any record column
- Search and display a record on any sub-string
- Database can be edited directly with any text editor (or ED)
- Delete any record group
- View any group of records
- View entire database
- Outputs to printer
- Compatible with AmigaDOS 2
- Imports and exports to and from Superbase

Quite impressive for a program written entirely in the machine's DOS batch language I think you will agree. But even if you have a database, this application will show you how to control many parts of AmigaDOS in previously unexplored avenues.

The entire program (actually it's a series of modules) is much too large to explain in one chunk; instead, I've divided it over the several scripts as each module is added. All the scripts are compatible with AmigaDOS 2 and probably AmigaDOS 3, but they do not take advantage of extra facilities in the new systems: such as the /f find argument

The Menu System

The main part of AmigaDOS Database is its front-end menu system. This allows anyone with little or no knowledge of AmigaDOS to operate the system without fuss. All commands can be operated by selecting the first (highlighted) letter and pressing return "P" for print: "S" for sort and so on. However, many commands also take parameters. View for instance takes one or two parameters as the starting and finishing numbers. These can be supplied as part of the command line, for instance:

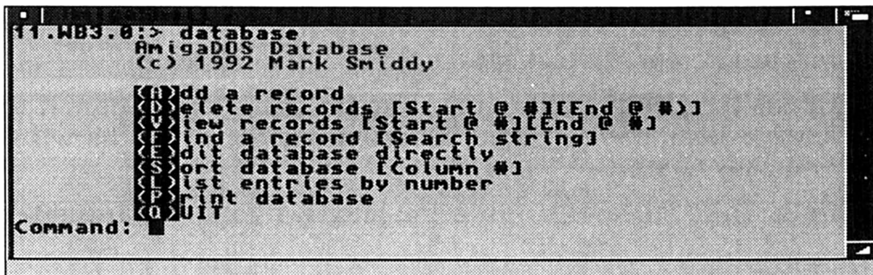
```
Command: V 2 5
```

displays records 2 to 5 inclusive. However if a parameter is not supplied the module will prompt for it automatically. For instance:

```
Command V

View record #:
```

The task of deciding which parameters are required is handled partly by the main menu module and partly by the view module.



Database Menu

Line-By-Line

1. This is the key to the main menu, although under normal circumstances you won't even see it. Database is normally started without parameters so it displays its menu. Experienced users have the option of giving a command directly like this:

   ```
   1>DATABASE P ; print database now

   -or-

   1>DATABASE S 20 ; Sort database from column 20
   ```

   The key breaks down into three sections.

   ```
   Command: The single letter command.

   d1…d9: Nine data elements (or arguments)

   Option:  A reserved variable
   ```

2-3. Set the BRA and KET characters to { and } respectively (because I happen to like them like that).

4. Sets the Option variable to the contents of d1...d9. This is reserved for the FindData and AddData modules described later.

5-7. This is provided as a time saver. Under normal circumstances when Database is launched, the Command variable will be empty, that is equal to "" and this causes the program to skip immediately to the menu starting at Step 41.

8. If the command variable equals "A" (add) control continues at Step 9, otherwise it continues at Step 11.

9. Control only reaches here if the Add selection has been made. This line deletes a temporary file created by the AddData module. Re-direction to NIL: is used to suppress the error message when the file does not exist. (This could be tested with IF EXISTS... but that's just overkill.)

10. Calls the AddData module directly passing any automatic data entered by the user. (Automatic data is collected in the Option variable at Step 4).

11. Marks the end of the AddData control block. Control only reaches here if the test at Step 8 was not successful.

12. Opens the DelData block and tests for the delete option. If the key has been pressed, control resumes at Step 13; otherwise it branches to Step 14.

13. Calls the DelData module passing up to two parameters collected in D1 and D2.

14. Control only reaches here if the test at Step 12 was unsuccessful and continues directly at Step 15.

15-17. Open the ViewData block. This works just like DelData described above.

18-20. Open the FindData block. These work like AddData, passing any automatic data in the Option variable.

21-23. Open the ED block. The ED screen editor is opened. Note that ED is called directly within the script unlike most commands which execute a new script.

24-26. Open the SortData module. One optional automatic parameter is passed in D1

27. Open the List by number module if "L" was passed as a command parameter.

28. Copies the entire database to a file in the Ram disk, but adds line numbers to each record. This is a good way to view and edit blank records.

29. Displays the temporary data file a screen at a time.

30. Terminates the List block. Control will continue from here when MORE exits.

31-33. Operate the PrintData block. Control does not resume at Step 33 unless the test at Step 31 is unsuccessful.

34-40. Looks after the Quit section, terminating the program nicely. The block 35-39 just stops users from accidentally exiting. If they enter N at the prompt, execution falls off the end of the program and the script runs itself again.

41-51. Display the menu. Several escape sequences are used here, viz:

```
*e[I   Tabulate
*e[7m  Inverse video
*e[0m  Normal video
*e[33m Foreground orange (blue in AmigaDOS 2)
*e[31m Foreground white
```

52. Displays the command prompt. Note the use of the NOLINE switch to suppress the automatic line feed character.

53. More trickery. EXECUTE is used to call the Database module and display its command line in interactive mode; which is also suppressed by the re-direction to NIL:. The result is you can enter commands at an invisible prompt!

Listing

```
1.  .key Command,d1,d2,d3,d4,d5,d6,d7,d8,d9,option/k
2.  .bra {
3.  .ket }
4.  .def option "{d1} {d2} {d3} {d4} {d5} {d6} {d7} {d8}
    {d9}"
5.  if "{command}" EQ ""
6.   SKIP menu
7.  Endif
8.  if "{command}" EQ "A"
9.   delete >NIL: T:Temp
10.  execute S:AddData {option}
11. endif
12. if "{command}" EQ "D"
13.  execute S:DelBlock {d1} {d2}
14. endif
15. if "{command}" EQ "V"
16.  execute S:ViewBlock {d1} {d2}
17. endif
18. if "{command}" EQ "F"
```

```
19.  execute s:FindData {option}
20. endif
21. if "{command}" EQ "E"
22.  ED s:Data
23. endif
24. if "{command}" EQ "S"
25.  execute S:SortData {d1}
26. endif
27. if "{command}" EQ "L"
28.  TYPE >RAM:data{$$} S:DATA OPT N
29.  MORE RAM:data{$$}
30. endif
31. if "{command}" EQ "P"
32.  execute S:PrintData
33. endif
34. if "{command}" EQ "Q"
35.  ASK "Are you sure y/N?"
36.  if warn
37.   echo "Thanks for using AmigaDOS DataBase*nPlease come
    again..."
38.   QUIT
39.  endif
40. endif
41. LAB Menu
42. echo "*e[33mAmigaDOS Database*n*e[I(c) 1992 Mark
    Smiddy*e[31m*n"
43. echo "*e[7m*e[I(A)*e[Omdd a record"
44. echo "*e[7m*e[I(D)*e[Omelete records [Start @ #][End @
    #)]"
45. echo "*e[7m*e[I(V)*e[Omiew records [Start @ #][End @ #]"
46. echo "*e[7m*e[I(F)*e[Omind a record [Search string]"
47. echo "*e[7m*e[I(E)*e[Omdit database directly"
48. echo "*e[7m*e[I(S)*e[Omort database [Column #]"
49. echo "*e[7m*e[I(L)*e[Omist entries by number"
50. echo "*e[7m*e[I(P)*e[Omrint database"
51. echo "*e[7m*e[I(Q)*e[OmUIT"
52. echo "Command: " noline
53. execute >NIL: s:database ?
```

PrintData

Synopsis: [EXECUTE] PrintData

Template: None

Path: S:

Requires: V1.3+

See also: Database, FindData, AddData, SortData etc

Type: Script

Brief: The printing module for the Database

Description

This script supports the main Database engine described under DATABASE and provides simple print facilities. It should not be executed directly.

Line-by-line

1. Pauses the script and waits for you to press return. The WARN/OK return from ASK is not used at this point, but the command provides a useful pause.
2. Gives an opportunity to set the Printer preferences. If "Y" is entered before pressing <Return> a WARN condition is returned to AmigaDOS.
3. Checks the WARN condition set at Step 2. If the user wants to set the printer preferences, the WARN condition is active and control continues at Step 4; otherwise it jumps to Step 5.
4. Calls the System preferences. Note that if you are using AmigaDOS 2 or above, you should set the path to activate the PRINTER tool: usually "Prefs/Printer".
5. Terminates the IF...ENDIF construct opened at Step 3.
6. Creates a new file in the Ram disk starting with an opening line as follows: "AmigaDOS Database on:". Note use of the NOLINE switch to prevent ECHO sending a newline character. The filename created depends on the current Shell number: appended automatically by the use of "{$$}".
7. Adds the current date and time to the headline string in the "Title" file.
8. Creates a new file by joining the contents of the header string to the data file and copies the whole lot to a new file in RAM, "Printfile#".
9. Raises the default failure code to 11. This is support for Step 10 below.

10. Copies information to the printer device (PRT:) and prints it using the default settings. Note that PRT: is used in preference to PAR: to ensure the correct printer is used. This also ensures that the correct margins and other preferred modes are used. If the command fails because the printer is busy (or not connected) it returns an ERROR condition and error message. The message is absorbed to NIL: and never appears and the script is not halted because the failure level has been raised to 11.
11. Tests if the COPY command (in effect, print) failed. If this is the case, control continues at Step 12. If everything worked according to plan, control jumps to Step 13 – lucky for us.
12. Prints a helpful message. This technique (even though it's stating the obvious) is preferable to just ignoring the fact, or simply crashing the script.
13. Terminates the IF...ENDIF construct opened at Step 11.
14. Re-sets the failure level back to its default, 10.
15. Re-starts the DATABASE main menu program.

Listing

```
1.  ASK "Position paper, ready printer and press Return"
2.  ASK "Do you want to check/alter your printer setup?"
3.  if warn
4.   SYS:Prefs/preferences printer ; AmigaDOS 2 use
    Prefs/Printer
5.  endif
6.  ECHO >RAM:title{$$} "AmigaDOS Database on:" noline
7.  DATE >>RAM:title{$$}
8.  JOIN RAM:title{$$} S:Data AS RAM:Printfile{$$}
9.  FAILAT 11
10. COPY >NIL: RAM:PrintFile{$$} to PRT:
11. if error
12.   echo "Your printer is not responding!*nPlease check
    *"On-Line*" is on; paper loaded; and cable connected"
13. endif
14. FAILAT 10
15. Execute S:DataBase
```

SortData

Synopsis:	[EXECUTE] SortData [Col = Column]
Template:	Col
Path:	S:
Requires:	V1.3+
See also:	Database, AddData, FindData etc
Type:	Script
Brief:	The sort module for the Database

Description

If this database were a commercial application (which it isn't) you would expect to find indexes removing the need to sort data. However, since this is a database writen purely in AmigaDOS the need to sort data may arise, and therefore, that's what this module is all about.

Line-By-Line

1. Defines the key for this script. Only one argument is required – the start column for the sort. This is provided as an option: those of you setting up fixed width fields will be able to sort on a field using this.

2-3. Re-define bra and ket to { and }.

4. Sets a default value of 1 to the starting column number. This is not absolutely necessary, but provided for example.

5. Displays the progress message – this is necessary to show something is going on when the module is executed from the main menu.

6. Raises the failure level above ERROR (10) in anticipation of the next command.

7. Temporarily raises the current stack to 16000 bytes. Re-direction to NIL: is used to suppress any messages. There are a couple of things you might want to consider here:

   - The stack size determines how large a file can be sorted (the sort function is recursive and requires a lot of stack space). Stack overflow will result in the machine vanishing into the land of the Guru and this is unpleasant, although it will not affect the database's security.
   - The amount of stack given should be OK on 512K or 1M machines, if you find the command fails (see below) reduce the amount or, if you need more stack, increase it.

- It is possible to add another option to the command line here: STACK/K. You might want to do this as an experiment and optionally define the stack space when the sort option is selected from the main menu.

8. If the command at Step 7 fails because of low memory ("Insufficient free store" in AmigaSpeke) the ERROR flag is returned and control continues at Step 9, otherwise it jumps to Step 10. (This would normally stop the script in its tracks, but since the FailAt level has been increased to 11 this command can execute.)
9. Lets you know what's happened and why the sort operation cannot progress.
10. If execution reaches here from Step 9 it jumps to Step 13; otherwise it continues below.
11. The sort is carried out by AmigaDOS using the options defined by the user. The sorted file is sent to data{$$} in the Ram disk. This could, just as easily, have been T:data{$$}.
12. Assuming there were no cock-ups in the sort, the sorted file is copied from RAM to disk. (You may wish to add some extra safety measures here – such as viewing the sorted file with MORE before committing yourself).
13. Closes the IF...ELSE...ENDIF construct opened at Step 8.
14. Resets the FailAt level back to its default setting of 10.
15. Resets the stack back to 4000 bytes. (You may want to change this if you usually work with a larger stack.)
16. Calls the main database module.

Listing

```
1.  .key Col
2.  .bra {
3.  .ket }
4.  .def col 1
5.  ECHO "Sorting*nPlease wait..."
6.  FAILAT 11
7.  STACK >NIL: 16000
8.  if error
9.   echo "Out of memory...*nCan't sort, sorry"
10. else
11.  SORT S:data RAM:data{$$} COLSTART={col}
12.  COPY RAM:data{$$} S:data
13. endif
14. FAILAT 10
15. STACK 4000
16. execute s:database
```

ViewBlock

Synopsis: [EXECUTE] ViewBlock [start=] [number=]

Template: start,number

Path: S:

Requires: V1.3+

See also: Database, DelBlock, FindData

Type: Script

Brief: The view data module for the Database

Description

This module is an integral part of the AmigaDOS database and is not usually called from Shell.

Line-by-Line

1. The key for this script takes the two arguments representing the start record number and the number of records to display.
2. Ensures the module always displays at least one record by defaulting Number to 1.
3. Makes sure that if the module is called without correct arguments, the Start number contains something.

4-5: Re-set the bracket characters to { and }.

6. Tests if the Start variable contained something when the module was called. If it did, control jumps to Step 7; otherwise it jumps to Step 8.
7. Jumps to Step 30 and terminates the script normally.
8. Terminates the IF...ENDIF construct opened at Step 6.
9. Determines how many records will be printed and stores the result in the global variable "ThisMany".
10. This 1.3 compatible operation increments the value in ThisMany by 1 and stores the result in T:Temp. AmigaDOS 2 users can replace this calculation and line 11 with the following:

```
EVAL $ThisMany + 1 TO ENV:ThisMany
```

11. Re-sets the value in ThisMany. This line is not required if you have made the release 2 modification noted at Step 10.
12. In a 1.3 compatible fashion, this checks if the value in ThisMany is less than 1. Control continues at Step 13 if it is and Step 14 otherwise. The release 2 version of this line is as

follows:

```
if VAL $ThisMany NOT GT 1
```

13. Sets the global variable ThisMany to 1.
14. Terminates the IF...ENDIF construct opened at Step 12.
15. Raises the failure level to 11.
16. Creates the first part of an EDIT macro which will step Start records in. If Start=3, this macro file will read:

```
3n;p;
```

17. Creates the second part of the EDIT macro. This file expands as:

```
(?;n;)
STOP
```

18. Joins the two macro files with the contents of ThisMany sandwiched in between. The final macro, ViewMac looks like this:

```
3n;p;
4(?;n;)
STOP
```

19. Extracts the required records from the data file sending the result to a file. You could send the result straight to screen if you prefer.
20. EDIT produces an ERROR condition if the line numbers – records in this application – did not exist in the file. ERROR is trapped by the raised fail level (Step 15). If this test proves false, control jumps to Step 23; otherwise it continues at 21.
21. Displays a useful error message. Hopefully, this was because you entered a dodgy Start or Number value.
22. Jumps straight to Step 28 and exits back to the main menu.
23. Control only gets here when the EDIT command has produced a workable file.
24. Changes the current console screen colours...
25. Displays the file. You might prefer to use MORE here.
26. Re-sets the console colours.
27. Terminates the IF...ELSE...ENDIF construct opened at Step 20.
28. Marks an entry point for error handling.
29. Re-calls the main DATABASE program.
30. Marks an entry point for special handling.
31. Displays a prompt for the user to enter a number and...

32. ...the module calls itself. This allows a number to be entered interactively.

Listing

```
1.  .key start,number
2.  .def number 1
3.  .def start SKIP
4.  .bra {
5.  .ket }
6.  if "{start}" EQ "SKIP"
7.   skip restart
8.  endif
9.  eval {number} - {start} to ENV:ThisMany lformat "%n"
10. eval >NIL: <env:ThisMany op=+ value2=1 to T:Tmp lformat
    "%n" ?
11. copy T:Tmp to ENV:ThisMany
12. if >NIL: <ENV:ThisMany VAL NOT GT 1 ?
13.  setenv ThisMany 1
14. endif
15. failat 11
16. echo >T:ViewMac1 "{start}n;p;"
17. echo >T:ViewMac2 "(?;n;)*nSTOP"
18. join T:ViewMac1 ENV:ThisMany T:ViewMac2 AS T:ViewMac
19. edit s:Data with T:ViewMac ver=T:ViewRec
20. if error
21.  echo "Record(s) not found"
22.  SKIP ReRun
23. else
24.  echo "*e[33m"
25.  type T:ViewRec
26.  echo "*e[31m"
27. endif
28. LAB ReRun
29. execute s:DATABASE
30. LAB restart
31. echo "View record #: " noline
32. execute >NIL: s:ViewBlock ?
```

DCopy

Synopsis: [EXECUTE]DCOPY<[pat=]dir|pattern> [SINCE=<date>] [UPTO=<date>]

Template: .key pat/a,dest/a,opt1,since/k,upto/k

Path: S:

Requires: V1.3+

See also: CCOPY

Type: Script

Brief: To copy files (optionally by date) without failing

Description

This is a modified version of the DEL script. It will not overwrite "delete protected" files in the destination directory, but it will not stop either. An extra couple of lines have been added to ensure that the destination directory exists – if not, the directory is created – then it calls itself recursively. Recursion is not actually required for this example but this shows simply how the method works.

Line-by-Line

1-3. Sets the key and other script parameters. Note that source and destination parameters are required here.

4. Sets the default "since" date to the earliest supported by the Amiga's RTC.

5. Sets the default "upto" date to the current date. This should be used with care if you don't have a RTC (or it has not been programmed correctly). A very late date such as 01-Jan-99 might be a better idea.

6. This tests to see if the destination directory actually exists for this invocation. If it does the script carries on as normal, if not it jumps to Step 10.

7. Creates a file "copy" using the parameters for date and time defined by the user. This file is a script which contains the path of all files listed and a copy command for each one. Typically, a few lines of the script might look like this:

```
COPY "Workbench1.3:C/Dir" TO RAM: Clone
COPY "Workbench1.3:C/Copy" TO RAM: Clone
COPY "Workbench1.3:C/List" TO RAM: Clone
```

8. Raises the failure level to above that returned by all AmigaDOS commands. The script is now unstoppable.

9. Executes the list of commands and effectively performs the copy operation. You might like to make COPY resident before this point and remove it afterwards to speed the operation.
10. If control reached this point from Step 9, it jumps to the end of the script; otherwise it continues at Step 11. During a recursive phase this provides the way out.
11. This sets the fail level to WARN. There's a reason for this...
12. ...because if the destination directory cannot be created the script must exit. If MAKEDIR cannot create the directory because "object not found" it returns ERROR (v1.3) or WARN (v2). This could be tested but it's faster to crash the script.
13. This calls the script a second time recursively with the original parameters. There are other ways to do this (loops for instance) but this one best serves the purpose of the demonstration. (Incidentally, recursion is the only way to jump backwards in AmigaDOS 1.2!)
14. Closes the IF...ELSE...ENDIF construct opened at Step 6. ENDIF is required for correct recursion.

Listing

```
1.  .key pat/a,dest/a,opt1,since/k,upto/k
2.  .bra {
3.  .ket }
4.  .def since 01-Jan-78
5.  .def upto Today
6.  if exists {dest}
7.   list >T:copy{$$} {pat} since={since} upto={upto} FILES
    LFORMAT "COPY *"%s%s*" TO {dest} {opt1}"
8.   failat 21
9.   execute T:copy{$$}
10. else
11.  failat 5
12.  makedir {dest}
13.  execute DCOPY {pat} {dest} {opt1} since={since}
    upto={upto}
14. endif
```

DEL

Synopsis: [EXECUTE] DEL <[pat=]file|pattern> [QUIET] [ALL] [FORCE] [SINCE=<date>] [UPTO=<date>]

Template: pat/a,opt1,opt2,opt3,upto/k,since/k

Path: S:

Requires: V1.3+

See also:

Type: Script

Brief: Delete files without failing. Optionally window to a date.

Description

This is a very simple script which gets around one of those annoying little problems – pattern matched deletes which fail when they encounter a protected file or a directory. Much the same effect is possible using SPAT incidentally, but this script is designed for the job and does it better. Also, this script adds the option of a "date windowed" delete. Refer to LIST to see how to use the SINCE and UPTO options.

With little modification this script could be used to copy files by date in the same way that they are deleted.

Line by line

1: The key used here is a simple one because this is, in essence, a simple script. The pattern is required and can be any valid AmigaDOS wildcard. Of the other three switches, only "opt1" is usable in v1.3 and 1.3.2. This is the QUIET switch. Note: The order of the arguments is important it could be fixed later in the script – but this is a simple example. UPTO and SINCE work just like normal keywords allowing you to window the delete.

2-3. Re-define < and > to { and }.

4. Sets the default starting date to 1st January 1978 – no files can exist before this date. This keyword must have a default value since it is a keyword in LIST.

5. Sets the other part of the window to Today's date. Like SINCE, this is also treated as a keyword by LIST so it must have some value. It is possible for files to exist in the future (if you don't have a real time clock) these should either be deleted manually or the option removed from the script. Better still, get a RAM expansion with a clock.

6. This is the crucial part in this script. LIST creates a file in T: called DELETE<n> where n is the current CLI number, making it unique to this invocation. The pattern used for this list is retrieved from the pattern the user entered. SINCE and UPTO provide a date window for the files. If not supplied, all files are listed. Note: The files switch prevents LIST from displaying directories. The escaped quotes surrounding %S%S allow spaces and other odd characters in filenames and stops them interfering with the command line. The output from this command could look something like this:

```
DELETE "RAM:Temp1"
DELETE "RAM:Test"
DELETE "RAM:Space file"
DELETE "RAM:Work"
```

- In AmigaDOS 2 and above, the ALL and FORCE switches come into effect here. These must be used with great care – you have been warned!

7. Sets the failure level to 21. Generally a bit risky – but it's OK.

8. Executes the newly created script – deleting the files one-by-one. If the files are protected against deletion the failure level of 21 prevents the script from stopping.

   With little modification this script could be used to delete empty directories or copy files by date in the same way that they are deleted. The copy script needs some extra work – which we'll see later on, for now, here is the amended line to delete directories:

Listing

```
1.  .key pat/a,opt1,opt2,opt3,upto/k,since/k
2.  .bra {
3.  .ket }
4.  .def since 01-Jan-78
5.  .def upto Today
6.  LIST >T:delete{$$} {pat} {opt2} since={since}
    upto={upto}files lformat "DELETE *"%s%s*" {opt1} {opt3}"
7.  FAILAT 21
8.  EXECUTE T:delete{$$}
```

DIRS

Synopsis: [EXECUTE] DIRS

Template: none

Path: S:

Requires: V1.3+

See also: VOLS

Type: Script

Brief: List just the current directory assignments

Line-By-Line

1. Sends the current assignment list to a temporary file. Note the use of <$$> to prevent multi-tasking clashes.
2. Prints a heading of what is about to happen...
3. ...and displays (by searching for) all those names with two spaces. This is only true for the current logical directory assignments, FONTS:, ENVARC: and so on.

Listing

```
1.  ASSIGN >T:temp<$$>
2.  ECHO "Directories:"
3.  SEARCH T:temp<$$> "  " nonum  ; note "two spaces"
```

DiskDoc

Synopsis:	[EXECUTE] DISKDOC
Template:	none
Path:	S:
Requires:	V1.3 - 1.3.3
See also:	DOCTOR (Alias)
Type:	Script/Alias combined
Brief:	Multi-task DISKDOCTOR in the background.

Description

Diskdoctor cannot normally be run in the background, but there is more than one way to skin a command. This solution uses two techniques – an alias and a script. The script will do the work of running DISKDOCTOR and the alias will run the script.

Add this line to the Shell-startup script (using ED S:Shell-startup). A detailed explanation appears in the definition of this ALIAS proper.

```
ALIAS DOCTOR NEWCLI WINDOW CON:0/3/500/100/DiskDoc FROM
    S:DiskDoc
```

Now close the Shell and re-open it to ensure the alias is defined and type DOCTOR to get started.

Line-By-Line

1 Raises the failure level to 21 – beyond anything generated by AmigaDOS commands. In other words, this script cannot be stopped by any errors!

2 Executes DISKDOCTOR and starts processing drive 0 – you can change this to any drive you require. This script cannot take parameters because it is executed specially from the alias.

3 Checks if DISKDOCTOR generated a serious error. (For instance, if there is no disk in the target drive – df0: in this case.) Normally the script would grind to a halt at this point and leave you at the CLI prompt but this has already been prevented at line 1. Since we have turned normal error handling off, we must deal with this and that's what this does. If DISKDOCTOR exits normally, control skips to line 6, if not it passes to 4...

4 ...where the error message is printed. Note the ASK command is used here: it prints the error message and waits for the user to react – giving them time to study what has happened.

5 This line shuts the CLI down and closes its window. This is the reason for pausing at line 4 – if an error had occurred you might not get to see it.

6 Terminates the IF...ENDIF construct opened at 3. This is used as a marker by the IF command but it must be present for the script to handle errors correctly. Control only gets here if DISKDOCTOR terminates normally.

7 This behaves like line 4, giving the user chance to react to any warnings or messages generated by DISKDOCTOR before the CLI window is finally closed...

8 ...here.

Listing

```
1   FAILAT 21
2   DISKDOCTOR df0:
3   IF fail
4     ASK "A serious error occurred! Press Return to exit"
5   ENDCLI
6   ENDIF
7   ASK "Press Return to exit"
8   ENDCLI
```

DRIVES

Synopsis:	[EXECUTE] DRIVES
Template:	none
Path:	S:
Requires:	V1.2+
See also:	VOLS
Type:	Script
Brief:	Show drives with mounted disks

Description

AmigaDOS's INFO command generates a lot of information. There are times when much of this is redundant as you just need to know about the mounted disks and how much space you have left on them. This script selectively locates the drives with mounted disks and lists them. Extraneous information on mounted volumes and empty drives is ignored.

Line-By-Line

1. This is a dummy key it does not affect the function of the script from a user's point of view. It must be provided although we are going to use {$$} (the current CLI number) – this allows the script to multi-task correctly.

   When EXECUTE reads a .KEY argument it copies the script into a temporary directory either :T (T on the current disk) or T: (the temporary assignment – usually RAM:T). Then it expands variables enclosed by bra "<" and ket ">" characters. That is, it replaces the variables with what the user entered. <$$> is a special case – it expands to the calling CLI number.

2-3. Change the bracket characters to { and }.

4. Asks AmigaDOS to give information (see INFO) on all the current drives. That includes the external disks, hard drive partitions and mounted disks. The output from this command is sent to a file: RAM:t{$$} using output re-direction.

5: This is the all important line. Used here to give experts a clue as to what's coming and beginners a reason for the command. The INFO command spits out a lot of information, but every drive with a disk present is listed with nn% full. SEARCH homes in on this and, therefore, only lists the drives with a disk inserted.

   Similar code can be used to list mounted disks. All you have

to do is replace "%" with "[". This causes SEARCH to list the disks showing [Mounted]. Similarly, you could add a line 6 to perform this function.

If you are using AmigaDOS1.3 or higher, you can add the NONUM switch to the end of the line. This suppresses the line numbers and makes the output more usable.

Listing

```
1   .key dummy
2   .bra {
3   .ket }
4   info >ram:t{$$}
5   search ram:t{$$} "%"
```

EDS

Synopsis: EDS
Template: na
Path: na
Requires: V1.3+
See also: FRED
Type: Alias
Brief: Edit the startup-sequence
Definition: ALIAS EDS ED S:Startup-sequence

Description:

How many times have you mis-typed ED S:Startup-sequence? This has to be one of the most difficult sequences of letters to get your pinkies round yet invented, so here's an ALIAS to get you going quickly.

Example:

```
1>EDS ; edit the startup-sequence
```

EDU

Synopsis:	EDU
Template:	na
Path:	na
Requires:	V2+
See also:	FREUD
Type:	Alias
Brief:	Edit the startup-sequence
Definition:	ALIAS EDU ED S:User-Startup

Description:

This little alias is a modification of the EDS and is used to call ED for the User-startup.

Example:

```
1>EDU ; edit the user-startup
```

EggTimer

Synopsis:	[EXECUTE] EggTimer [SOFT | MEDIUM | HARD] [TIME=<time>]
Template:	soft/s,med/s,hard/s,time/k
Path:	S:
Requires:	V2+
See also:	Pest
Type:	Script
Brief:	Time a hard-boiled egg!

Description

This script was devised as a bit of light-relief in a weak moment. As it turns out, it demonstrates some interesting problems: particularly how to use "/S" in scripts. The program forms the basis for the Pest idea (although the code is very different). No checking is performed to see if you have entered more than one switch (say SOFT and HARD) or that the time is some ridiculous value. You might like to try this for yourself.

Line-By-Line

1. Defines the argument template. Notice how this looks just like a template for a real AmigaDOS command – it's processed in a very similar way too. In the synopsis described above the switch options: Soft, Med and Hard are shown as a combined argument – but only *one* of these should be supplied at any time. It is important to note the AmigaDOS parser will *not* check the presence of too many or too few switches. Such error checking will usually be performed in the script – it has not been implemented here to keep the listing simple.
2. Changes the default opening angle bracket character to "{".
3. As Step 2 for the closing bracket.

4-6. Adds EVAL, WAIT and TYPE to the resident list. This pre-loads the commands from disk and makes them available in system RAM where they can be executed faster. This technique is also handy when a disk-based command is used more than once in a script.

7. This checks if the user has entered a time via the time keyword. The exact position of this conditional test is not crucial although it should be placed early in the script. The exact workings of this line are a little complex, so let's examine them. Assume you had entered a command line thus:

```
1>EggTimer Time="3 mins"
```

The keyword Time absorbs the argument '3 mins' (quotes are required to ensure all the text is taken in). This process "sets" the internal script variable, time to '3 mins'. This can be picked up at any time by enclosing the name in special brackets – set as { and } in this script. AmigaDOS reads this line as:

```
IF "3 mins" NOT EQ ""
```

Similarly, if you do not enter a time keyword, AmigaDOS reads this:

```
IF "" NOT EQ ""
```

This statement checks if the expression on the left does not match the expression on the right – it seems a little backward at first, but it will all become clear shortly. If the test passes (first example – "3 mins" does not equal "") the script continues at the next line. If the test fails, it jumps to the closest ENDIF – at Step 9 in this case.

8. Sets a local environmental variable to the value defined by the keyword. Remember, this line is only called if a keyword and argument for "time" is supplied. Variables are like temporary containers. Local variables are held in system memory making them convenient for private storage. It is not possible to alter a local variable directly though and this must be borne in mind when deciding which type to use.

9. Closes the IF...ENDIF construct opened at Step 7. Put simply, this "command" acts as a place marker to inform AmigaDOS where to jump to when the "IF" test fails.

10-13. These lines check for the presence of the soft option on the command line (no pun intended). The position of this test is crucial in case you supply more than one switch. As programmed the switches have priority over the keyword and of those, the hard option is preferred. If soft has been supplied the variable "time" is set to three minutes – you can set this lower if you like runnier eggs or higher if you have oversize ones. An ostrich egg for instance, will take a lot longer and a much larger pan.

14-17: Sets the time for an average cooked egg. Typically this should be enough for a nicely done size three egg with a slightly hardened yolk. Adjust this timing to your own taste.

18-21: Like the previous brace of options, this sets the timing for a hard boiled egg – probably enough to kill off any trace of Salmonella. This switch overrides all others if it is supplied.

22. This pauses the script and waits for the <Return> or <Enter>

key to be pressed. This command is normally used to check for a yes or no answer, but it does this job just as easily. The "*n" enclosed in the text forces the Amiga to print a line break so the text appears split over two lines.

23. Waits for the time determined by the contents of the variable time. If, for instance, you had asked for a soft boiled egg, AmigaDOS reads this as:

```
WAIT 3 mins
```

You can insert the contents of any user-defined (environmental) variable by prefixing its name with a dollar symbol as shown here. The dollar symbol is a special variable operator and is not affected by the ".DOLLAR" operator. Note: if a badly formed command line is used, the WAIT statement will kill the script dead in its tracks. This can be avoided – but is too bothersome to warrant inclusion here.

24. This line is actually simpler than it looks and uses one of those little tricks of the trade. EVAL is generally thought of as being a mere calculator, although it is capable of much more than that. This line splits into two distinct parts:

```
EVAL >env:bleepz   7
```

This calls the command and makes it write a global environmental variable. The variable's name is taken from the text "bleep" plus the number of the Shell process executing the script. If the process was say 2, AmigaDOS reads the line as:

```
EVAL>env:bleep
```

27. In this form, this would usually send a text string to the variable – just like ECHO. However, the second part of the command does something special:

```
lformat "Dinner's up... %c"
```

This defines the output string as a message plus a non-printing character code – 7. In ASCII this code is called "BELL" and is used to flash the screen, or more usually, sound the terminal bell. (The screen flash is a peculiarity of the Amiga: under Workbench 3 you can change the simple bleep to a sampled sound.) In other words, when this variable is displayed the message will appear and the screen will flash.

25. Sets a global variable "count" to 10. Note how "{$$}" is used to attach the process number? This makes the name unique so avoiding clashes if it is executed from several Shells at once. The actual value determines how many loops will be made later on.

26. Marks the current position in the script.

27. Prints the message described at Step 24 and flashes the screen.

28. Decrements the counter variable, count. Expanded this line might read:

    ```
    eval 10 -1 to env:count2
    ```

    therefore, the variable "count2" receives the result of 9.

29. Checks if the value of "count2" is equal to zero. If it is (TRUE) control continues at Step 30 otherwise it jumps to the next ENDIF at Step 31.

30. The script reaches this point when the counter has reached zero and jumps to Step 33.

31. Terminates the IF...ENDIF construct formed at Step 29.

32. Jumps backwards to Step 26 – the label "loop". Backward jumps are quite slow because the script starts from the beginning and works down looking for the label. Generally these are placed at the start of a script wherever possible, but it makes little difference here.

33. Marks the bail out point for the SKIP command defined at Step 29.

34-36. Removes the resident commands from the system list and frees up memory. This *must* be done otherwise each successive invocation of the script will add more copies of the commands and waste memory.

Listings

```
1.  .key soft/s,med/s,hard/s,time/k
2.  .bra {
3.  .ket }
4.  Resident c:Eval add
5.  Resident c:Wait add
6.  Resident c:Type add
7.  if time NOT EQ ""
8.   set time "{time}"
9.  endif
10. if {soft} EQ "soft"
11.  set time "3 mins"
12.  echo "Computing for soft boiled egg"
13. endif
14. if {med} EQ "med"
```

```
15.  set time "4 mins 30 secs"
16.  echo "Computing for medium boiled egg"
17. endif
18. if {hard} EQ "hard"
19.  set time "6 mins"
20.  echo "Computing for hard boiled egg"
21. endif
22. ask "Place egg in boling water*nThen press <Return>"
23. wait $time
24. eval 7 lformat "Dinner's up... %c" TO=t:bleep{$$}
25. setenv count 10
26. lab loop
27. type t:bleep{$$}
28. eval >NIL: $count-1 to env:count
29. if val $count eq 0
30.  skip end
31. endif
32. skip loop back
33. lab end
34. Resident c:Eval remove
35. Resident c:Wait remove
36. Resident c:Type remove
```

EMove

Synopsis: [EXECUTE]EMove<[from=]source> <[to=]destination>

Template: from/a,to/a

Path: S:

Requires: V1.2+

See also: ...

Type: Script

Brief: The Eclectic MOVE script

Description

RENAME cannot be used to move files or directories across disks – but COPY can. However, using copy it is necessary to remove the source file after copying, so RENAME is better used on the same disk. This script solves the problems of remembering which command(s) to use by doing it all for you automatically!

You use EMove as you might use COPY/DELETE or RENAME. However the script does the hard work of deciding which to use for you. In fact, it attempts to use RENAME first, then if that doesn't work goes on to use the more longwinded version for moving between devices. The PROTECT part is optional in this script, it just makes doubly sure the source file is removed.

This example moves a file in RAM:C to DF0:C-Backups

```
1>EMOVE RAM:C/Myfile.C DFO:C-Backups
```

Under AmigaDOS 1.3+ you can use pattern matching for this function – the S bit must be set in the MOVE script for this to work:

```
1>SPAT EMOVE RAM:#? DFO:RAM-Backups
```

In AmigaDOS 2 there is no need to use SPAT since all the functions can use pattern matching anyway. The example above becomes:

```
1>EMOVE RAM:#? DFO:RAM-Backups
```

Line-By-Line

1-3. Defines the key and sets bra and ket to { and }. Note that source and destination arguments are required here.

4. Provides a simple progress message confirming what the script is up to.

5. Sets the fail level to indestructible.

6. Attempts to move the object using RENAME. This may or may not work depending on where the destination is. RENAME returns FAIL if you attempt a rename across devices; returns OK otherwise.
7. Tests if RENAME failed. If it did, control continues at Step 8; otherwise it jumps to Step 12 (and out of the script since the operation was successful).
8. Resets the failure level back to its default.
9. Performs a simple COPY operation. Redirection to NIL: is used here for the sake of users with 1.2. Later versions can make use of the QUIET switch instead which is more appropriate, in case errors are reported:

   ```
   copy "{from}" "{to}" QUIET
   ```

10. Makes the source object deletable. This isn't strictly necessary, but it might be a good idea.
11. Deletes the source object. From AmigaDOS 2, you can omit Step 10 and re-write this line as follows:

   ```
   DELETE "{from}" FORCE
   ```

12. Marks the exit point for this script.

Listing

```
1.  .key from/a,to/a
2.  .bra {
3.  .ket }
4.  echo "Moving from {from} to {to}"
5.  failat 21
6.  rename >NIL: {from} TO {to}
7.  if fail
8.   failat 10
9.   copy >NIL: "{from}" "{to}"
10.  protect >NIL: {from} +d  ; This is optional
11.  delete >NIL: "{from}"
12. endif
```

ENABLE

Synopsis: [EXECUTE] ENABLE

Template: none

Path: S:

Requires: V1.3+

See also: ListDel

Type: Script

Brief: Enable access ListDel (a script presented later)

Description

This is just about as short as scripts get – so much so, it could have been an alias. However, it does have a use – as a support routine for a script later in this book. The "jammer" variable prevents access to a potentially dangerous script. Unless this command has been issued, the script refuses to run...

Line-by-Line

1. Sets the environmental variable "jammer" to the string OFF.

Listing

```
1.  SETENV JAMMER OFF
```

EX

Synopsis: EX <Directory>

Template: na

Path: na

Requires: V1.3+

See also: ...

Type: Alias

Brief: Check an assignment without removing it

Definition: ALIAS EX ASSIGN []: EXISTS

Description:

This short alias is a very useful (and quick way) to check an assignment. Given the assigned name without the ":" it checks if the assignment exists and displays it. Example:

```
1>EX FONTS
FONTS:  Workbench3.0:Fonts
```

FACTOR

Synopsis: [EXECUTE] FACTOR <[n=]n> [[result=]private]

Template: n,result

Path: S:

Requires: V1.3.2+

See also: ...

Type: Script

Brief: To calculate factorials in the range 1..12

Description

This is one of those scripts which you don't really need. The reason that it's here is because it introduces some more of those clever little tricks AmigaDOS is capable of – with a little imagination almost anything is possible. It's also an excellent excuse to use recursion to solve a tricky little problem.

Factorials, for those who didn't do too well in mathematics, are a sequence of numbers. The factorial of a number is calculated by multiplying together all the whole numbers from one up to the number concerned. Factorials can only be obtained for positive, whole numbers. The exclamation mark is used by mathematicians to indicate a factorial – probably because they result in huge numbers! For instance, factorial 8:

```
8! = 8*7*6*5*4*3*2*1 = 40320
```

You can calculate this directly using the EVAL command in v1.3.2+:

```
1>EVAL 8*7*6*5*4*3*2*1
40320
```

it's much easier to type:

```
1>FACTOR 8
40320
```

This script has not been designed to multi-task to keep things clearer: this script uses some tough concepts to get the effect. Once these are understood, they act as a gateway to some very exciting script programming...

Line-By-Line

1. The key for this script is an unusual one because, although the user only enters a single parameter, a second parameter is available. This is used during recursion as an internal

variable; and actually ends being the result when the script "unwinds"!

2-3. Guess what, sets the bracket characters to { and }.

4. This sets the default value of the result to the value entered by the user. This has two effects: first, if the user requested factorial 1 (1!) the script exits immediately. Since 1! is 1, this forces the correct result. More importantly, for values of three and above the starting value of the result required by this script is the initial value of the required factorial. You'll see why at Step 6.

5. Recursive scripts must have an exit point – otherwise they will keep on looping until something interrupts them or the machine crashes. The latter is far more likely! This tests for a factorial value of N = 1 and forces an exit if this condition has occurred. Unless the user has entered 1, this must be calculated in the script...

6. ...here. One is subtracted from the current value of N and sent to the file T:N. A LFORMAT has been used to create a special format for this file. This is required later on in the script when the recursion takes place and will be covered in more detail then. All you need to know now is if the result of the calculation was five, the output file would read: N=5.

7. This is where things start to get a little hairy – so we'll break this line into bite sized chunks. (Or should that be "byte sized Hunks...")

```
eval {result} * ({n}-1) to t:fact
```

Takes the current expanded value of "result" and multiplies it by "n-1". Imagine – result=60 and n=3. This is equivalent to:

```
eval 60 * (3-1) to t:fact
```

The result of this calculation (120 in this case) is directed to the file T:FACT using the LFORMAT string which follows...

```
lformat ".k i*n.bra (*n.ket )*n
```

At this point you may have noticed what is about to happen – then again you may not. Don't worry too much if this seems complex – it is! That's what I mean when I say "Experts are not born, they are hewn from the bedrock of effort." This line writes a standard script header in the form:

```
.k i
.bra (
.ket )
```

You've already met something like this in an earlier example (ListDel) so if you don't remember go back to it now – you'll

need to understand that example if you are going to understand this one! This header is being generated by EVAL, which is unusual but necessary.

```
EXECUTE <t:n >nil: factor result=%n ?*n"
```

This is the second part of the LFORMAT. It generates the script main part. The header is just for support. To understand how the resultant script is going to work, it's necessary to examine a possible case. We'll maintain the assumption that the result of the calculation was 120. The completed file will look like this:

```
.k i
.bra (
.ket )
EXECUTE <t:n >nil: factor result=120 ?
```

The key "i" is just a dummy, but it must be there for the script to work. The reason for (and) brackets has already been explained. The clever bit is in the EXECUTE. This calls the script which generated it – double-recursion.

Adding to the confusion only one value "result" is passed directly. The other value is retrieved from the file "t:n". That's the reason why "t:n" was generated with the N=%n format. In AmigaDOS 2 this is far easier to achieve, as we'll see later.

8. This executes the file just created by EVAL in 7, and starts the recursion process.
9. If the value passed to "n" from the main command line equals 1 control passes here...
10. ...and the final result – also passed through the command line – is printed. The script unwinds itself after this point.

Changes for AmigaDOS 2

If that lot got your brain waving the white flag don't worry – imagine what it was like to write! In fact – it's easier than it looks once you get a hang of the basics. This script can be made a lot simpler if you have AmigaDOS 2 because there's no need to use interactive mode in the EVAL created script. Here are the amended lines and how they work:

4. The only changes to this line are the use of a real environmental variable rather than the T: assignment. Also the lformat string has been removed because it is not required:

   ```
   eval {n} - 1 to env:n
   ```

5. This line is also far simpler. It still creates a script, but now

the "n" variable which had to be expanded interactively, is expanded automatically by AmigaDOS. This removes the need for mucking around with dummy keys and bracket characters:

```
eval {result} * ({n}-1) to t:fact lformat "execute factor2
n=$n result=%n*n"
```

Listing

```
1.  .key n,result
2.  .bra {
3.  .ket }
4.  .def result {n}
5.  if val "{n}" not EQ "1"
6.   eval {n} - 1 to t:n lformat "n=%n*n"
7.   eval {result} * ({n}-1) to t:fact lformat ".k i*n.bra
    (*n.ket)*nEXECUTE <t:n >nil: factor result=%n ?*n"
8.   execute t:fact
9.  else
10.  echo "{result}"
11. endif
```

FancyList

Synopsis: [EXECUTE] FANCYLIST <[dir=]dir> [pat=<pattern>]

Template: dir/a,pat/k

Path: S:

Requires: V1.3+

See also:

Type: Script

Brief: To enhance LIST and ListAll by adding pattern matching

Description

In essence, this is an upgrade of the ListAll script shown elsewhere – but this has a far more attractive display. It searches all the directories on a disk just like the other examples, but adds the possibility of matching directories and files to different patterns. If no files match, this script tells you.

This script does not do anything new, so I haven't provided a description. You may like to discover its workings for yourself – refer back to the previous examples if you need any guidance. As an exercise you may like to add some of the extra features supported by LIST too.

Listing

```
1.  .key dir/a,pat/k
2.  .bra {
3.  .ket }
4.  .def pat #?
5.  list "{dir}" pat={pat} files to ram:Fancy{$$}
6.  echo "*nDirectory:*e[33m{dir}*e[31m" noline
7.  search ram:Fancy{$$} ":" nonum
8.  if warn
9.    echo "No files match pattern {pat}"
10. endif
11. list >T:L{$$} {dir} dirs lformat "execute s:FancyList
    *"%s%s*" pat={pat}"
12. execute T:L{$$}
```

FCD

Synopsis:	[EXECUTE] FCD [[number=] # | dir | pat]
Template:	number
Path:	S:
Requires:	V2+
See also:	RCD
Type:	Script
Brief:	Store recent directory changes and make them menus

Description

This command is very useful if you have a hard disk. It stores a list of the last ten directory changes to disk and allows you to pick one by selecting it from a numbered menu. Every path feature available to CD, including patterns, may be used. The command line is sensitive to arguments so that the script can completely replace CD (using an ALIAS) if you prefer. This command is very similar to RCD and is fully compatible with it. Several modes are available:

• Called without arguments. The script shows the current list and prompts you to interactively select an existing entry, load or save the list or enter a new directory. Note: you can enter LOAD or SAVE at this prompt. Example:

```
1>FCD
    1. "Workbench3.0"
    2. "Workbench3.0:Fonts"
    3. "Apps:"
    4. "Workbench3.0:Fonts"
    5. "Workbench3.0:Devs/Keymaps"
Enter directory or pick a number, any number:
```

• Called with a new directory path: FCD selects the directory (if available) and adds its full path to the menu. (The oldest directory is removed.) Example:

```
1>FCD SYS:
```

• Called with a number from the directory menu. The directory is selected from the list and changed. Example:

```
1>FCD 3
```

```
11.3.MB:S> fcd
     1  "DH2:lc/Mark/mac"
     2  "DH2:lc"
     3  "DH2:lc"
     4  "Apps:Cli_Fun/mark.FOF"
     5  "Apps:Cli_Fun"
     6  "WB3.0:Devs/Keymaps"
     7  "WB3.0:Libs"
     8  "DH2:"
     9  "Apps:"
Enter directory or pick a number, any number:
```

FCD Menu

Line-by-Line

1-3. Defines the template as "number" and the angle brackets as braces. It's important you don't change the template name since it is used in a recursive call.

4. Checks if some argument has been supplied. If not, control continues at Step 5; otherwise it jumps to Step 14.

5. Checks if the FCD preferences file (CDS) exists in the current S: assignment. If not, control moves to Step 10; otherwise it continues at Step 6 where...

6. The contents of the preferences file is listed with line numbers: this generates the menu.

7. Displays the main part of the interactive prompt.

8. Calls FCD recursively with interactive mode. This finishes the prompt with "number:". Note also, if you change the name of this script, you must also change this call.

9. Provides an easy exit for the script when it unwinds the recursion. Control transfers to Step 33.

10. If control gets here from Step 5, it continues at Step 11; otherwise it continues to Step 13.

11. Displays a progress message/warning.

12. Creates the CDS file by echoing the current directory. Note that automatic insertion (`command`) is used here so the directory can be enclosed in quotes. This protects CD against spaces in the directory name.

13. Terminates the IF...ELSE...ENDIF construct opened at Step 5.

14. Terminates the IF...ENDIF construct opened at Step 4.

15. Checks if the value of the entry made for number was 0. This is the case if a text entry – a directory path – was made. If text was entered, control continues at Step 16; otherwise it jumps to Step 22.

16. Attempts to set the new directory. If this command fails because the directory cannot be found (or more than one directory matches, for patterns) the script stops. Normally, the directory is made current.

17. Creates a temporary file with the new current directory name enclosed in quotes.
18. Joins the new current directory to the existing list and saves the resulting file as t:CD0#.
19. Creates a simple edit macro thus:

    9n Move down nine lines (to line 10).

    d Delete the current line.
20. Uses the macro created at Step 19 to hack off the last entry in the file. Note if there are less than ten entries (directory paths) in the file, this macro has no effect. This macro therefore, only trims off the oldest entries. Changing the line count at Step 19 affects how many lines are stored in history. More than about 25 is getting silly and less than 3 is pointless. (If you increase this number, you will have to make changes later in the script too.)
21. Replaces the old directory list with the new one.
22. If control reaches here from Step 21, it branches to Step 32; otherwise it continues at Step 23.
23. Subtracts 1 from the menu entry and stores the result in the global, Usr#.
24. Tests if the value of Usr# is less than 1 and if it is, control continues at Step 25; otherwise control jumps to Step 25.
25. Writes a simple macro to skip the first line of a file (n) and delete the next 9 lines (9d).
26. If control gets here from Step 25 if jumps to Step 28; otherwise it continues at Step 27.
27. Writes a simple macro to delete the first "Usr#" lines of a file.
28. Closes the IF...ELSE...ENDIF construct opened at Step 24.
29. Edits the history file with the macro created at Step 25 or 27 and creates a global, CD# using that information. Note that the contents of this variable can be 2 or more lines, but only the first line will be read by $CD#.
30. Changes to the selected directory.
31. Terminates the IF...ELSE...ENDIF construct opened at Step 15.
32. Marks the bail-out point for the recursion.

```
1.  .key number
2.  .bra {
3.  .ket }
4.  if "{number}" EQ ""
```

```
5.    if exists s:cds
6.     type s:cds number
7.     echo "Enter directory or pick a number, any " noline
8.     execute s:fcd ?
9.     skip number
10.   else
11.    echo "No entries in file - I'll create one!"
12.    echo >s:cds "*"`cd`*""
13.   endif
14.  endif
15.  if VAL "{number}" EQ 0
16.   cd "{number}"
17.   echo >t:cd{$$} "*"`cd`*""
18.   join t:cd{$$} s:cds AS t:cd0{$$}
19.   echo >t:ed{$$} "9n;d"
20.   edit t:cd0{$$} with t:ed{$$} ver=nil:
21.   copy t:cd0{$$} s:cds QUIET
22.  else
23.   eval >env:usr{$$} {number} -1
24.   if val $usr{$$} NOT GE 1
25.    echo >t:ed{$$} "n;9d"
26.   else
27.    echo >t:ed{$$} "$usr{$$} d"
28.   endif
29.   edit S:cds with t:ed{$$} to env:cd{$$} ver=nil:
30.   cd $cd{$$}
31.  endif
32.  lab number
```

FFIND

Synopsis: FFIND <file|pattern> <start directory>
Template: na
Path: na
Requires: 1.3+
See also: PFIND
Type: Alias
Brief: Find a file
Definition: ALIAS FFIND SEARCH SEARCH=[] FILE ALL

Description:

How often have you found yourself wondering where some file went? You know you saved it somewhere, but it seems to have disappeared into the depths of your data disk. This problem is especially nasty on a hard disk. FFIND is a solution to that problem and will easily allow you to locate any file on a disk. Typically you'll use this from the root directory, but the search can start anywhere.

Example:

```
1>FFIND StartPest SYS:  ; search for StartPest
Workbench3.0/WBStartup/StartPest
```

FRED

Synopsis: <drive number>

Template: na

Path: na

Requires: V1.3+

See also: EDS

Type: Alias

Brief: Edit the floppy disk startup-sequence on any disk

Definition: ALIAS FRED ED DF[]:S/Startup-sequence

Description:

FRED is very similar to EDS, but is useful for editing the startup on other disks. To use it just type FRED and the number of the drive whose startup you want to edit, viz:

```
1>FRED 1   ; edit startup on drive one
```

FREUD

Synopsis: <drive number>

Template: na

Path: na

Requires: V2+

See also: EDU, FRED

Type: Alias

Brief: Edit the floppy disk user-startup on any disk

Definition: ALIAS FREUD ED DF[]:S/User-Startup

Description:

FREUD is very similar to FRED, but this version edits the User-startup. To use it just type FREUD and the number of the drive whose user-startup you want to edit, viz:

```
1>FREUD 1  ; edit user-startup on drive one
```

FTEXT

Synopsis:	FTEXT <file|pattern> <start directory> [ALL]
Template:	...
Path:	...
Requires:	2.0+
See also:	FFIND
Type:	Alias
Brief:	Find ASCII text within a file or group
Definition:	ALIAS FTEXT SEARCH [] SEARCH=[a-z] PATTERN NONUM

Description:

Hands up, this one is a little weird – but I can imagine some folk will find a use for it. (Looking for passwords in protected files comes to mind.) This alias will search any file, group of files or entire tree for some files containing strings of text. No guarantee is offered for the reliability of this alias, but it actually works better than you might imagine on things like intermediate object (.o) files from C compilers and the like!

- Warning: this alias can generate a lot of output!

GetEm

Synopsis: [EXECUTE] GETEM [[pat=]name|pattern]

Template: pat

Path: S:

Requires: v1.3-1.3.3 only

Type: Script

Brief: To list the environmental variables currently defined

Description

This script is intended for use with the earlier versions of AmigaDOS because environmental variables were not fully supported. The only Amiga command to use them is the text viewer "MORE". The GETENV command can only get an environmental variable by name. This script remedies that by listing and displaying the variables by name and value.

As an extra freebie, this approach allows the implementation of pattern matching. If the script is called without a pattern, it displays all the variables in use by using "#?". This is what you'd normally do.

Line-by-Line

1. Sets the simple header for this script. A single argument is used: and it's optional too.

2-3. Set the bracket characters to { and }.

4. Sets the default pattern to "#?" – everything.

5. Lists the files in the ENV: assignment by the specified pattern – and once again the ubiquitous LFORMAT string comes into play. This script utilises a slightly unusual use of the "%s" substitution. It's obvious this line creates a simple script, so let's take a peek at what this script would look like. For the purposes of example I've only defined one variable – Editor – the script repeats for every file.

   ```
   ;env:
   ```

   How? As you may recall, if more than one "%s" expansion is used in LIST's LFORMAT, the first %s expands to the complete path minus the filename.

   Why? We want to throw this bit away! Who needs to know that the environment variables are stored in ENV:? You should already know that – and if you didn't, you should not need to

worry about it. In this way we can lose the path description safely and no one is any the wiser!

```
ECHO "Editor = " noline
```

This is reasonably obvious. It just prints the name of the variable... OK, it does a bit more than that. We sneaked in a little one here. The spaces are generated by I "*e[I" this makes echo produce a tab – helping the output line up better.

```
GETENV "Editor"
```

Displays the current value held by the variable. This is because the 3rd %s expansion, produces the name of the file. You could use TYPE by modifying the line end:

```
TYPE *"%s%s*"
```

Not as daft as it looks – you're more likely to have TYPE resident than GETENV after all.

5. Executes the script created at line 5: and displays the requested variables.

Listing

```
1.  .key pat
2.  .bra {
3.  .ket }
4.  .def pat #?
5.  list >t:getem{$$} env:{pat} files lformat ";%s*nECHO *"%s
    *e[I=*e[I*" NOLINE*nGETENV *"%s*""
6.  execute t:getem{$$}
```

GetEm 2

Synopsis: [EXECUTE] GETEM [Pat=name|pattern]

Template: Pat

Path: S:

Requires: v2

See also: GetEm 1.3

Type: Script

Brief: To list the environmental variables currently defined

Description

The original GetEm script would have worked under AmigaDOS 2 if it wasn't for Commodore fiddling around adding all those new (private) environmental variables about prefs and so on. Just kidding, those new variables are very necessary and in the proper place – the fault was with the original script. There's nothing actually wrong with the original, apart from it pre-supposing the presence of SETENV created variables. You may even wonder why this script is here at all – doesn't SETENV have the same effect? Actually it doesn't, but I didn't want you to feel left out and besides this has pattern matching!

Line-by-Line

1. Sets the simple header for this script. A single argument is used: and it's optional too.

2-3. Set the bracket characters to { and }.

4. Sets the default pattern to "#?" – everything.

5. This line comes under scrutiny again – but for an extra reason now. We've used this example to show how to use NOT pattern matching to great effect – if you haven't noticed yet, the tilde (~) symbol introduces a negative pattern. Let's take a closer look at the main group:

#?.prefs:	Ignore all private "preferences" files
#?.pat	Ignore any "pattern" files (Workbench background)
#?.info	Ignore all Workbench pictures and other info

Now let's take a look at the script this file generates using the same example...

```
Ram Disk:env/EditorRam Disk:env/
```

This bit is being thrown away. Unlike the previous example though, we are throwing away three "%s" substitutions...

```
echo "Editor = $Editor"
```

...because here we are using %s again to expand the name twice. This is a new feature of AmigaDOS 2. If %s is used more than four times, remaining substitutions are the filename. This also relies on the ability of AmigaDOS 2 to expand an environmental variable directly in a string using $. The remainder of the line is the same as the 1.3 implementation of this script – you should refer back to that example for further clarification.

5. Executes the script created at line 5: and displays the requested variables.

Listing

```
1.  .key pat
2.  .bra {
3.  .ket }
4.  .def pat #?
5.  list >t:getem{$$} env:~(#?.prefs|#?.pat|#?.info) files
    lformat "; %s%s%s*"*necho *"%s*e[I = $%s*""
6.  execute t:getem{$$}
```

Halt

Synopsis: [EXECUTE] HALT <[command=]command>

Template: command/a

Path: S:

Requires: V1.3-2.1

See also: Stop

Type: Script

Brief: To stop multiple processes with the same name

Description

This is just a modified version of the STOP script described elsewhere. The difference is that this script can stop more than one process of the same name in a single stroke. Once again, a backwards loop could have been used to get the effect - however using recursion requires less commands and tends to be more reliable. This script is meant *for emergencies* only.

Line-by-Line

1. Sets the argument template. Note that the command name is required.

2-3. Redefine bra and ket characters to { and }.

4. Find the first command in the current process list with the name defined by "Command". Note this will not work with AmigaDOS 3, because it lists all current processes of the same name.

5. This is exactly the same as the original Stopper script. However, under certain circumstances BREAK may fail with a FAULT code (Return Code=10). You should trap against this by setting the fail level to 11 with FAILAT - no higher. The script exits (rather dramatically) when there are no processes to stop and BREAK fails RC=20. This is deliberate - HALT is only meant to be used as a last resort. In very early releases of AmigaDOS 2, BREAK does not fail correctly, and this script will cause the machine to grind to a halt!

6. This line forces the script to run itself again and again until BREAK stops it.

Listing

```
1.   .key command/a
2.   .bra {
3.   .ket }
4.   status >env:stopper{$$} com={command}
5.   break <env:stopper{$$} >nil: all ?
6.   execute halt {command}
```

Host-Chat

Synopsis: [EXECUTE] HOST-CHAT

Template: none

Path: S:

Requires: V1.3+

See also: REMT-CHAT

Type: Script

Brief: Read piped messages from remote terminal

Description

Implementing and experimenting with a dual-user system is fun. Elsewhere I described some file-based messaging programs and in this part, I'll take that a step further. You may recall those programs waited for a specific time and checked for any pending messages. However, it would be much nicer if messages could be instantly display – like the chat system found in better BBSs; CIX for instance. Although this is possible using pipes, there are few minor limitations which are due to AmigaDOS's scripting language.

Users with one machine can try these examples by running two shells at once. In these examples, the "host" Shell's prompt is "1>" and the "remote" Shell's prompt is "2>". Enter the two scripts HOST-CHAT and REMT-CHAT and try the following. If you are trying this on a single machine, you will probably find it easier to open two Shells before proceeding. Also, make sure you enter the commands in the correct Shell window.

```
1>RUN HOST-CHAT
2>RUN REMT-CHAT
1>ALIAS CHAT COPY * TO PIPE:B
2>ALIAS CHAT COPY * TO PIPE:A
```

The first two lines start the pipe-based chat system as an asynchronous process. Once activated it cannot be turned off, but the operation is completely transparent. Note also, both programs must be run before an inter-shell conversation can take place. The last two lines might seem a little strange, but rely on a feature of ALIAS whereby aliases are local to the Shell process in which they were created. In other words, "CHAT" in Shell 1 will execute the command:

```
COPY * TO PIPE:B
```

and "CHAT" in Shell 2 will execute:

```
COPY * TO PIPE:A
```

This version of COPY may seem unfamiliar because the source file is an asterisk – normally used as a wildcard character in other systems. Under AmigaDOS, the * used in this way refers to Shell window; specifically, take input directly from the keyboard and copy it to the named pipe. When this command is executed, all keyboard input is directed to the pipe. To return to the Shell, you must enter the EOF (End of File) sequence by holding down CTRL and pressing \. This forms the basis of the chat system.

Assuming you have started the chat scripts in two Shells as detailed above, you can now start chatting. Try this:

```
1>CHAT
Hello World
Is anyone out there?
[Press CTRL+\ here]
1>
```

All being well, that message will appear instantly in the other Shell window:

```
2>Hello World
Is anyone out there?
```

The same command can be repeated from the other Shell to reply or send another message. Note the prompt (2>) will not appear in the receiving Shell after the message. This is quite normal and does not affect the Shell's operation. Pressing return in the receiving Shell will return a prompt. An interesting feature of AmigaDOS is once you start typing, the chat system is disabled – it won't interrupt half-way through entering a command line.

Line-by-Line

1. Defines an arbitrary label for the script to jump to when it loops.
2. Serves two functions. First, if a message is waiting in the pipe, it is displayed (typed) immediately. Second, if no messages are waiting, TYPE halts until one appears. This is an action of the pipe device – not the command.
3. The script only reaches this point after a message has been posted to the receiving pipe (above) and displayed. After this happens the script is sent back (via the label at Step 1) and waits for the next message at Step 2.

Listing

```
1.  lab start
2.  type pipe:A
3.  skip start back
```

HostRead

Synopsis: [EXECUTE] HostRead [time]

Template: time

Path: S:

Requires: V

See also: RemoteRead, Mail-2-Host, Mail-2-Remote

Type: Script

Brief: Read mail messages from the host terminal

Description

This system for reading your messages employed by Mail-2-Host and Mail-2-Remote works – but it would be nicer to get the machine to read them for you. HostRead and RemoteRead were devised to do just that. They work in a similar fashion to the others, but take more advantage of the Amiga's multi-tasking properties.

This script is an unusual one because it is designed to multi-task – even though it starts its own tasks too. There are two ways of doing this, the obvious way:

```
1>RUN EXECUTE HostRead
```

and the less obvious way:

```
1>NEWSHELL
1>RUN EXECUTE HostRead
```

In the second case, you start a new Shell process before starting HostRead. This allows you to work as normal without the messages suddenly appearing in the middle of your screen. Note however, the second technique cannot be used on the remote terminal because the new Shell window will still appear on the host's terminal – phew. Also, once this script has been started, it can only be stopped by setting the StopItNow environmental variable. You can do this thus:

```
1>SETENV StopItNow ON
```

- AmigaDOS 2 users only should enter:

```
1>ECHO >ENV:StopItNow "ON"
```

The actual value is arbitrary, but once this has been done, the program will halt during its next loop. You may want to write an alias to perform this function.

Line-by-Line

1. This defines the argument template for this script. Only one argument can be supplied here, the time delay in minutes. Unless you have a fast machine and a hard disk do not set this

below 10 minutes. If no time limit is supplied, the script will check messages every 30 minutes (defined at Step 4).

2...3 Redefine the bracket characters as before.

4. Sets the variable for the time limit if none is supplied to 30.

5. This label is supplied so the script has somewhere to return to during looping. In fact, this script has been designed to loop continuously until stopped; more of that shortly.

6. This line is identical to the one used in the Mail-2-Remote program. It displays and removes the current mail messages.

7...9 Check for the existence of the StopItNow environment variable. Actually, this could have been a temporary file placed anywhere, but it is more convenient here. Note this line is identical in both versions of this program – so once the variable is set, both users will cease to get update messages.

10. Here the script is executed as a new process with the RUN command. Now this might seem a little strange, but a minor bug in AmigaDOS EXECUTE causes the SKIP at Step 12 to fail if this is not done!

11. This puts the script to sleep for the predetermined time – default of 30 minutes in this case. You might want to change this to seconds (by substituting SECS for the MINS switch) but don't forget to change the default time value. If the delay is too short the machine tends to get very tied up attempting to read messages which simply aren't there.

12. After the WAIT at Step11 times out, this forces the script to go back and do it all again.

Listing

```
1.  .key time
2.  .bra {
3.  .ket }
4.  .def time 30
5.  Lab Start
6.  list >T:ItsForMe{$$} T:#?.rmt lformat "TYPE %s%s*nDELETE
    %s%s*n"
7.  if exists env:StopItNow
8. quit
9.  endif
10. run execute T:ItsForMe{$$}
11. wait {time} mins
12. skip Start BACK
```

InterDel

Synopsis: [EXECUTE] InterDEL <[pat=]dir|pattern> [ALL]

Template: pat/a,op

Path: S:

Requires: V1.3+

See also: ...

Type: Script

Brief: Interactive delete – asks before deleting each file

Description

The idea for this script is borrowed from the *WIPE command on BBC DFS. (DFS was Acorn's original Disk Filing System or DOS). It works in very much the same way: it allows you to get a list of the files one-by-one according to your pattern and just delete the ones you want to. It is possible to add date windowing as in DEL but this tends to clutter line #5 – LIST (which is complex enough as it is). You might like to add this yourself however.

Line-by-Line

1. This line gets the user options. Only a pattern is required. If you're lucky enough to have Workbench, the ALL switch can be used.

2-3. Redefine bra and ket to { and }.

4. This line is not required for this example but is here because some possible versions of this script do require it and it's best explained sooner rather than later. It is possible that you might want to use command expansion and/or re-direction in the script which LIST is about to create. This means you need a standard .key header, plus .bra and .ket directives. This line will write the necessary lines for you. In this example the result looks like this:

```
.key i
.bra (
.ket )
```

   I've used (and) in this standard header because { and } are already being used by the main script.

5. The crux of this script is here. I've already shown how it's possible to create a script using LIST. Before explaining the ins and outs of the line let's just take a short look at the output generated by this command:

```
ASK "RAM:TempFile - delete y/N"
IF WARN
 DELETE "RAM:TempFile"
 ECHO "Deleted"
ENDIF
```

There's nothing unusual about this script. Unless you consider these five lines are generated for every single file matching the pattern! This has the effect of generating a very long linear (that's top to bottom) script. Also there is another slight flaw in this – if the file is protected against deletion, the script still says it was deleted; AmigaDOS will disagree and complain.** AmigaDOS is right – the file wasn't removed after all.

But let's take a look at how the LFORMAT part of this line works – the remainder is quite standard. This discussion is quite complex so don't worry if you have to read it a couple of times.

First, we can break it down by splitting it at every "*N" combination; remember this is where LFORMAT will break its output on a newline – just like ECHO. Already the script begins to take shape:

```
5.1 "ASK *"%s%s - delete y/N*"*n
5.2   IF WARN*n
5.3 DELETE *"%s%s*"*n
5.4 ECHO *"Deleted*"*n
5.5 ENDIF"
```

5.1. The first quote (") marks the start of the LFORMAT – this will be thrown away. Next, the command outputs ASK and a quote is escaped in with * so it is included in the output; this will become the opening quote of the ASK statement. Now the %s%s is expanded to the file and path of the current file. Next, we add the message " – delete y/N". N is capitalised because it's default. That is: assumed if you press Return. Finally, the closing quote for ASK is escaped in with a *" and the linefeed added with *n. This *must* be present on every newline in the script.

5.2 Just adds IF WARN to the output file.

****Note:** The way to cure this slight malady is to call another script which does all this for you correctly. The problem with that approach is speed – the line could have read:

```
list >>T:dele{$$} {pat} {op} FILES LFORMAT "EXECUTE IDEL-2
    *"%s%s*" "
```

5.3 First adds DELETE then escapes a quote character so the %s%s

will be interpreted as a literal string; just in case someone has used a space in a filename; also happens with files in Ram Disk! Now %s%s is expanded again to the complete path and filename. Finally the closing quote is escaped in with *"

5.4. This uses the techniques already described to add the "DELETED" message.

5.5. The closing ENDIF. This must be included or the script will not have anywhere to branch to. The closing quote is only required by LFORMAT and not included in the output file. This is fine on a 68030 based machine like the 3000, but it tends to slow things down quite a bit on the humble 7Mhz A500s. Why? Because for every file listed, EXECUTE has to expand a new miniscript (called IDEL-2 in this theoretical example) and that does take time.

6. Raises the failure level to the indestructible limit!

7. Executes the interactive delete script.

8. Assuming everything has gone according to plan, this removes the script. You might want to omit this line and read the example scripts for yourself.

Listing

```
1   .key pat/a,op
2   .bra {
3   .ket }
4   echo >T:dele{$$} ".key i*n.bra (*n.ket )*n"
5   list >>T:dele{$$} {pat} {op} FILES LFORMAT "ASK *"%s%s -
    delete y/N*"*n IF WARN*n DELETE*"%s%s*"*nECHO
    *"Deleted*"*n ENDIF"
6   failat 21
7   execute T:dele{$$}
8   delete T:dele{$$} quiet
```

IntelliRes

Synopsis: [Execute] IntelliRes <[Script=]Scriptname>

Template: Script/a

Path: S:

Requires: V1.3+

See also: ...

Type: Script

Brief: Determine which commands should be made resident

Description

Even with the latest release, a large number of AmigaDOS commands are stored on disk. (Workbench 3.1 at the time of writing). However, the RESIDENT command can be used to store transient (disk-based) commands in RAM and share them as if they were ROM based. If you use scripts a lot, making the required commands resident at the start of a script is a real chore, especially during development. This program takes the hard work out of RESIDENT by working out which commands we're using and create a simple script automatically.

It works on all versions from AmigaDOS 1.3 upwards (probably not ARP though). This script is deceivingly simple because it writes a script, which itself writes another script; almost like a friendly virus.

Understanding this program completely requires a good knowledge of AmigaDOS, so for the sake of those who are more interested in what it does, here is a short explanation. You can start IntelliRes in the following manner:

```
1>Execute IntelliRes S:ScriptName

Searching S:Scriptname

Please wait...
```

There will now be a long delay and a fair amount of disk thrashing for 1.3 users (more on that shortly). During its first phase, IntelliRes creates a script similar to that shown in Listing 2. The pause will continue as IntelliRes runs Listing 2 and creates Listing 3 – the list of commands you need to make resident. All you have to do is insert the result of Listing 3 at the head of your script. Strictly speaking you should also add a set of RESIDENT... REMOVEs at the end to clean up.

IntelliRes is not perfect and the human angle is required to a

certain extent. It is intelligent enough to work out which commands are present in the C directory so it will work on most disks. But, it cannot determine the difference between pure commands and dirty commands (those which cannot be made resident). This much is up to you.

It will occasionally become confused when it finds a sub-string which appears to be a command. This is fixed partly by including a space after the name in Step 7e of Listing 1 (every command must be followed by a space but it is not perfect. One solution is to write your scripts so every command starts at the beginning of every line (I usually use indents). In this case you can modify Line 7 thus:

```
7.  list >T:AutoRes c: lformat ";%s*nsearch >NIL: T:SearchMe
    *"**n%s *" *nif not warn*necho >>T:ResIt *"Resident
    %s%s*"*nendif"
```

Did you spot the difference? Here it is in detail:

```
search >NIL: T:SearchMe *"**n%s *" *n
```

The extra part is the "**n" just before %S. This causes SEARCH to look for a carriage return character just before the command's name. Use whatever suits you best.

Line-By-Line

1. Defines a single required command argument which will be the name of the script to process.

2-3. Change the default AmigaDOS bracket characters from < and > to { and }. There are two reasons for this: partly because this saves clashing with re-direction operators and partly because I like them that way!

4. Copies the script to be processed to the Ram Disk – T: being a logical assignment which usually resides in RAM. You should also note the name of the destination file is forced as "SearchMe". The reason for this will be explained shortly.

5. Creates the first line of Listing 3, overwriting any previously created files with the same name. This also is necessary to fix a feature (bug) in the >> (append) operator in the 1.3 Shell. Note: RESIDENT should be moved for 2.0+ machines - it is already in ROM.

6. Displays a short progress message to let the user know the script is operating normally. The message contains the source script's full path and filename specified in the command line – not the copy of it being processed.

7. This is the heart of this program – this one line creates the program-writing, program Listing 2. Here's how it works. Let's assume the command being processed is C:DIR:

7a. `LIST >T:AutoRes`

Calls the LIST command and informs it to send all its output to a file called AutoRes; Listing 2 in other words.

7b. **C:**

Is the directory to be LISTed. Every command in the C: assignment (usually SYS:C) will be listed using the format string explained below.

7c. **LFORMAT**

introduces the quoted format string. Every file displayed by LIST is processed using this string like this:

7d. **";%S*n**

The resultant script requires the filename, then the path and filename, so the first pathname must be discarded and that's what this does. When %S is used once in LIST's LFORMAT mode, it is replaced by the name of the file in the final output. Used, twice the first occurrence displays the path, the second displays the filename and so on. The ";" is a comment, which is ignored by AmigaDOS. Finally the "*n" part creates a new line. At this stage the program consists of one line:

```
;C:
```

7e. **search >NIL: T:SearchMe *"%s *" *n**

This creates the next line of the program. The command's name appears at %S and *" is used to force the inclusion of quotes in the output string without confusing AmigaDOS. You should notice there is a single space after the %S and this is very important. "*n" adds a new line and the program now looks like this:

```
;C:
search >NIL: T:SearchMe "DIR "
```

7f. **if not warn*n**

Adds a conditional branch to the script which now looks like this:

```
;C:
search >NIL: T:SearchMe "Dir "
if not warn
```

7g **echo >>T:ResIt *"Resident %s%s*"*n**

Adds the next line, which is notable for escaped quotes and the inclusion of the forced "PATH/Filename" combination. This was the reason for adding ;%S earlier. If this had not been done %S would be out of step and the line would have received "filename/PATH". (This fix is for the sake of AmigaDOS 1.3; it could have been achieved differently in AmigaDOS 2.) The program is now all but complete and looks like this:

```
    ;c:
search>NIL: T: SearchMe "Dir "
if not warn
echo >> T:ResIT " Resident C:Dir"
```

7h. **endif"**

Completes the conditional branch and ties the whole thing together. What you may not have realised yet is this program is generated for every single command in the C directory! I'll explain what it does shortly.

8. Runs the script created at Step 8 and creates the final output ready for inclusion in another program.

9-10. Ties the whole thing together so you know something happened! This displays a short message and displays the script fragment created at Step 8.

Line-By-Line: AutoRes

1. Does nothing! This is some detritus from LIST that got thrown away.
2. Searches the copy of the source script for any occurrences of the command name – DIR in this case. (See how it works, yet?) If the command is found, search CLEARS the WARN flag.
3. Tests if the WARN condition is not set. If it was, execution branches to Step 6. If the search string *was* successful, the command *is* in the script and needs to be made resident, which is taken care of...
4. ...here. Just refresh your memory and look back to Step 5 in Listing 1. That created a file which this command is going to append (>>) its output to; and its output consists of RESIDENT and the path/filename combination of the command just searched for. A complete path is required by RESIDENT, by the way.
6. Terminates the IF...ENDIF construct.
7. This listing repeats for every command in the C: assignment.

Listing 1: IntelliRes

```
1.  .key script/a
2.  .bra {
3.  .ket }
4.  copy "{script}" T:SearchMe
5.  echo >T:ResIt "Resident c:Resident"
6.  echo "Searching {script}*nPlease wait..."
```

```
7.  list >T:AutoRes c: lformat ";%s*nsearch >NIL: T:SearchMe
    *"%s *" *nif not warn*necho >>T:ResIt *"Resident %s%s
    ADD*"*nendif"

8.  execute T:AutoRes

9.  echo "Command file now available in T: as the following:"

10. type T:resit
```

****

Listing 2: AutoRes

```
1.  ;C:

2.  search >NIL: T:SearchMe "Dir "

3.  if not warn

4.  echo >>T:ResIt "Resident C:Dir ADD"

5.  endif

6.  [etc.]
```

****

Listing 3: ResIt

```
1.  Resident C:Resident

2.  Resident C:Search

3.  Resident C:Type

4.  [etc.]
```

LD

Synopsis: LD
Template: na
Path: na
Requires: V1.3+
See also: TD, ListD
Type: Alias
Brief: Select a memorised directory
Definition: ALIAS LD CD DIR_[]: ""

Description:

This command might not seem very flash, but when you consider it can select from over 90 previously memorised directories... In fact, its the other part of TD – see later – which memorises the current directory. Here is a sample of TD in operation:

```
1>CD SYS:  ; change directory to root
1>TD 0  ; mark root as directory 0
1>CD Code:LC/Examples/Headers/Include/Devices
1>TD 1  ; mark this as directory 1
1>LD 0  ; go back to SYS:
1>CD
Workbench3.0:
1>LD 1  ; go back to 1
1>CD
Code:LC/Examples/Headers/Include/Devices
1>CD
Workbench3.0:Fonts
1>LD 1
1>CD
Code:LC/Examples/Headers/Include/Devices
```

You use this alias to switch back to a directory previously saved by TD. Neat isn't it. See TD for a full description of how this works.

ListAll

Synopsis: [EXECUTE] LISTALL [[dir=]<directory>]]

Template: dir

Path: S:

Requires: V1.3+

See also:

Type: Script

Brief: To provide an ALL switch for LIST (1.3)

Description

This is another of the support scripts that bridges the gap between AmigaDOS 1.3 and AmigaDOS 2. It's a very apt demonstration of recursion at work – although only a few lines long, it can examine every file and every directory on a hard disk. It can be improved of course, but for this example I wanted to show how much recursion can do in a few short lines. This apparently simple script undergoes some very complex looping, so I'll leave the fancy bits for later...

Line by line

1. You've met this many times before – there's nothing sinister about it here. This lists the contents of the current directory or the directory specified in "{dir}". That's a clue to how this works.

2-3. For the record, these set bra and ket to { and }.

4. Lists the current directory: determined by "dir".

5. You've met lines similar to this several times before too. So what makes this one so special? Let's take a look at what it does:

   First, this creates a new file – let's call it L1 for the sake of argument. Next, all the directories matching the pattern "{dir}" are listed and sent to that file. The LFORMAT string is used to create a recursive script – this could look something like this – the order is unimportant:

```
execute ListAll "SYS:DEVS"
execute ListAll "SYS:S"
execute ListAll "SYS:System"
execute ListAll "SYS:C"
```

6. This calls the script created at Step 4 – forcing ListAll to call itself. This also acts as an exit (or unwinding) point for the

script because recursion will only occur if the directory listed at Step 4 contains sub-directories. If not, the script is exited, and control resumes in the script which called it. This will be a temporary script held in T:. When control returns to the last level and there are no more directories left to list, the script finishes.

All this may seem a bit confusing – but it's very easy once you get the hang of the idea.

Programmers often refer to the depth of recursion. There's nothing mysterious about this – it's just the number of times the routine has called itself in direct succession. We've also used the term "unwinding". These two terms mean very much the same thing: as the depth of recursion increases, the script is winding up; as the depth decreases, it is unwinding. This is better illustrated by analogy – here's one you can try for yourself that's so simple it could have come from the annals of Blue Peter:

A: Get a piece of string and make five knots in it, each about two inches apart. Each knot represents a directory somewhere in the hierarchy.

B: Grasp the first knot (the main directory) between your thumb and index finger and hold the string tight. This is what is happening when the program starts – it knows about the first directory and has a list of the four sub-directories it contains.

C: Wind the string around your finger once (loosely, this isn't worth losing a pinkie over). This is what happens when the script has called itself once. It has listed the sub-directories (knots) contained by the first sub-directory (knot).

D: This is the fiddly bit. Carefully untie the knot you have just wound up to. In effect this is what the program does – it knows it has passed this point and can never return to it. When a subscript is executed, control always returns to the line after the one calling EXECUTE. It doesn't matter to AmigaDOS if the script just happens to call itself!

E: Repeat steps C...D until you run out of knots. This is what happens when the program runs out of sub-directories to list. All it can do is unwind the string until it comes to a point where it finds more directories or has to stop.

Listing

```
1.  .key dir
2.  .bra {
3.  .ket }
4.  list "{dir}"
5.  list >T:L{$$} "{dir}" dirs lformat "execute ListAll
    *"%s%s*""
6.  execute T:L{$$}
```

ListD

Synopsis:	[EXECUTE] ListD
Template:	dum
Path:	S:
Requires:	V1.3+
See also:	TD, LD, WD
Type:	Script
Brief:	List directories memorised by TD

Description

This script is a support routine for the LD/TD alias couplet. TD uses ASSIGN to take a snapshot of the current directory and since these are added to a list, you can end up with a large number. This script just lists the assignments belonging to TD/LD and lists the labels associated with them. For example:

```
1>ListD
Label   Directory
0       Workbench3.0
1       Code:LC/Examples/Headers/Include/Devices
2       Code:LC/Examples/Headers
C       Workbench3:C
1>LD 3
1>CD
Workbench3:C
1>LD 1
1>CD
Code:LC/Examples/Headers/Include/Devices
```

Line-By-Line

1-3. Comprise a simple header. Note that the KEY variable is just a dummy: but it is required to trigger the pre-parser (so that things like {$$} are expanded correctly).

4. Sends the current assignment list to a temporary file: we'll call it "qwe0" here.

5. Searches the assignment list for names starting with "DIR_". This is a special string created by TD as a prefix to the assignment label. In other words, this line finds just those

assignments we're interested in: for speed, the script could stop here! The output is sent to "qwe1"

6. Creates a simple EDIT macro and saves it as "ED"
7. Edits the temporary file created by Step 5 and makes a new file, "qwe2". The macro used here removes the "DIR_" from the start of each line: leaving just the label intact. The macro will cope with up to 99 assignments which should be enough for the most demanding user.
8. Sorts the label file. This isn't necessary, but it helps to keep things in order for quick reference. If this line is omitted, the labels appear in the order they were created.
9. Displays the header. The extra spaces are required after the tab (*e[I) to take account of the "DIR_" that was removed.
10. Displays the list of labels.
11. Deletes the temporary files.

Listing

```
1.  .key dum
2.  .bra {
3.  .ket }
4.  assign >t:qwe0{$$}
5.  search >t:qwe1{$$} t:qwe0{$$} "DIR_" nonum
6.  echo "99(4#;n)" to=ed{$$}
7.  edit t:qwe1{$$} t:qwe2{$$} with ed{$$} ver=nil:
8.  sort t:qwe2{$$} t:qwe1{$$}
9.  echo "Label*e[I   Directory"
10. type t:qwe1{$$}
11. delete >NIL: t:(qwe(0|1|2){$$}) T:ED{$$}
```

ListDel

Synopsis: [EXECUTE] ListDEL <[pat=]dir|pattern> [ALL]

Template: none (v2+ for ALL switch)

Path: S:

Requires: V1.3+

See also: ENABLE

Type: Script

Brief: To remove a list of files in one operation (date windowed)

Description

The basic idea for this script is borrowed from the *DESTROY command on the BBC micros Disk Filing System. This allows you to get a list of the files you are about to delete before the process starts. You must set a flag to enable this operation because the script is capable of deleting not-deleteable files listed separately if some exist. The ENABLE script is used with this script. You must enable the script every time you wish to use it – this feature prevents accidents – remove it at your own risk!

Line-by-Line

1-5. Form a standard header for this type of script. Note that the date windowing feature may need to be modified for machines that lack a battery-backed RTC: a basic 1200 for instance.

6-8. Check for the existence of the global "Jammer" variable. If it does not exist the script has not been enabled and stops immediately. This line is not required for AmigaDOS 2 – if the modifications noted under 7 are implemented.

9-11. If one of these lines is encountered the script stops at once, the operation aborted, and the Jammer flag cleared. Under 1.3 and 1.3.2 this line checks the variable Jammer and makes sure it is set off – that is, not jammed! For AmigaDOS 2 this line is simply:

```
if $JAMMER NOT EQ "OFF"
```

12. This initialises a variable which will be used later on in the script.

13. Creates a file which contains the files specified by pattern and windowed by date.

14. Searches the file created at 5: for any files which have been

protected against deletion. It keys on "-" in REW- 01-Jun-90. This has been done because it's possible the file does not have default flags. For example "RWE-" would work unless the executable flag was missing too! It is far less likely that any files have this string as part of their name. This line returns WARN if no files are protected in this way. If any are found, they are listed.

15-21. Tests if any protected files are found and continues to warn the user about them. Any such files will be deleted by this script!

22-24. If the script gets here, the user has been warned that some files selected have been protected against deletion and has agreed to go ahead in any case. This line sets a variable to let the rest of the script know.

25. Tests to see that some files actually do match the pattern and displays them. If no matching files are found a WARN condition appears.

26-29. If no matching files are found (SEARCH returns WARN), an error message is displayed and the script exits.

30-33. This is the user's last chance to change his/her mind. After this point there is no return, no second chance – all files are going to be removed. Returns a WARN if there is agreement to go ahead.

34. This tests if there are delete protected files matching the supplied pattern and the user really did want to delete them. If so, execution continues at Step 35. In AmigaDOS 2 this can be written thus:

```
if $DELS{$$} EQ "ON"
```

35. This sets up a script which will set the "deleteable" protection bit on every file before deleting it. For AmigaDOS 2 this line can be written:

```
list >T:dele{$$} {pat} {opt} since={since} upto={upto}
files lformat "DELETE *"%s%s*" FORCE"
```

36-38. This sets up a delete script for all files matching the pattern (and windowed within the dates if any were specified).

39. Does the dirty bit of running the deletion script just created.

40. If the script reaches this point it has finished normally, so this makes sure that it jumps out neatly without issuing the standard error code.

41-42. The bitter end: ensures that, no matter what has happened, the "Jammer" variable is reset so "ENABLE" has to be run again before this script can be re-run.

Listing

```
1.  .key pat/a,since/k,upto/k,opt
2.  .bra {
3.  .ket }
4.  .def since 01-Jan-78
5.  .def upto Today
6.  if not exists env:jammer
7.  skip end
8.  endif
9.  if <env:JAMMER >NIL: NOT EQ "OFF" ?
10. skip end
11. endif
12. setenv DELS{$$} OFF
13. list >T:dele{$$} {pat} {opt} since={since} upto={upto}
    files nohead
14. search "T:dele{$$}" "- " nonum
15. if not warn
16. echo "*n*e[33mWARNING*e[31m: These files are marked
    against deletion!"
17. ask "They will be deleted! Do you wish to continue
    y/N?"
18. if not warn
19. skip end
20. else
21. setenv DELS{$$} ON
22. endif
23. endif
24. echo ""
25. search T:dele{$$} ":" nonum
26. if warn
27. echo "Hummm - no files seem to match pattern?"
28. skip end
29. endif
30. ask "*nDelete y/N?"
31. if not warn
32. skip end
33. endif
34. if <env:DELS{$$} >NIL: EQ "ON" ?
```

```
35. list >T:dele{$$} {pat} {opt} since={since} upto={upto}
    files lformat "PROTECT *"%s%s*" +d*nDELETE *"%s%s*""
36. else
37. list >T:dele{$$} {pat} {opt} since={since} upto={upto}
    files lformat "DELETE *"%s%s*""
38. endif
39. execute T:dele{$$}
40. echo "All gone now..."
41. skip fini
42. lab end
43. echo "Jammed... access denied"
44. lab fini
45. setenv Jammer ON
```

Mail-2-Host

Synopsis: [EXECUTE] Mail-2-Host [message=<Message>] [Name=<name>]

Template: message,name/k

Path: S:

Requires: V1.3+

See also: Mail-2-Remote

Type: Script

Brief: Send a message to the host AmigaDOS terminal

Description

This script is broadly similar to its companion script, Mail-2-Remote. See that command for a full description of the techniques involved.

Listing

```
1.  .key message,name/k
2.  .bra {
3.  .ket }
4.  .def name ItsForYou.rmts
5.  list >T:ItsForMe{$$} T:#?.hst lformat "TYPE %s%s*nDELETE
    %s%s*n"
6.  execute T:ItsForMe{$$}
7.  echo >>T:{name}.rmt "{message}"
    if "message" EQ ""
8. quit
9.  endif
10. if exists T:{name}.rmt
11. ask "Message pending. Delete y/N?"
12. if not warn
13.  quit
14. endif
15. endif
16. echo >T:{name}.rmt "Posted on: " noline
17. date >>T:{name}.rmt
18. echo >>T:{name}.rmt "{message}"
```

Mail-2-Remote

Synopsis: [EXECUTE] Mail-2-Remote <[message=]Message> [Name=<name>]

Template: Message,name/K

Path: S:

Requires: V1.3+

See also: Mail-2-Host

Type: Script

Brief: Send a mail message to a remote AmigaDOS terminal

Description

While Bruce Smith and I were compiling the original Mastering AmigaDOS 2, we often found we needed to use the same machine at once. The simplest method would have been to buy another A3000, but at the time Amiga 3000s were in short supply and very expensive. The machine has since been replaced by the even more expensive A4000/40: such is progress.

For the sake of speed, the only solution was to connect two machines back to back using AmigaDOS 2 on both. Impossible? Well it might seem like that – especially when you realise the second machine need not be an Amiga at all! In fact, just about any small computer system you happen to have lying around can be pressed into service – you will need the following items.

Remote (parasite) machine

- A small computer with monitor and serial interface
- A null modem cable with connections for the Amiga (see note)
- A simple terminal package for the second micro
- An assistant

Host (fileserver) machine

- Any Amiga
- Either: AmigaDOS 1.3 or better
- *or*: AmigaDOS 1.2 and AUX from the Fish PD collection

Note: The Amiga 1000 and A2000 models have a non-standard serial interface. Although it will be possible to use these, you must get the correct lead as a standard cable fits the printer port. This

does not affect the B2000 machine.

Typically you can use: other Amigas, Atari STs, Amstrad PCW with serial option, MTX 500, Acorn BBC B, and most cheap PC clones. If you do not have any of these, you may find a cheap CP/M machine at amateur (HAM) radio rallies. HAMs use these for packet radio but a suitable second-hand setup can be had for as little as 30 quid.

Whatever you do, try not to spend too much money and read this chapter in its entirety before parting with any cash. Using the Amiga in this way is simple but raises some interesting problems.

Important! Never, *ever*, plug or unplug a serial lead without first switching both computers off. Failure to comply with this caution can cause serious damage to your hardware and bank account. The authors and publisher cannot accept claims for damage or injury however caused, arising from following the instructions detailed here.

Getting Started

Interfacing two machines in this way is relatively easy, but once the two machines are connected, you must decide on a serial protocol. That is the way the machines will talk to each other. It's no use having the Amiga speaking Serbo-Croat at the remote and the remote trying to answer back in ancient Greek. The result will look something like this:

Τακε με το ψουρ λεαδερ

Δο ωηατ φοην?

Protocols in serial communications are just like language - so long as the two speakers agree to speak the same tongue there is no problem. There are four main parameters to consider here which are:

Baud rate	The transmission speed. 300, 1200. 2400. 4800
Word length	The number of bits in each data byte. 7 or 8
Parity	Error checking. YES or NO
Stop bits	The number of bits to send after each data byte.1, 1.5 or 2

Errors in these are what give rise to garbage.

For the purposes of this type of communication, a fairly fast data rate is required. In practise some setups refuse to work at speeds exceeding 4800 baud, the best speed can be achieved through trial and error. The other parameters should be set to 8 data bits, no parity and 1 stop bit. (Comms nuts refer to this as 4800-8N1.) You should set these at the Amiga end with the Preferences tool.

However, if your remote computer's terminal software does not support this (for instance a Prestel™ emulator) you can try 7 data bits and even parity.

Testing 1-2-3

Once the computers are configured correctly you can perform the initial test. Open a Shell on the Amiga and enter the following (remember not to type the 1> part this just shows where each new line starts):

```
1>ECHO >T:msg "Hello World"
1>COPY T:msg to AUX:
```

alternatively you can enter:

```
1>ECHO >AUX: "Hello World"
```

However, the latter method has been found to lock the AUX device thus preventing two way communication. The remote computer will echo the message "Hello World". You are now ready to enter the world of the multi-user Amiga. Note, if at any time the Amiga or the remote terminals freeze you may have to reset the machine.

Now enter:

```
1>NEWSHELL AUX:
1>
```

Nothing should happen at the host machine (your Amiga) instead the new shell will start on the remote terminal. Its screen will look something like this:

```
New shell process 2
1>
```

Now ask your assistant to enter LIST on the terminal. This will provide them with a listing of your Workbench disk *or* the currently selected volume (disk). The LIST command is usually left resident in the 1.3 Startup-sequence and this avoids troublesome disk swapping if you only have one drive. This is a nuisance in any case, but when two people are sharing one machine it can become a nightmare of Orwellian proportions.

Problems, problems...

Before going any further, it is worth noting this technique is anything but perfect: at least it's cheap. The main problem is it only works with commands that only affect the CLI or Shell window. In other words, programs which rely on the Intuition (and that includes the ED editor) will not work. Any Intuition programs launched on the terminal appear on the host machine – usually when the operator is in the middle of something!

Actually, it can be quite amusing to install a beginner on the Amiga, launch a Workbench hack such as Viacom from the remote and watch them squirm! This is because the remote terminal is purely operating as a keyboard and screen – not as a separate computer. Because of this it is not strictly correct to call the host Amiga a network fileserver. Nevertheless, there is a vast range of CLI-based commands – not just AmigaDOS ones – which do work correctly.

What next?

At this point you should be able to control many aspects of your machine remotely from your old hardware, so that's alright. But what about sending messages between the two machines? In a real situation the two terminals could be rooms apart – so chatting is out of the question – or is it? I've already mentioned the remote terminal is, in reality, just an extension of the existing machine, so provided you can send messages between two Shells, you can send messages between the two machines.

The AmigaDOS 1.3 release saw the inclusion of FIFO pipes written by Matt Dillon and these provide one method of communication between Shells. For now though, I'll concentrate on a different method. Each has pros and cons but this serves as an interesting introduction into the use of files as they compare to pipes. Listings 1 and 2 are the command scripts used to communicate using a form of Email, that is: you leave messages for the other party which can be collected later. The two programs are almost identical so I'll just describe Mail-2-Remote here.

Line-by-line

1. This first line is very important because it determines how the script will react to command lines. In this case, the script is given two parameters – a message and a filename. As you will see, the script usually determines its own filenames but you can override this feature by entering a name here. You must enclose the message body text in quotes (speech marks) or the script will fail. Typical examples might look like this:

```
1>mail-2-remote "Hi Bruce! Nice weather huh?"
1>mail-2-remote "Ol' man river" name=river
```

2...3 Set the bracket characters to { and } respectively. You may remember these default to < and > which conflicts with the redirection operators used extensively in this program.

4. This line comes into effect if no value is supplied to the name (filename) parameter. The default name is "ItsForYou.hsts".

5. This line (which makes use of my favourite command) is used to read and delete any new mail messages from the remote terminal. It's very complex so we'll break it down into its

component parts:

5.1 list

The LIST command itself

5.2 >T:ItsForMe{$$}

Names the destination script file which is created at 5.4. The file will be place in the T: assignment (usually RAM:T) and called ItsForMe[XX], where XX is the process number of the current Shell; this allows the script to create a unique name for itself in the multi-tasking environment.

5.3 T:#?.rmt

Selects any mail messages in T: which have been sent from the remote. This is done using the AmigaDOS "everything" wildcard #? with the extension .RMT. The .RMT extension is added to every message written by Mail-2-Host. This program does much the same thing at step 10 using the extension .HST.

5.4 lformat "TYPE %s%s*nDELETE %s%s*n"

Creates a temporary script program to read and remove the current messages. Assuming the remote's operator has used the name option to create a message called hello, the resulting program would look like this:

```
TYPE T:hello.rmt
DELETE T:hello.rmt
```

6. This runs the script created at Step 5, reading any pending messages and deleting them afterwards. This is done to prevent a lot of useless files jamming up the T: assignment.

7...9. These steps are used to determine if you have actually entered a message – if not, the program stops. This allows you to check for mail periodically. (Thanks to the nature of the Amiga, it is possible to design a special version of the Mail-2-system which will periodically check for incoming mail after a short time – more of which later.)

10. This determines if a message file already exists with the same name. This means the other terminal has not yet read the message so you are given the option to leave or update it...

11. ...here. Note the default operation (if you just press Return at the prompt) is to leave the message untouched. As an aide-memoire, the y/N? prompt indicates this with a capital N.

12. This determines what happen at the ASK statement (Step 11). ASK sets the WARN condition if Y was pressed and clears it otherwise. By negating the action of IF, with the NOT switch,

the script branches to 14 when you enter Y.

13. If control reaches this point the script terminates immediately.

14...15 Close the two IF...ENDIF constructs opened at Steps 10 and 12 respectively.

16...18 These lines create a compound message using the current time to show when the message was posted. This allows the receiving party to determine when the message was posted. Note the use of the *append* redirection operator ">>". Under AmigaDOS 1.3 and above, this tacks the output of any command onto the end of an existing file.

Listing

```
1.  .key message,name/k
2.  .bra {
3.  .ket }
4.  .def name ItsForYou.hsts
5.  list >T:ItsForMe{$$} T:#?.rmt lformat "TYPE %s%s*nDELETE
    %s%s*n"
6.  execute T:ItsForMe{$$}
7.  if "message" EQ ""
8.  quit
9.  endif
10. if exists T:{name}.hst
11.  ask "Message pending. Delete y/N?"
12.  if not warn
13.    quit
14.   endif
15. endif
16. echo >T:{name}.hst "Posted on: " noline
17. date >>T:{name}.hst
18. echo >>T:{name}.hst "{message}"
```

MD

Synopsis:	MD <name>
Template:	na
Path:	na
Requires:	V1.3+
See also:	DEL
Type:	Alias
Brief:	Short name for MAKEDIR
Definition:	ALIAS MD MAKEDIR

Description:

This alias is not included for padding (as it might seem to be) but it has a very serious us. MD is the MS-DOS command for MAKEDIR and MS-DOS users will feel much more at home with AmigaDOS if the command works like this.

MemBar

Synopsis: Called from Workbench

Template: ...

Path: ...

Requires: V2+

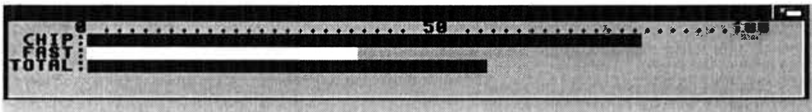
See also: FreeMemP, UsedMemP, MemBar

Type: Script

Brief: Show available memory as a bar graph

Description

This program is similar to BarClock and uses many of the same procedures.



MemBar

Line-by-Line

1-3. Comprise a standard IconX script header.

4-6. Makes some essential commands resident.

7. Sends the complete available output to a file, T#.

8. Writes the EDIT macro to extract the total amount of CHIP memory fitted to the machine. It translates thus:

d	Delete the title line
25#	Delete the first 25 characters
10>	Skip over 10 characters (total CHIP)
20#	Delete to end of line
n	Next line
2d	Delete the FAST and Total lines

9. Writes the EDIT macro to extract the total amount of FAST memory fitted to the machine. It translates thus:

2d	Delete the title and chip lines
25#	Delete the up to the total Fast fitted
10>	Skip over the total fast
20#	Delete to end of line

n Move to the total line...

d ...and delete it

10. Writes the EDIT macro to extract the total amount of memory fitted to the machine.

    3d Delete up to the total line

    25# Delete up to the total amount of RAM

    10> Skip over it and...

    20# ...delete the remainder of the line

11. Creates a global CHIP# with the total amount of CHIP memory fitted. (Macro from Step 8.)
12. Creates a global FAST# with the total amount of FAST RAM fitted. (Macro from Step 9.)
13. Creates a global TOTAL# with the total amount of memory fitted. (Macro from Step 10.)
14. Multiplies the total amount of CHIP by 10 and stores the result in CHIP#.
15. Multiplies the total amount of FAST by 10. Result is stored in FAST#.
16. Displays the "Percentage" axis.
17. Creates the editable bar of spaces. This will be used to create the bar graphics.
18. Marks the start of the repeating loop.
19. Positions the cursor at the top-left of the window and makes it invisible.
20. Gets the amount of CHIP memory free using a snapshot (`AVAIL CHIP`) and multiplies the effective result by 1000 by adding 000. This value is divided by the amount of CHIP fitted (to get the percentage) and halved (for graphing purposes). The result is sent to the global, C#.
21. Gets the amount of FAST memory free using a snapshot (`AVAIL FAST`) and multiplies the effective result by 1000 by adding 000. This value is divided by the amount of FAST fitted (to get the percentage) and halved (for graphing purposes). The result is sent to the global, F#.
22. Gets the amount of memory free using a snapshot (`AVAIL TOTAL`) and multiplies the effective result by 100 by adding 00. This value is divided by the total memory fitted (to get the percentage) and halved (for graphing purposes). The result is sent to the global, T#.
23. Checks if the global C# is the same as the local, C. If not, control passes to Step 24; otherwise it jumps to Step 27. This

function prevents the bars "flashing" on every run.

24. Positions the cursor ready to print the amount of CHIP free, clears the line and displays: CHIP:.
25. Uses ECHO's string slicing feature to output a bar proportional to the percentage of CHIP Ram in use. (A value of 50 is the full length of the scale, or 100%. Fifty characters is quite enough for an accurate(ish) display and is why the percentage ratings were halved above.)
26. Copies the current setting of C# to C, memorising it for future runs.
27. Terminates the IF...ENDIF construct from Step 23.
28. Checks if the global F# is the same as the local, F. If not, control passes to Step 29; otherwise it jumps to Step 32. This function prevents the bars "flashing" on every run.
24. Positions the cursor ready to print the amount of FAST free, clears the line and displays: FAST:.
25. Uses ECHO's string slicing feature to output a bar proportional to the percentage of FAST RAM in use.
26. Copies the current setting of F# to F, for future runs.
27. Terminates the IF...ENDIF construct from Step 28.
33. Checks if the global T# is the same as the local, T. If not, control passes to Step 34; otherwise it jumps to Step 37.
34. Positions the cursor ready to print the amount of CHIP free, clears the line and displays: CHIP:.
35. Uses ECHO's string slicing feature to output a bar proportional to the total amount of RAM in use.
36. Copies the current setting of T# to T for future runs.
37. Terminates the IF...ENDIF construct from Step 33.
38. Waits for a second. You might want to increase this on slower machines.
39. Re-starts the loop from scratch.
40. Is some information only used by WX.

Listing

```
1.  .key dummy
2.  .bra {
3.  .ket }
4.  resident c:avail
5.  resident c:wait
6.  resident c:eval
```

```
7.  avail >t:t{$$}
8.  echo >t:ed1{$$} "d;25#;10>;20#;n;2d" ; extract CHIP
9.  echo >t:ed2{$$} "2d;25#;10>;20#;n;d" ; extract FAST
10. echo >t:ed3{$$} "3d;25#;10>;20#" ; extract TOTAL
11. edit t:t{$$} to env:CHIP{$$} with t:ed1{$$}
12. edit t:t{$$} to env:FAST{$$} with t:ed2{$$}
13. edit t:t{$$} to env:TOTAL{$$} with t:ed3{$$}
14. eval $chip{$$} *10 to env:chip{$$}
15. eval $fast{$$} *10 to env:fast{$$}
16. echo "     0 ....................... 50
    .....................100"
17. echo >env:bar{$$} "
    "
18. lab start
19. echo "*e[2;0H*e[0 p" noline
20. eval (`avail chip`000/$chip{$$})/2 lformat "%n" to
    env:c{$$}
21. eval (`avail fast`000/$fast{$$})/2 lformat "%n" to
    env:f{$$}
22. eval (`avail total`00/$total{$$})/2 lformat "%n" to
    env:t{$$}
23. if $c{$$} NOT EQ $c
24.  echo "*e[40m*e[2;0H CHIP:*e[K*e[41m" noline
25.  echo "$bar{$$}" first 1 len=$c{$$} noline
26.  set c $c{$$}
27. endif
28. if $f{$$} NOT EQ $f
29.  echo "*e[40m*e[3;7H*e[K*e[3;0H FAST:*e[42m" noline
30.  echo "$bar{$$}" first 1 len=$f{$$}
31.  set f $f{$$}
32. endif
33. if $t{$$} NOT EQ $t
34.  echo "*e[40m*e[4;7H*e[K*e[4;0HTOTAL:*e[43m" noline
35.  echo "$bar{$$}" first 1 len=$t{$$}
36.  set t $t{$$}
37. endif
38. wait 1 secs
39. skip start back
40. ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory
    Gauge/SMART/NOSIZE
```

MemFreeP

Synopsis:	Called from Workbench
Template:	...
Path:	...
Requires:	V2+
See also:	MemInK, MemUsedP, MemBar
Type:	Script
Brief:	Show available memory as a percentage of total

Description

This script is a development of some of the others presented earlier, but it's a fair bit more complex. The reason is that while AVAIL can show the total amount of memory FREE as a single result, it cannot show the total amount fitted: except in tabulated form thus:

```
1>AVAIL
Type  Available   In-Use  Maximum  Largest
chip    1974040   122088  2096128  1960864
fast    1561392  1060048  2621440  1555064
total   3535432  1182136  4717568  1960864
```

Of course, your machine will probably look completely different to this: but the effect is the same. To get a percentage of memory free we have to divide the total free by the maximum available and multiply by 100. For example, for CHIP in this example:

```
1974040/2096128*100 = 94.2%
```

From AmigaDOS, this calculation is:

```
1>EVAL 1974040/2096128*100
```

Umm... The reason is AmigaDOS does some internal rounding and the fractional part of "1974040/2096128" is thrown away before the result is multiplied by 100. In fact, we can cheat and just add a couple of zeros on the top of the fraction and come up with this:

```
1>EVAL 197404000/2096128
94
```

Much better. The other problem is how do you get the values out of the tabulated output from AVAIL. This is neatly solved with some edit macros described below.

CHIP 86 percent free
FAST 43 percent free
TOTAL 62 percent free

MemFree

Line-By-Line

1-3. Comprise a standard IconX script header.

4-6. Makes some essential commands resident.

7. Sends the complete avail output to a file, T#.

8. Writes the EDIT macro to extract the total amount of CHIP memory fitted to the machine. It translates thus:

d	Delete the title line.
25#	Delete the first 25 characters.
10>	Skip over 10 characters (total CHIP).
20#	Delete to end of line.
n	Next line.
2d	Delete the FAST and Total lines.

9. Writes the EDIT macro to extract the total amount of FAST memory fitted to the machine. It translates thus:

2d	Delete the title and chip lines.
25#	Delete the up to the total Fast fitted.
10>	Skip over the total fast.
20#	Delete to end of line.
n	Move to the total line...
d	...and delete it.

10. Writes the EDIT macro to extract the total amount of memory fitted to the machine.

3d	Delete up to the total line.
25#	Delete up to the total amount of RAM.
10>	Skip over it and...
20#	...delete the remainder of the line.

11. Creates a global CHIP# with the total amount of CHIP memory fitted. (Macro from Step 8).

12. Creates a global FAST# with the total amount of FAST RAM fitted. (Macro from Step 9).

13. Creates a global TOTAL# with the total amount of memory

fitted. (Macro from Step 10).

14. Multiplies the total amount of CHIP by 10 and stores the result in CHIP#.
15. Multiplies the total amount of FAST by 10. Result is stored in FAST#.
16. Marks the start of the repeating loop.
17. Positions the cursor at the top-left of the window and makes it invisible.
18. Gets the amount of CHIP memory free using a snapshot (`AVAIL CHIP`) and multiplies the effective result by 1000 by adding 000. This value is divided by the amount of CHIP fitted and the result displayed as a percentage. AmigaDOS sees the line like this after expansion (assuming Shell #5):

```
eval 1974040000/$chip5 lformat "CHIP %n percent free *n"
```

Note you can't use the "%" symbol because this confuses EVAL's LFORMAT parser.

19. As 18 for FAST memory.
20. As 18 for the total amount of memory fitted. Note this value is expanded by a factor of 100, not 1000 as used for the other two calculations in order to avoid overflow.
21. Waits a short time...
22. ...before re-starting the loop and doing it all again.
23. Is some private information used by WX.

Listing

```
1.  .key dummy
2.  .bra {
3.  .ket }
4.  resident c:avail
5.  resident c:wait
6.  resident c:eval
7.  avail >t:t{$$}
8.  echo >t:ed1{$$} "d;25#;10>;20#;n;2d" ; extract CHIP
9.  echo >t:ed2{$$} '2d;25#;10>;20#;n;d" ; extract FAST
10. echo >t:ed3{$$} "3d;25#;10>;20#" ; extract TOTAL
11. edit t:t{$$} to env:CHIP{$$} with t:ed1{$$}
12. edit t:t{$$} to env:FAST{$$} with t:ed2{$$}
13. edit t:t{$$} to env:TOTAL{$$} with t:ed3{$$}
```

```
14. eval $chip{$$} *10 to env:chip{$$}
15. eval $fast{$$} *10 to env:fast{$$}
16. lab start
17. echo "*e[0;0H*e[0 p" noline
18. eval `avail chip`000/$chip{$$} lformat "CHIP %n percent
    free *n"
19. eval `avail fast`000/$fast{$$} lformat "FAST %n percent
    free*n"
20. eval `avail total`00/$total{$$} lformat "TOTAL %n percent
    free*n"
21. wait 1 secs
22. skip start back
23. ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory
    Gauge/SMART/NOSIZE
```

MemG (DOS 2)

Synopsis: Called from Workbench

Template: ...

Path: ...

Requires: V2+

See also: MemInK, FreeMemP, MEM6

Type: Script

Brief: Show available memory by type

Description

This version really shows the true power of AmigaDOS 2 and above. This minimal version performs exactly the same function as MEM for 1.3, but is even more succinct. You can start this script from AmigaDOS with WX thus:

```
1>WX MEM4
```

Mem in Byte

Line-by-Line

1-2. Makes some essential commands resident.

3. Marks the start of the repeating loop.

4. Positions the cursor at the top left of the window, makes it invisible, displays the memory free message, goes to the next line, displays the CHIP: message and finally inserts the amount of CHIP currently free. The "*e[K" string clears everything to the end of the line – for cases when the current amount of memory is lower than the last time.

5. Displays the current amount of FAST mem and clears to the end of the line.

6. Displays the total amount of free memory and clears to the end of the line.
7. Waits a short time before the next update...
8. ...jumps back to Step 3 and starts the whole process off again.
9. Is the window description for WX. It is not used by the script.

```
1.  resident c:avail
2.  resident c:wait
3.  lab start
4.  echo "*e[0;0H*e[0 pMemory free*nCHIP: `avail chip`*e[K"
5.  echo "FAST: `avail fast`*e[K"
6.  echo "TOTAL: `avail total`*e[K" noline
7.  wait 5 secs
8.  skip start back
9.  ;WX:WINDOW=WINDOW=con:0/0/190/40/Memory
    Gauge/SMART/NOSIZE
```

MemG

Synopsis: Called from Workbench

Template: ...

Path: ...

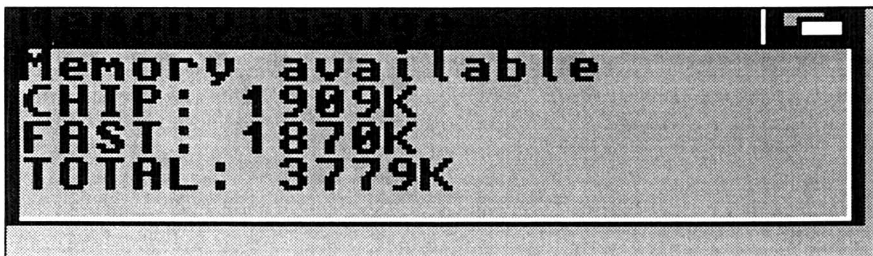
Requires: V1.3+

See also: MemG2

Type: Script

Brief: Show available memory by type

This script is the basis and the simplest memory gauge devised for this book and shows the basic idea. It's also the fastest. The idea is to open a window with the currently available memory shown and update it on a regular basis.



MemGauge

Line-by-Line

1. Provides a dummy key for IconX to use.

2-6. Make some essential commands resident. These commands are particularly relevant to the early versions of AmigaDOS and dramatically improves the performance of this script.

7. Marks the start of the repeating loop.

8. Switches the cursor off, positions the cursor at the start of the first line on the window and echoes CHIP:. The extra line-feed is suppressed so the output from AVAIL...

9. ...here, gets tacked onto it.

10-11. Display the FAST memory on one line.

12-13. Display the total amount of free memory currently available.

14. Waits a few seconds. You can change this value if you like, but five seconds is fast enough to catch large changes in memory

allocation. Very small fast changes will be too quick for such a simple configuration.

15. Re-starts the loop again by jumping to Step 7.
16. Is some information for WX to use. It is not required by the script.

Listing

```
1.  .key dummy
2.  resident c:echo
3.  resident c:Lab
4.  resident c:skip
5.  resident c:avail
6.  resident c:wait
7.  lab start
8.  echo "*e[0;0H*e[0 pCHIP: " noline
9.  avail chip
10. echo "FAST: " noline
11. avail fast
12. echo "TOTAL: " noline
13. avail total
14. wait 5 secs
15. skip start back
16. ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory_Gauge
```

MemInk

Synopsis: Called from Workbench

Template: ...

Path: ...

Requires: V1.3-1.3.3

See also: MemG

Type: Script

Brief: Show available memory (in K) by type

Description

This memory gauge is similar to Mem, but this time it shows the amount of memory available in kilobytes. Surprisingly, this is more complex than you might imagine.

Line-by-Line

1-3. Comprise a standard script header for IconX.

4-8. Make some essential commands resident.

9. Marks the start of a loop.

10. Positions the cursor at the top-left of the window and makes it invisible.

11. Displays the headline.

12. Gets the total amount of CHIP currently free and sends the result to T:CM#.

13. Uses EVAL in its interactive mode to divide the result from Step 12 by 1024 (1K). The LFORMATed result is sent to T:mem#...

14. ...and displayed here.

15-17. As 12-14 for FAST memory.

18-20. As 12-14 for the total amount of memory fitted.

21. Re-starts the loop.

22. Is some information for WX: not used by this script. This information is the WINDOW= tooltype for IconX.

Listing

```
1.  .key dummy
2.  .bra {
```

```
3.   .ket }
4.   resident c:echo
5.   resident c:Lab
6.   resident c:skip
7.   resident c:avail
8.   resident c:wait
9.   lab start
10.  echo "*e[0;0H*e[0 p"
11.  echo "Memory available"
12.  avail >T:cm{$$} chip
13.  eval <T:cm{$$} >nil: op=/ value2=1024 to=T:mem{$$} lfor-
     mat "CHIP: %nK bytes    " ?
14.  type T:mem{$$}
15.  avail >T:fm{$$} fast
16.  eval <T:fm{$$} >nil: op=/ value2=1024 to=T:mem{$$} lfor-
     mat "FAST: %nK bytes    " ?
17.  type T:mem{$$}
18.  avail >T:tm{$$} total
19.  eval <T:tm{$$} >nil: op=/ value2=1024 to=T:mem{$$} lfor-
     mat "TOTAL: %nK bytes    " ?
20.  type T:mem{$$}
21.  skip start back
22.  ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory_Gauge
```

MEM6

Synopsis: Called from Workbench

Template: ...

Path: ...

Requires: V2+

See also: MEM3, MEM4, MEM5

Type: Script

Brief: Show memory in use as a percentage

Description

This script is a development of MEM5 and this one provides a percentage of memory in use. Much better. The problem of how to get the required values out of the tabulated output from AVAIL is neatly solved with some edit macros described below.

Mem Used Percentage

Line-by-Line

1-3. Comprise a standard IconX script header.

4-7. Makes some essential commands resident.

8. Sends the complete avail output to a file, T#.

9. Writes the EDIT macro to extract the total amount of CHIP memory fitted to the machine. It translates thus:

    d Delete the title line

    25# Delete the first 25 characters.

    10> Skip over 10 characters (total CHIP).

    20# Delete to end of line.

    n Next line.

    2d Delete the FAST and Total lines.

10. Writes the EDIT macro to extract the total amount of FAST memory fitted to the machine. It translates thus:

2d Delete the title and chip lines.

25# Delete the up to the total Fast fitted.

10> Skip over the total fast.

20# Delete to end of line.

n Move to the total line...

d ...and delete it.

11. Writes the EDIT macro to extract the total amount of memory fitted to the machine.

    3d Delete up to the total line.

    25# Delete up to the total amount of RAM.

    10> Skip over it and...

    20# ...delete the remainder of the line.

12. Writes the EDIT macro to extract the amount of CHIP memory fitted in use now:

    d Deletes the header line.

    15# Deletes everything up to CHIP...In Use...

    10> ...and skips over that value.

    25# Deletes to the end of the line.

    n Moves to the next line and...

    2d ...deletes the remainder of the table.

13. Writes the EDIT macro to extract the total amount of FAST memory currently in use:

    2d Deletes the header and CHIP lines.

    15# Deletes everything up to FAST in use...

    10> ...and skips over it.

    25# Deletes everything up to the end of the line.

    n Moves to the next line and...

    d ...deletes it.

14. Writes the EDIT macro to extract the total amount of memory currently in use:

    3d Deletes the first three lines.

    15# Deletes up to the start of the total in use...

    10> and skips over it.

    25# Deletes the remainder of the line.

15. Creates a global CHIP# with the total amount of CHIP memory fitted. (Macro from Step 9.)
16. Creates a global FAST# with the total amount of FAST RAM fitted. (Macro from Step 10.)

17. Creates a global TOTAL# with the total amount of memory fitted. (Macro from Step 11.)
18. Marks the start of the repeating loop.
19. Positions the cursor at the top-left of the window and makes it invisible.
20. Creates a global NCHIP# with the total amount of CHIP memory currently in use. (Macro from Step 12.)
21. Creates a global NFAST# with the total amount of FAST RAM currently in use. (Macro from Step 13.)
22. Creates a global NTOTAL# with the total amount of memory currently in use. (Macro from Step 14.)
23. Displays the current amount of CHIP memory used as a percentage.
24. Displays the current amount of FAST memory used as a percentage.
25. Displays the total amount of memory used as a percentage.
26. Waits for a second – you can increase this delay if you prefer – but this script takes a long time to run in any case.
27. Re-starts the whole calculation/display phase again.
28. Is some private window information for WX.

```
1.  .key dummy
2.  .bra {
3.  .ket }
4.  resident c:avail
5.  resident c:wait
6.  resident c:eval
7.  resident c:edit
8.  avail >t:t{$$}
9.  echo >t:ed1{$$} "d;25#;10>;20#;n;2d" ; extract CHIP avail
10. echo >t:ed2{$$} "2d;25#;10>;20#;n;d" ; extract FAST avail
11. echo >t:ed3{$$} "3d;25#;10>;20#" ; extract TOTAL avail
12. echo >t:ed4{$$} "d;15#;10>;25#;n;2d" ; extract CHIP now
13. echo >t:ed5{$$} "2d;15#;10>;25#;n;d" ; extract FAST now
14. echo >t:ed6{$$} "3d;15#;10>;25#" ; extract TOTAL now
15. edit t:t{$$} to env:CHIP{$$} with t:ed1{$$}
16. edit t:t{$$} to env:FAST{$$} with t:ed2{$$}
17. edit t:t{$$} to env:TOTAL{$$} with t:ed3{$$}
18. lab start
```

```
19. echo "*e[0;0H*e[0 p" noline
20. edit t:t{$$} to env:NCHIP{$$} with t:ed4{$$}
21. edit t:t{$$} to env:NFAST{$$} with t:ed5{$$}
22. edit t:t{$$} to env:NTOTAL{$$} with t:ed6{$$}
23. eval ($Nchip{$$} *100)/$chip{$$} lformat "CHIP %n percent
    used *n"
24. eval ($Nfast{$$} *100)/$fast{$$} lformat "FAST %n percent
    used*n"
25. eval ($Ntotal{$$} *100)/$total{$$} lformat "TOTAL %n per-
    cent used*n"
26. wait 1 secs
27. skip start back
28. ;WX:WINDOW=WINDOW=con:0/0/190/60/Memory
    Gauge/SMART/NOSIZE
```

MID

Synopsis: [EXECUTE]MID<[name=]name>

Template: name/a

Path: S:

Requires: 1.3 - 1.3.3

See also:

Type: Script

Brief: Make a directory with an icon

Description

If you create a new directory from AmigaDOS, it does not receive a Workbench icon. This short script – MID for Make Iconified Directory – will rectify that. Note however, because the Drawer icon is copied using AmigaDOS, you will need to use the Workbench Clean-up and Snapshot functions before using it. Note also, this script is only suitable for machines fitted with AmigaDOS 1.2, 1.3 or 1.3.2.

Use your favourite text editor to create the script and save it as S:MID (no pun intended). AmigaDOS 1.3 and 1.3.2 users should set the "S" protection flag to get the best from this command.

Use: MID <name of new directory>

Example: MID DF1:Toolkit

Listing

```
1.  .key name/a
2.  .bra {
3.  .ket }
4.  MakeDir {name}
5.  Copy SYS:Empty.info TO {name}.info
```

MID1

Synopsis: [EXECUTE] MID1 <[name=]Directory>

Template: name/a

Path: S:

Requires: V1.2 - 1.3.3.

See also: MID

Type: Script

Brief: Make a directory with an icon

Description

This is a better version of MID, but the extra work it has to do makes it slower. This version looks for a suitable icon in the current directory and copies that rather than requiring the boot disk.

Example

```
1>MID1 DF1:Toolkit
```

Line-By-Line

1. Sets the key for this command which is similar to the existing MAKEDIR. Note that the name of the directory is required for this command (as it should be).

2-3 Re-set the bracket characters to { and } respectively.

4. Creates the new directory using the supplied name.

5. Discovers if an "Empty" icon exists in the current directory. If one is found, control resumes at Step 6; if not, control jumps to Step 7.

6. Creates a new drawer icon in the required directory by copying the existing "Empty" icon file.

7. If control reaches this point from Step 6, it jumps to Step 9, otherwise it continues at Step 8.

8. Copies the "Empty" icon from the boot disk to the destination directory.

9. Terminates the IF...ELSE...ENDIF construct opened at 5 and terminates the script.

Listing

```
1.   .key name/a
```

```
2.   .bra {
3.   .ket }
4.   MakeDir {name}
5.   if exists empty.info
6.    Copy Empty.info TO {name}.info
7.   else
8.    Copy SYS:Empty.info TO {name}.info
9.   endif
```

MRun

Synopsis: [EXECUTE] MRUN <[Com=]command> [parameters...] [pri=nn] [stack=nn]

Template: Com/a,x1,x2,x3,x4,x5,x6,x7,x8,x9,pri/K,stack/K

Path: S:

Requires: V1.3+

See also:

Type: Script

Brief: An improved version of RUN

Description

This script implements some ideas borrowed from ARP DOS's ARun command. The idea is to "run launch" a command with its own stack and priority. This saves having to use CHANGETASKPRI and STACK twice – once to set the required parameters for the sibling process; and again to put things back to normal. This allows you to put the multi-tasking ability of the Amiga to good use. It also demonstrates the use of EDIT in automatic mode.

Line-by-Line

1. This command key forces a command to be loaded as a required argument. The command's parameters are passed in x1...x9. The new values of STACK and PRIORITY are passed as keywords. While this does make both options, there would be little point executing this script without supplying at least one! You can use re-direction operators like this:

```
MRUN dir >prt: all stack=12000 pri=-5
```

Quotes, such as might be used in LIST's LFORMAT, should themselves be escaped with *". Very long command lines should be enclosed in quotes. In practice this script can usually be used without resorting to such methods – but they're there if you need them. If you're using output re-direction ">" and the command has a TO option (LIST for example) use that instead.

4-7. Construct a macro which will be used by the line editor, EDIT, to write a new script. This macro takes the output from STATUS and uses it to create a script to restore the original process priority and stack values after the sibling command has been launched. The starting point looks something like this:

```
Process 1: stk 4000, gv 150, pri 0 Loaded as command: status
```

and we need to end up with a script which reads:

```
STACK 4000
CHANGETASKPRI 0
```

4. **dtb /stk/;**

   removes **"Process 1:"**

   e /stk/STACK

   exchanges **"stk"** with STACK so the line now reads:

   STACK 4000, gv 150, pri 0 Loaded as command: status

5. **sb/,/**

   Splits the line at "," Note the cursor moves with "," so the line now becomes two lines:

   STACK 4000

   , gv 150, pri 0 Loaded as command: status

6. **dtb /pri/**

   Deletes **", gv 150, "**

   e /pri/CHANGETASKPRI

   Exchanges **"pri"** for CHANGETASKPRI and the lines now read:

   STACK 4000

   CHANGETASKPRI 0 Loaded as command: status

7. Finally, **dtb /L/**

   Deletes **"Loaded as command: status"**

   The original line has now become the required script, which has adopted the priority and stack from the current process.

8. This line gets the status of the current command. The second {$$} expansion ensures that MRUN only receives the status of the process which launched it.

9. Actually does the work of creating the recovery script. A bug (ok, a feature) in EDIT means an output file has to be created. The output file is the script executed at Step 13.

10. Sets the priority of the current process to either the user supplied value or a default value of 0.

11. Sets the stack of the current process to either the user supplied value or a default value of 4000 bytes.

12. RUN launches the new process passing the current values of stack and priority to the sibling. See note below.

13. The recovery script is executed here, restoring the stack and process priority back to what they were previously.

   You might have noticed by this point, something curious about this script. There's nothing unusual about it except

there is no reason why it should read: RUN... In fact, RUN could be removed and the new script called something like LAUNCH. With little extra alteration, this script can start a synchronous (as opposed to asynchronous) process with its own stack and priority. You should also insert the following before line 10:

```
FAILAT 21
```

just to make sure that, if the command fails, the old priority and stack values are restored correctly. The format of this command could read:

Listing

```
1.  .key Com/a,x1,x2,x3,x4,x5,x6,x7,x8,x9,pri/K,stack/K
2.  .bra {
3.  .ket }
4.  echo >T:mrun-0-{$$} "dtb /stk/;e /stk/STACK"
5.  echo >>T:mrun-0-{$$} "sb /,/"
6.  echo >>T:mrun-0-{$$} "dtb /pri/;e /pri/CHANGETASKPRI"
7.  echo >>T:mrun-0-{$$} "dfb /L/"
8.  status >T:mrun-2-{$$} {$$} FULL
9.  edit T:mrun-2-{$$} T:mrun-3-{$$} with T:mrun-0-{$$}
10. CHANGETASKPRI {pri$0}
11. STACK {stack$4000}
12. RUN {com} {x1} {x2} {x3} {x4} {x5} {x6} {x7} {x8} {x9}
13. EXECUTE T:mrun-3-{$$}
```

NOT

Synopsis: [EXECUTE] NOT <[com=]command> <[pat=]pattern> [options...] [only=files|dirs] [include=file|pattern]

Template: com/a,pat/a,opt1,opt2,opt3,opt4,opt5,only/k, include/k

Path: S:

Requires: V1.3-1.3.3

Type: Script

Brief: To simulate NOT pattern matching for AmigaDOS 1.3

Description

This script was developed in a similar vein to SPAT and DPAT and works in very much the same way. This particular script is very complex to implement and relies heavily on EDIT. It involves producing two lists: one matching the NOT pattern and another taking in everything. The two lists are then compared and the duplicated lines removed... Rather like using a sledgehammer to crack an Amiga – but it works.

Line-by-Line

1-3. Define the template described above and re-define bra dnd ket as { and }.

4. This creates the first EDIT macro – this one will do the job of removing the unwanted flotsam from output listing:

```
O(f/! /;p;2d;)
It breaks down as follows:
O(
```

Repeats everything enclosed in brackets until input is exhausted.

```
f/! /
```

Finds the next occurrence of a line with the string "! ". The meaning of this will become clear shortly.

```
p;2d;)
```

Moves back one line, then deletes the current line and the one below it. The closing bracket tells this command to repeat until input is exhausted. This is the end of the macro.

5. The second EDIT macro – this one just inserts a command at

the start of each line:

```
0(b/  /{com}/;n)
```

You may wonder why I didn't just do this in the LIST commands – there is a reason which I'll cover shortly. The macro breaks down as follows:

```
0(
```

As in the first macro, this starts at the top of the file and works down until it runs out of lines.

```
b/  /{com}/
```

Searches for two spaces from the start of the current line and inserts the user's command {com} before those. After inserting the command moves to the next line and continues until input is exhausted. In other words, until the whole file has been processed.

6. Creates a file which is a list of files matching the user's directory specification plus any other supplementary files. In fact, it is the supplementary files which will form the final output. LFORMAT is used here, inserting two spaces at the start of each line before the complete path and name of each file.

7-9. This block is only processed if the user has entered an optional "include" pattern at the command line – forcing NOT to include some extra files. It does this by appending a second copy of each file matching the "include" pattern to the file list. In effect, this will override the NOT pattern already specified.

8. This appends the original list of files to be excluded to the current list. Using a special LFORMAT, these files which uniquely match the NOT pattern are marked with a " !" string before the filename.

10. This line is vital – without it the script just wouldn't work. At this point, the script has built a file list which might look something like this:

```
"Startup-sequence"
"DPAT"
"Startup2"
"SPAT"
"NOT"
"PCD"
"SPAT"
!"Startup-sequence"
!"Startup2"
```

```
!"SPAT"
```

Files matching the NOT pattern (S#?) are listed twice. First with a double space and next with a " !". Files matching the include pattern are listed three times – once with " !". In this case only SPAT was specifically included. Now if we sort the file we end up with a file list which could look like this:

```
"DPAT"
"SPAT"
"SPAT"
!"SPAT"
"Startup-sequence"
!"Startup-sequence"
"Startup2"
!"Startup2"
"NOT"
"PCD"
```

11. The files are sorted in alphabetical order by name – this is forced using "colstart=3". This also explains why it is not possible to include the command at this stage. If it were, the value of colstart would be variable making the list difficult, if not impossible, to sort into the right order.
12. Fail level is adjusted here because the EDIT macros actually cause a failure with Return Code 10. This is quite normal under these circumstances.
13. This uses the first EDIT macro to remove all the lines starting with " !" including the previous line. The result of this operation using the previous example looks like this:

```
"DPAT"
"SPAT"
"NOT"
"PCD"
```

Note that SPAT has been included because it appeared in the file list three times and the EDIT macro only removes two lines. This is what was required of course.

14. This appends a blank line to the file just generated. Under normal circumstances this will not be required. It is here in case the user specified a pattern which causes the input file to become empty. A blank line gives the EDIT macro at 10 something to chew on.
15. Using the second EDIT macro, this inserts the user command

at the start of every line. This can give rise to some very interesting effects if the LIST command is combined with the DIRS option.

16. At long last, the script is executed.

NOT is a very predictable script once you get used to it; but being powerful implies that it's also prone to users' mistakes! Most of this is "bread and butter" script programming which you should be capable of. If you are at all unsure about your abilities with patterns, leave it for now.

Listing

```
1.  .key
    com/a,pat/a,opt1,opt2,opt3,opt4,opt5,only/k,include/k
2.  .bra {
3.  .ket }
4.  echo >t:auto{$$} "0(f/! /;p;2d;)"
5.  echo >t:aut1{$$} "0(b/ /{com}/;n)"
6.  list >t:temp{$$} {pat}|#? {only} lformat " *"%s%s*"
    {opt1} {opt2} {opt3} {opt4} {opt5}"
7.  if "{include}" NOT EQ ""
8.  list >>t:temp{$$} {include} {only} lformat="*"%s%s*"
    {opt1} {opt2} {opt3} {opt4} {opt5}"
9.  endif
10. list >>t:temp{$$} {pat} {only} lformat "! *"%s%s*" {opt1}
    {opt2} {opt3} {opt4} {opt5}"
11. sort t:temp{$$} t:sort{$$} colstart=3
12. failat 11
13. edit t:sort{$$} t:edit{$$} with t:auto{$$} ver=NIL:
14. echo >>t:edit{$$} "*n"
15. edit t:edit{$$} t:edi1{$$} with t:aut1{$$} ver=nil:
16. execute t:edi1{$$}
```

Pathfind

Synopsis: [EXECUTE] PATHFIND <PathName> [QUIET]

Template:

Path: S:

Requires: V1.3+

See also: ...

Type: Script

Brief: List the current paths (by letter)

Description

With Workbench 2 it is possible to get a list of the current devices, volumes or directories simply by adding an option to the ASSIGN command. Pathfinder uses a similar method to that described for DIRS and VOLS to check on the setting of any particular path. As always, if you are going to use these scripts much you should save these in the S: directory and set their S (script) protection bit. See SETS.

This little goody first appeared in Volume 2 of Mastering AmigaDOS 2, and while nothing earth shattering, it's still quite useful. The idea of this one is to allow you to view a single path, if it exits. Also, a feature of the search command means you only have to type the first few letters to get info on the required path. For instance:

```
1>PATHFIND S
Workbench 1.3:System
Workbench 1.3:S
```

or

```
1>PATHFIND SY
Workbench 1.3:System
```

It works like this:

1. Determines the command's argument template:

   PAT/A,Opt. Pat is a substring of the pathname you're interested in; Opt is passed directly to PATH and will normally be the QUIET option.

2. Sends the current path settings to a temporary file. If the QUIET option has been specified as part of the command line, PATH will not put up any "Please insert volume..." requesters.

3. Searches and displays any paths matching the substring. Note a colon is inserted prior to the search string. This forces the search to start immediately after the volume name. This

should be omitted if you want to search for partial strings anywhere in names.

Directory paths are always searched from top to bottom, so you may wish to omit the NONUM option in line 3. This will show the priority of the particular path in the search, for instance:

```
1>PATHFIND S
4 Workbench 1.3:System
5 Workbench 1.3:S
```

In other words, "System" will be searched fourth, and "S" fifth.

Listing

```
1.  .KEY PAT/A,opt
    .BRA {
    .KET }
2.  PATH >T:ptemp{$$} SHOW {opt}
3.  SEARCH T:ptemp{$$} ":{PAT}" NONUM
```

Pest (AmigaDOS 1.3)

Synopsis: none

Template: none

Path: S:

Requires: V1.3-1.3.3 only

See also: Pest (AmigaDOS 2)

Type: Startup script supplemental

Brief: Appointment/reminder program

Description

I can understand folks still using the 1.3 ROM to retain downward compatibility, but still being stuck with AmigaDOS 1.3 must be a comparative nightmare. Like all classics, AmigaDOS 1.3 is still the weapon of choice for many of you. The compatibility problem with the AmigaDOS 2 Pest is this: in AmigaDOS 2 an environmental variable can be read directly by a command by preceding its name with a dollar symbol. For instance, say the you gave the arbitrary variable, NAME a value of "Mark", the following would be true:

```
1>ECHO "Hello $NAME"
Hello Mark
```

In "The Pest" the current date is sent to a file and processed into a global environmental variable (NOW) using EDIT. Typically, a date such as:

```
Monday 2-Mar-92 12:30:04
```

becomes:

```
2-Mar-92
```

Pest2 creates a print file using NOW which will contain a string like this:

```
== Reminders for: 2-Mar-92 ==
```

The same thing can be achieved in AmigaDOS 1.3 by joining files together:

```
echo >T:pf1 "== Reminders for: " noline
echo >T:pf2 " ==*n"
join T:pf1 ENV:now T:pf2 AS T:pf
```

an alternative method which achieves the same effect looks like this:

```
echo >T:pf1 "== Reminders for: " noline
join T:pf1 ENV:now AS T:pf
```

```
echo >>T:pf " ==*n"
```

Of course, both those methods assume you want to exactly mirror the original function provided by the AmigaDOS 2 version. In practice, it would be better to just use a simpler string:

```
echo >T:pf "== Reminders for today ==*n"
```

A more subtle problem arises where the reminders file is being searched for specific dates, because the search string is read directly by AmigaDOS from the NOW variable. The solution is to trick AmigaDOS 1.3 into reading the variable from a file, and this can be accomplished using interactive mode, like this:

```
search <ENV:now s:Reminders ?
```

Here, I've reduced the command to its lowest required format. The file "S:Reminders" is being searched for the string contained in "ENV:now". Interactive mode is an important, misunderstood and very under used concept. You are probably already aware if you supply a question mark as part of a command line, AmigaDOS spits out a command's template and waits for you to enter something.

This technique was quite widely used in older versions (1.2 and earlier) to pre-load commands such as DIR. The arrival of RESIDENT in 1.3 and ROM-based AmigaDOS at 2.x means this technique has been almost forgotten however.

The key thing to remember is this: when a command enters interactive mode, it can read input from anywhere – including files. This effect can be achieved by supplying a command's argument in a file and preceding the filename with "<" (redirect input from file). Here for example, the search string is read from the contents of the file "ENV:now":

```
search <ENV:now s:Reminders ?
```

Interestingly enough, it is also possible to supply further parameters on the command line too. Therefore, since The Pest uses the NONUM switch we can add that too:

```
search <ENV:now s:Reminders NONUM ?
```

You might like to try executing the following illustration for yourself – but take a note of what happens:

```
LIST >T:Temp SYS:
ECHO >T:Search ".info"
SEARCH <T:Search T:Temp NONUM ?
```

The first two lines create a dummy file to search and something to search for respectively. This just ensures the SEARCH command will do something. Execute the search a couple of times and watch what happens. Notice how the command's template appears? If this output were being sent to a file, that template would also appear

and it looks messy. This technique is usually used with output redirected to the NIL: device and the condition codes (WARN, ERROR etc) tested, but The Pest creates a file based on SEARCH's output.

The solution therefore is to create another EDIT macro which will hack out the extraneous information and make the output look better. As it turns out, this is quite simple to do. The file consists of a header, one blank line, then the unwanted template. Therefore the EDIT macro is constructed to skip 2 lines and delete the next one like this:

```
2n;d
```

A complete script based on this idea is Pest1.3 and should be inserted just before the LOADWB command in the Startup-sequence. Alternatively, you can execute the script in its own right – but this should be done late on in the Startup-sequence.

Line-By-Line

1. Creates the first EDIT macro as "Auto1". Note this name does not require special multi-tasking treatment since it is always created by the startup-sequence. The macro reads as follows:

   ```
   DTA/ /*n
   DFA/ /
   ```

2. Creates the second EDIT macro as follows:

   ```
   2n;d
   ```

3. Makes the display file "PF" in a simple format as described above.
4. Gets the current time and date from the RTC and sends it to the file "Today".
5. Edits the date part from the time/date output from DATE leaving the result in a global variable, "now".
6. Uses SEARCH as described above in interactive mode to check for any dates in the current day and appends the list to the display file "PF" . If no dates are found for that day, SEARCH fails and sets the WARN condition.
7. If the WARN condition is set, control continues at Step 11, otherwise it jumps to Step 9.
8. Prints a simple message in the startup-sequence to let you know Pest is active and working.
9. If control reaches here from Step 8, it jumps to Step 12; otherwise it continues at Step 10.
10. Trims the "fluff" off "pf" using an EDIT macro and creates the final display message as "pf1".

11. Launches MORE as a process and displays the reminder diary in its own window. The startup script continues normally at Step 12...

12. ...closes the IF...ELSE...ENDIF construct opened at Step 7 and leaves the startup to continue as normal.

Listing

```
1.  echo >T:Auto1 "DTA/ /*nDFA/ /"
2.  echo >T:Auto2 "2n;d"
3.  echo >T:pf "== Reminders for today ==*n"
4.  date >T:today
5.  edit T:today to ENV:now with T:Auto1
6.  search >>T:pf <ENV:now s:Reminders nonum ?
7.  if warn
8.   echo "Nothing in reminder diary today..."
9.  else
10.  edit T:pf to T:pf1 with T:Auto2
11.  run more T:pf1
12. endif
```

Pest (AmigaDOS 2)

Synopsis: [EXECUTE] Pest

Template: none

Path: S:

Requires: V2+

See also: Pest 1.3

Type: Startup-script additional

Brief: Appointment scheduler, reminder

Description

When was the last time you forgot an important appointment? Moreover, if you keep a diary do you remember to check it every day? Isn't it just too easy to get engrossed in a computing session and forget you had to nip to the dentist for a filling... This little AmigaDOS 2 specific program will check your appointments every time you start or reset your machine. In a few seconds it will calculate the current date and check your schedule for any due appointments. Although it would be possible to construct a similar script for AmigaDOS 1.3, it would slow down the Startup-sequence too much due to the extra complexity required. Also, your machine must be fitted with a real-time clock.

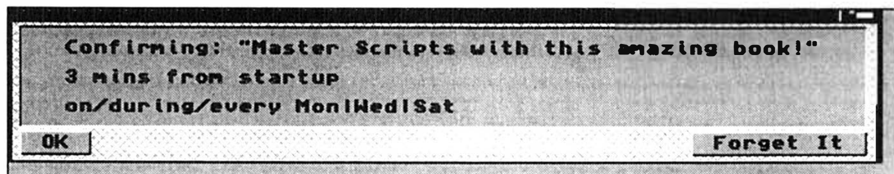
All you have to do is construct a file of appointments in the S: directory under the name: "Reminders". The format is as follows. Each reminder must fit on one line and must contain the date in AmigaDOS format, for example DD-MMM-YY.

The following are acceptable:

```
01-Mar-95 - Go to Mr Andrews for inspection at 10:00
Buy flowers for P.'s birthday: 02-apr-95
```

whereas these are not:

```
Monday: Dentist
3-2-1995 Take car to Bob for oil change
Go to show: 15/5/95
```



Pest Add Check

Line-By-Line

1. Create an auto-executing macro for EDIT using DTA and DFA. This will be used to extract just the date portion from the DATE's output. More of that later.

2. Get the current day, time and date and send it to the file: "T:Today"

3. Use the macro created at Step 1 to create a new file "ENV:Now" which contains the date string in the correct format. Here's how it works:

   DATE's output file contains a string which might look like this:

   ```
   Monday 2-Mar-94 12:30:04
   ```

   The EDIT macro removes just the day's name and the time like this:

   DTA/ / or Delete Start After the next space. Removes the day's name including the trailing space. Our example date now looks like this:

   ```
   2-Mar-94 12:30:04
   ```

   DFA/ / or Delete From After the next space. Removes the time starting with the space after the date to the end of the line. This produces the final output to file, viz:

   ```
   2-Mar-94
   ```

4. Now for another little bit of AmigaDOS 2 trickery. This line creates a file in T: containing the reminder title and the date. For instance:

   ```
   == Reminders for: 2-Mar-94 ==
   ```

   But hang on, where does the date come from? Look at the line more closely. Notice how the date appears at the position $now. In other words "$now" is replaced by the contents of the global environmental variable, "now". This variable was created right under AmigaDOS's nose in Step 3. EDIT's output file is called ENV:now.

5. The same kind of trickery is used here. The reminder file (S:Reminders) is searched for any lines containing the current date. Any lines containing that date are appended to the print

file T:pf created at Step 4. The actual date is retrieved at run-time from "$Now". (This is possible in AmigaDOS 1.3 but is much more complex to achieve.)

6. This line opens a conditional test. The WARN flag will be set if the date doesn't match any dates in the schedule file. If this is the case, control continues at Step 7; if not (a date was found) control passes to Step 8.

7. Displays a short confirmation to let you know your day is free from appointments.

8. If control reaches here directly from Step 7, it passes to Step 10, does not pass go and does not collect £200. If control came from Step 7 (a date was found) it continues at Step 9.

9. Opens MORE and displays all the appointments/reminders for that day. RUN is used to start more so the Startup-sequence can continue and launch Workbench while you study your calendar.

10. Closes the script.

Listing

```
1.  echo >T:Auto1 "DTA/ /*nDFA/ /"
2.  date to T:today
3.  edit T:today to ENV:now with T:Auto1
4.  echo >T:pf "== Reminders for: $now ==*n"
5.  search >>T:pf s:Reminders "$now" nonum
6.  if warn
7.   echo "Nothing in reminder diary today..."
8.  else
9.   run more T:pf
10. endif
```

Pest2 (AmigaDOS 1.3)

Synopsis: [EXECUTE] Pest

Template: none

Path: S:

Requires: V1.3-1.3.3 only

See also: The Pest 1.3 version

Type: Startup script supplemental

Brief: Alternative version of Pest

Description

Pest usually uses the SEARCH command but the AmigaDOS line editor also has a search feature and with a little cajoling it can be pressed into useful service. The basic idea is this: get EDIT to search for any lines containing the required string – a date in this case – and display them. In fact, this is more complex than it appears. Because EDIT is a line editor, it stops when a matching string is found on a line; initiate another search from the same position and EDIT finds the same occurrence. In other words it get locked in a loop – always assuming you can get it to loop in the first case that is.

The solution is a macro which looks like this:

```
O(f/"string"/;?;n)
```

Briefly, here's what it all does. The ";" semi-colon character is used as a command separator.

O() the commands contained in brackets are executed in a loop until the input is exhausted.

f/"string"/ Locate the string "String" anywhere in the current line, or search the text until any occurrence is found. (The string in the final script is assembled as part of the macro.) This function is case-sensitive so UPPER and lower case are different.

? Display the current line. Strictly speaking this is the verify function which sends output to EDIT's own verify display port. This is usually the current console and the relevance of this will become clear later on. Go to the next line, or stop if there is no input left to search.

Unfortunately, that is not the complete answer. EDIT normally outputs every line it scans to the console or the TO file. It also generates a separate "verify" output and this is the one we will use here. The main scan output will be sent to oblivion down the NIL:

device, and only the lines displayed with the ? command will be shown.

The complete EDIT-based Pest is longer, but the exercise gives rise to some interesting examples in its own right. You should note a lot of commands are grouped together, and thanks to the disk caching system, this reduces the amount of disk access. The Pest was only intended for AmigaDOS 2 because it takes advantage of the ROM-based (internal) commands but this script was provided as an alternative that will only work in AmigaDOS 1.3.

Line-By-Line

1. Creates an EDIT macro that will be used to extract the date component from the day/date/time format provided by DATE at Step 2.
2. Reads the current system time and date and sends it to a file called Today. Of course, your system must have a real time clock for this to be of any benefit.
3. Edits the date in the Today file as described at Step 1. The edited version is sent to the file, "now".
4. Looks in your reminder file to see if any dates match the current date read from the system clock. If no match is found, the WARN flag is set; it is cleared otherwise.
5. Tests the WARN condition from Step 4. If no matches were found (WARN=TRUE) execution continues at Step 6; otherwise it branches to Step 7.
6. Clears the screen and displays a short message. The screen is cleared using the short escape sequence: *e[0;0H*e[J. (This is available from Shell using the alias, CLEAR.)
7. If execution gets here from Step 6 it branches directly to Step 16; otherwise it continues at 8.
8. Clears the screen and displays a two line message. (See Step 6.)
9. Creates the EDIT macro, Auto2. This command tells EDIT to concatenate (join together) three consecutive lines. Literally, two lines, twice.
10. This is the first part of an EDIT macro which will form the search. The line ends at the first delimiting "/"; a line-feed will automatically be appended.
11. This is the third (not second) part of the EDIT macro mentioned above. Note how it begins with the closing "/" delimiter?
12. The three files are now married together to form something which (assuming the date was 12-Jun-94) would look like this:

```
0(f/
12-Jun-92
/;?;n)
```

Of course, that doesn't make a macro, but it is necessary to include a variable in a complex string such as this one. Next the string has to be assembled...

13. ...which is what this does. Look back at that macro, Auto2. It joins the three lines together as one and presto – a macro is created and ready to run.
14. In effect this just runs the macro, Auto3. The reminders file is scanned for the current date and any matches are displayed on the current console. The TO file is directed to NIL: so spurious rubbish produced by this command is not displayed.
15. Forces a short delay so you can examine the list of jobs to do.
16. Closes the IF...ELSE...ENDIF construct opened at Step 5.
17. Close the current Shell. It's important to note here, this command can be the last one in the normal Startup-sequence if you include either version as part of your usual startup. It must be included if you start Pest using the NEWSHELL command:

NEWSHELL FROM S:Pest

```
1.  echo >T:Auto1 "DTA/ /*nDFA/ /"
2.  date >T:today
3.  edit T:today to ENV:now with T:Auto1
4.  search >NIL: <env:now S:reminders ?
5.  if warn
6.  echo "*e[0;0H*e[JNothing in reminder diary today..."
7.  else
8.  echo "*e[0;0H*e[J== The Pest (1.3) ==*nOne moment
    please..."
9.  echo >T:Auto2 "2CL"
10. echo >T:a "0(f/"
11. echo >T:b "/;?;n)"
12. join T:a ENV:now T:b AS T:c
13. edit T:c TO T:Auto3 with T:Auto2
14. edit S:Reminders with T:Auto3 VER=* TO=NIL:
15. ask "Press <Return> to exit"
16. endif
17. endcli
```

Pest 3 (AmigaDOS 2)

Synopsis: [EXECUTE] Pest3 <[Time=]time> [[Message=]Message]

Template: time/a,Message

Path: S:

Requires: V3+

See also: Pest 3 (AmigaDOS 2 Version)

Type: Script

Brief: Bigger, better Pest scheduler

Line-By-Line

1. Defines the argument template. This will force the user into entering a time, but the message to display is optional.
2. Sets the default message string – you can enter any default message here.

3-4. Re-define the bra and ket characters from the default < and > to { and }.

5. This is a special syntax of the RUN command – little used but very useful ideal for Pest. Two re-direction operators < and > send input and output to the NIL: device. This stops the sub-process started by RUN from getting hold of the current console handles. If this were allowed to happen, the CLI window would stay open until the command is completed – and this is very messy. At the end of the string a "+" is used. This tells RUN to halt and wait for further command lines. Many commands can be chained in this way – when the last command line is encountered (the first one without the +) RUN actually starts.
6. Adds the command line to the RUN list – the process is NOT started here. When WAIT times out, the message is sent to a named pipe which is processed...
7. ...here. This starts the RUN process opened at Step 5 and allows the script to complete. When execution arrives here (after WAIT is complete) the current contents of the pipe are displayed using more.

Listing

```
1. .key time/a,Message
2.  .def Message "Wake up - time to die"
3.  .bra {
4.  .ket }
5.  run <NIL: >NIL: wait until {time} +
6.  echo >pipe:A{$$} "{Message}" +
7.  more pipe:A{$$}
```

Pest 3 (AmigaDOS 3)

Synopsis: [EXECUTE] Pest3 <[Time=]time> [[Message=]Message]

Template: time/a,Message

Path: S:

Requires: V3+

See also: Pest 3 (AmigaDOS 3 Version)

Type: Script

Brief: Bigger, better Pest scheduler

Description

Time... Have you ever noticed how life's full of it, but these days there never seems to be enough to go around. Now, just when you thought it was safe to go back to your computer, the ghost in your Startup-sequence is back with a vengeance. Pest 3 can be programmed to pop up and remind you of any appointment at any specified time (accurate to within a minute or so depending on processor load) and it won't even run down your battery. Pest 3 will run on any Amiga with AmigaDOS 2 (or above) and a real-time clock. A very powerful Workbench 3 specific version has been included for those lucky enough to have such luxuries.

Before launching headlong into a discussion of this Pest, it's worthwhile recounting how the original worked. Pest relies on reading the date from the internal BB-RTC and comparing it with a known date held in a file. Pest 3, in its most basic form, works in a different fashion – more like an event clock: you set a timed event, some time in the future, and Pest will "wake up" on (or slightly after) that event. The basic function is all based on a little used feature of the AmigaDOS WAIT command: here is the command's complete template:

```
WAIT /N, SEC=SECS/S, MIN=MINS/S, UNTIL/K:
```

The part we are interested in here is the keyword UNTIL. This forces WAIT to halt any CLI process until a specified time rather than FOR a specified time interval. Times are entered in 24 hour clock using the following format:

```
HH:MM
```

so, examples of valid times are 9:00; 12:00; 15:04 and so on.

(Using the DATE command reveals that AmigaDOS counts time in seconds too, but an exact seconds count cannot be guaranteed because of constraints imposed by the multi-tasking environment.

It is possible to write a program which will get very close – but this is unlikely to be of any real benefit and in itself would hog too much processor time.)

So much for the theory then: what happens in practice? Try entering an example like this (the exact time entered depends on what time you are trying this):

```
1>WAIT UNTIL 4:24
```

This example sets a time when most sane folk are tucked up in bed and computer freaks are excitedly bashing at keyboards. However, it is most likely your Shell has just frozen and gone to sleep. You could reset the machine now, or even wait until half-four in the morning. But a much more sensible approach would be to stop the command. Press CTRL and C together to "break" WAIT's effect.

Now let's put some more theory into practice. Start a new Shell (either from the existing one or Workbench, it doesn't matter for this). In the second Shell, which I'll call CLI 2 here, enter this:

```
2>WAIT UNTIL 5:00
```

now click back in the first Shell and enter this:

```
1>STATUS COMMAND WAIT
 2
```

Notice how AmigaDOS responds with the number of the CLI (Shell) process which is running WAIT?

Experienced AmigaDOS users already know how to start a new command in its own sub-Shell like this:

```
1>RUN WAIT UNTIL 5:00
[CLI 3]
```

While this system is perfect for many commands, it has no real practical purpose when used directly with WAIT. Several problems occur however:

- When RUN spawns the sub-process there is no way to signal back to the main process the WAIT command has completed.
- The WAIT state cannot be broken directly from the keyboard with CTRL+C – try it.

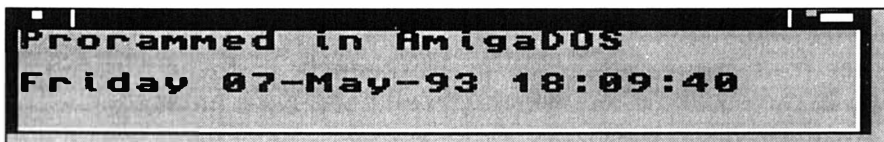
Taking this one step further by removing the "[CLI #]" message causes its own problems. Try this:

```
1>RUN >NIL: WAIT UNTIL 5-00-00
```

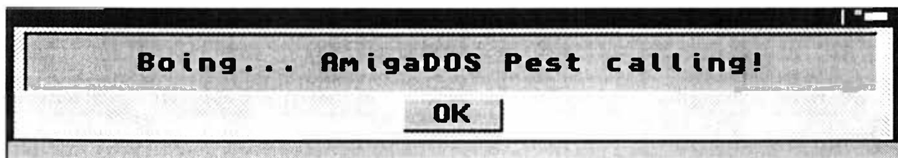
Everything seems to go according to plan but the error in the command line is not reported and WAIT does not start. Now to add insult to injury, here's the final fly in the ointment. Start a new Shell and enter this (for the sake of clarity, Shell's output has been shown):

```
1>RUN WAIT UNTIL 3:00
[CLI 4]
1>ENDCLI
Cli Process 4 ending
```

No matter what you do, Shell 4 will not go away! In fact, this Shell window will stay open until the WAIT command has completed or is forcibly stopped. All this discussion may seem far removed from Pest – but in truth it is all inextricably intertwined. The last two examples illustrate the events possible if Pest were started from the initial Shell window: therefore some kind of error checking will be required.



Dos Clock



Pest Calling

Your Basic Pest

The most basic version of Pest 3 forms a simple, message-based, alarm clock and may be sufficient for many needs. It could be run from a Startup-sequence because a special technique has been used to allow the machine to start normally – more of that in a moment. Two versions are supplied here: one for AmigaDOS 3 and a less elegant version for AmigaDOS 2. The AmigaDOS 3 version in particular can be run several times from the User-startup to warn of regular timed events: lunch, Star Trek or Coronation Street for instance... None of the simple scripts listed here do any error checking on the time format so it is up to you to get it right. Nevertheless they will not interfere with the machine's normal running: I have several Pest 3.0 alerts running while I'm writing this text in Transwrite.

The one constructed for AmigaDOS 3 is the simplest Pest and shows the most important techniques without the extra fuss required for other versions. You can call Pest from User-startup like this:

```
Pest 16:47 "Close all files - Star Trek is about to start on
    Sky 1"
```

Note that the quotes around the message are required. You can set one or more time events from any Shell like this:

```
1>Pest 18:00 "Looks like you missed Star Trek then..."
```

Line By Line

1. Defines the argument template. Only the event's time is actually required: Pest will generate its own message if you don't supply one.
2. Sets the default message if one is not supplied. You can enter any default message you want here.

3-4. Change bra (<) and ket (>) to { and }.

5. Initialises, but does not start a two command process. The command line is added to a list but the sub-Shell is not launched. When started, this line will time out and continue at Step 6 when the supplied time is reached.
6. Adds the second line to the process opened at Step 5. This line triggers the RUN...WAIT part and exits the script. The line will not actually execute until WAIT times out (that is, reaches the UNTIL time).

Listing

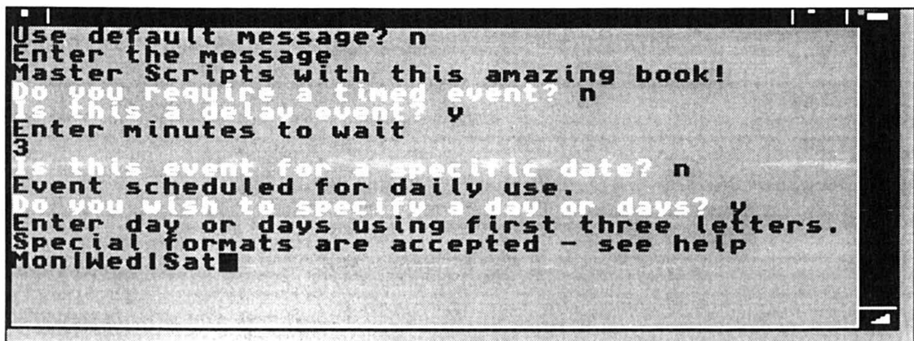
```
1. .key time/a,Message
2. .def Message "Wake up - time to die"
3. .bra {
4. .ket }
5. run <NIL: >NIL: wait until {time} +
6. RequestChoice >NIL: "Pest" "{Message}" "OK"
```

Pest 3: AddPestEvent

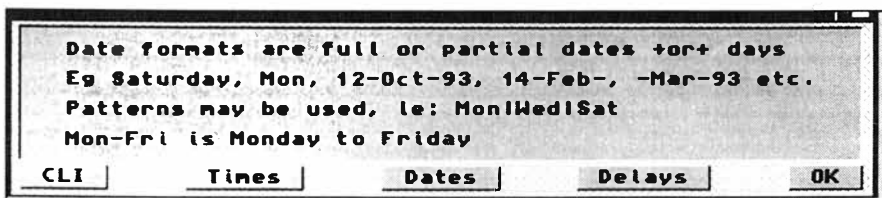
Synopsis:	Only used from Workbench
Template:	Time/K,Day/K,Wait/K,Go/K,Message/F
Path:	SYS:Pest3
Requires:	V3+ (as part of Pest 3)
See also:	DeletePestEvent, KillPestEvent, StartPest, GetArgs, DeletePestEvent, ChangeMessage, SetPestEvent, SetWaitEvent
Type:	IconX script
Brief:	The main module for Pest 3

Description

Pest is probably the most powerful and comprehensive time scheduling utility yet devised – but what makes it special and what will make other computer users pale, is the fact the whole thing is written in AmigaDOS. There are no clever assembly-language cheats, no hidden utilities: everything is handled by the Amiga's own DOS language. It's also a lot smaller and cheaper than anything you could reasonably buy off-the-shelf. It even has a help facility to get you started!



Pest Adding



Pest Date Help

Pest can handle an almost unlimited number of "events" and is accurate to within a couple of seconds of the computer's own clock

– depending on processor load. It is fully multi-tasking and works quietly in the background until it's needed. Memory requirements are frugal – each running event takes only a couple of K. Once active it will flash a message on the Workbench even if other applications (DPaint, Wordworth and so on) are running. Being disk-based, it is totally reset protected since the events are set every time the machine is started.

You may set an event (reminder) to trigger on any day, days, date, or dates: at a specified time or a set time after the last reboot. For example, you can set any event to happen at specific times on days of the week; on weekends only; every day in a specific month and so on. Typical uses include: birthdays, regular appointments, TV programs and so on. The list of possibilities is quite vast and will be seen by experiment.

Pest 3 will work on any Amiga with Kickstart/Workbench 3 and a real-time clock. Amazingly, Commodore did not supply a RTC with the A1200 so unless you have bought one separately, Pest will be of limited use. (It still works, but only the instant and delayed events are workable. Pest is so good though, it's just the excuse you need to go out and buy a RTC board.) The nature of the program means a hard disk is recommended but not required.

I can imagine a lot of readers spitting teeth and demanding to know why this all-time best Pest is AmigaDOS 3 only. The reason is simple: AmigaDOS 3 has a new command which suits Pest to a tee and implementing it in AmigaDOS 2 is technically quite tricky. It is possible to write such a utility (RequestChoice) in assembler or C but that rather defeats the object. Demand arising, I will fix this – but for now, here stands the Pest 3: batteries not required.

Starting Pest

The entire Pest3 drawer should be copied to your Workbench disk. Pest is activated by dragging the "StartPest" icon to your WBStartup drawer and re-booting the machine. After that, you can just forget it and let Pest do the rest. The program is like an alarm clock and as such you must remember a couple of things:

- It does nothing unless you tell it to.
- You have to be around to hear it.

The second point cannot be stressed strongly enough. Pest is intelligent enough to spot when you miss an appointment time on a specified day – but the machine *must* be used on that date. This is just like an alarm clock – if you're fast asleep (or not there) when it goes off, you'll be late for work just the same.

Just double click on the AddPestEvent icon and select "Start". Pest will ask you what it needs to know – the sequence of prompts is defined by exactly what the event will do. The most important, and

the only required input from you is the message Pest will display when the event occurs. You will then be given a choice of days, dates, times or delays. Note that in this version CLI options are *not* supported.

The message can be any string of text, although about 60 characters is about the maximum. There's no reason why you shouldn't use the message to remind you to view a longer note about that particular event. Times and dates need more explanation. Pest can accept times in 12 or 24-hour format and recognises the difference by the presence of AM/PM in the string. For example:

```
1:00 and 1:00am
```

are the same thing, just as are:

```
1:00pm and 13:00
```

The 24-hour method is less characters to type – it's entirely up to you. Dates and days are more complex – although you only have to use them if you want to specify an event for some particular day. Basic dates are simple, for example let's assume you have a dental checkup on 23rd July 1993, you would set an instant event (no time) dated as 23-Jul-93.

More regular occurrences (such as birthdays and anniversaries) are entered similarly – but without the year: the pest just registers the date and the month, viz.: 23-Jul would define a single dated event every 23rd July regardless of the year: birthdays and so on are a good candidate for this event. This technique can be extended to cover every day in a month such as Jan. Months and dates can be grouped using the bar (|) character – this is explained for days below.

Now let's imagine you have to pick the kids up at 3:00 every weekday and it takes you ten minutes to get to school, you might set the event day as: Mon-Fri. Pest recognises this special string and enters the string: Mon|Tue|Wed|Thu|Fri. Specific days should be entered as three letter strings, separated by bars if necessary, otherwise the entire week is taken as default.

Timed Events vs Delayed Events

Most Pest events are constructed from a time + day and/or date plus a message. You can opt for a reminder to occur as you boot the machine or for something to appear after a short delay – say 60 minutes. The timed delays are useful to remind you to take a break although a reset is required to re-trigger them.

How Pest Works

Pest stores its events in the S: assignment in a script titled "Pestfile" where every event takes a single line in the file. You can permanently remove an event by deleting the line or disable it by placing a ";" at the beginning of the line. You can edit this file directly and call the relevant Pest commands as you see fit, there is no requirement to use AddPestEvent or DeletePestEvent – that software is provided for beginners.

The system is constructed from a hideously complex set of AmigaDOS algorithms – but just four simple commands are central to the operation of the entire system; and only one of those is specific to AmigaDOS 3. Pest keys on dates, days and times shown by the DATE command like this:

```
1>date Monday 19-Apr-93 10:57:03
```

Provided your machine has a battery backed, real-time clock this date will be correct every time you switch on or reboot. By comparing the date portion to a known date, Pest can determine whether to set an event. The burning question is how can any day, date or combination be tested for? The entire test is performed in a single line command based on SEARCH: and this is what makes the system so powerful.

SEARCH can test for the presence of a sub-string within a file and report on its findings. Let's see this in action:

```
1>SEARCH S:SPAT FAILAT
 8 FailAt 21
18 FailAt 10
```

The command displays the lines the string was found on and their line numbers. You don't have to specify a full word of course:

```
1>SEARCH S:SPAT FAIL
 8 FailAt 21
12 IF NOT FAIL
18 FailAt 10
```

As you can see an extra line also containing the string "FAIL" appears. This is fine when used from a Shell, but useless when used in a script. Someone thought of that: and SEARCH returns an indication, in the system variable "RC", to say whether the string was found or not. Here is an example run:

```
1>SEARCH >NIL: S:SPAT FAIL
1>GET RC
0
1>SEARCH >NIL: S:SPAT FUDGE
1>GET RC
```

```
5
```

In the second instance, the string "FUDGE" could not be found in the file, so SEARCH sets the variable RC to 5. (This variable can be easily tested and acted upon with IF.)

Now how about our dates? The date itself can be easily written to disk like this:

```
1>DATE >DateFile
```

The ">" is a re-direction operator – it takes everything that would normally be displayed on screen and writes it to a named file – "DateFile". Ordinarily this file would be written in the Ram Disk (T: assignment) but to avoid confusion just take that for granted at this stage.

We can now use SEARCH to quickly test if the requested date is present in the date file like this:

```
1>SEARCH DateFile "2-Jun-93"
```

If the date is the 2nd June, SEARCH returns RC=0 (OK) otherwise it returns RC=5 (WARN). Simple enough, but remember how SEARCH does not differentiate between words and parts of words – what would happen in this case?

```
1>SEARCH DateFile "Jun"
```

The search is true if any part of the date contains the word "Jun", or, as Pest sees it, the date is *any* day in June of *any* year. Exactly the same method can be applied to days of the week too:

```
1>SEARCH DateFile "Monday"
```

This search will be true (OK, RC=0) whenever the day name is Monday and false (WARN, RC=5) at any other time.

Now for the 65-million dollar question: what happens when an event should be scheduled for more than one day or date? The immediately obvious solution is to set an event for each day required – but this is wasteful. SEARCH offers a less obvious, but far more elegant answer: pattern matching. The special bar character "|" pronounced "OR" solves this. In longhand if you want and event to activate on a Monday, Wednesday and Friday this would reduce to: "Mon OR Wed OR Fri" and further sublimate to its AmigaDOS equivalent, "Mon|Wed|Fri" thus:

```
1>SEARCH DateFile "Mon|Wed|Fri" PATTERN
```

Note the PATTERN switch used on the command line in this example tells SEARCH to interpret the string as a pattern (group) rather than a literal string.

All of this translates into a few lines of code which introduce the SetPestEvent and SetWaitEvent scripts and quickly determine whether or not to bother with a specific event. With the dates taken care of there is something else that can affect a Pest event: time.

Most events are set to appear at a specific time, but it is possible the time has already passed when the machine is booted, this is also taken care of and you have the option to view the event or even wait until the next day!

Line-By-Line

Pest is the largest suite of related scripts in the book: and AddPestEvent is the largest of those by a long way. To keep things to a reasonable size, the line breakdown of this script is necessarily succinct. You don't have to understand how it works to use it! AddPestEvent uses a great number of conditional tests to ensure it runs at a reasonable speed.

1-3. Define the key and reset bra and ket. Note that AddPestEvent does not have any required arguments. That's simply because it is meant to be run by IconX from Workbench – more of that shortly.

4-6. Determine if the script has already been run once (has already intilialised) and if so, skip to Step 43.

7-10. Initialise some local variables according to the user's input.

11. If the user has entered some message, control jumps to Step 42, otherwise it continues at Step 12.

12-42. Form the interactive help system based around RequestChoice. The flow of this part is not critical. RQ always contains 0 if OK is pressed; a number otherwise.

43. Marks the jump point for Step 5.

44-46. Looks for the special string "Mon-Fri" and converts it into a Pest (SEARCH) readable format. The result is held in the local "Day".

47-51. If no day has been supplied, the day is converted to every day of the week and stored in "Day". If a special string has been supplied, this is copied directly into "Day".

52-55. Determines if the event is timed or not. If not, an Instant event is set and control jumps to Step 104.

56-60. Checks for a ":" in the time string. If the colon is not found, control resumes at Step 23 (Help).

61-63. Tests if the time has been entered in 12 hour or 24 hour format. If 24 hour format is being used, control resumes at Step 64, otherwise it jumps to Step 70.

64-69. Parse the hours and minutes from the 24-hour time string, places them in the locals "hrs" and "mns" and jumps to Step 94.

70-78. Parses the hours, minutes and am/pm information from

the time string.

79-82. Validates the correct number of hours (no more than 12 for am/pm format) and exits the script on error.

83-93. Converts the hour part of 12-hour format into 24-hour format by adding 12 to the hours value. Note that the intermediate result is stored in a global variable.

94-98. Validates 24-hour (hours) format and exists if an error is found.

99-102. Validates no more than 59 minutes in either 12 or 24-hour formats and exits if an error is found.

103. Concatenates the hours and minutes in a format suitable for use with WAIT and saves the result in the local "EventTime".

104-117. Confirm and check the event settings. If the user is happy, these lines append the event to the PestFile script. A new one is created automatically.

118-132. Bail out and say goodbye!

133-172. Form the interactive response part of the program. Flow here is determined by user responses and variables are recalled by Pest's "GetArgs" support script.

173. Calls the program recursively – telling it the preamble (initialisation) is complete.

Listing

```
1.  .key Time/K,Day/K,Wait/K,Go/K,Message/F
2.  .bra {
3.  .ket }
4.  if "{go}" EQ "Now"
5.   skip DoItNow
6.  endif
7.  set msg {Message$"Boing... AmigaDOS Pest calling!"}
8.  set ArgTime {Time}
9.  set ArgDay {Day}
10. set ArgWait {Wait}
11. if "{Message}" EQ ""
12.  RequestChoice >ENV:RQ{$$} "Pest" "The Pest V3*nDesigned
    by Mark Smiddy" "Start" "Help" "Quit"
13.  if VAL $RQ{$$} EQ 0
14.  echo "When I grow up, I want to be a 555"
15.  skip out
```

```
16.  endif
17.  if VAL $RQ{$$} EQ 1
18.  skip Interactive
19.  endif
20.  if VAL $RQ{$$} EQ 2
21.  skip help
22.  endif
23.  lab help
24.  RequestChoice >ENV:RQ{$$} "Pest Help" "Give a time or
    delay*nDay/Dates are optional*nSelect a button for more
    help..." "CLI" "Times" "Dates" "Delays" "OK"
25.  lab again
26.  if VAL $RQ{$$} EQ 0
27.  skip out
28.  endif
29.  if VAL $RQ{$$} EQ 1
30.  RequestChoice >ENV:RQ{$$} "Pest CLI" "Message=*"Message
    to display*"*n*nTime=Time to activate (none=at
    reboot)*n*nWait=a number of
    minutes*n*nDate=Date(s)/Day(s) to activate on" "CLI"
    "Times" "Dates" "Delays" "OK"
31.  endif
32.  if VAL $RQ{$$} EQ 2
33.  RequestChoice >ENV:RQ{$$} "Pest Time" "Time formats are
    12 or 24Hour*n*neg 13:00, 09:00, 9:12am, 12:51pm" "CLI"
    "Times" "Dates" "Delays" "OK"
34.  endif
35.  if VAL $RQ{$$} EQ 3
36.  RequestChoice >ENV:RQ{$$} "Pest Dates" "Date formats are
    full or partial dates +or+ days*n*nEg Saturday, Mon, 12-
    Oct-93, 14-Feb-, -Mar-93 etc.*n*nPatterns may be used,
    ie: Mon|Wed|Sat *n*nMon-Fri is Monday to Friday" "CLI"
    "Times" "Dates" "Delays" "OK"
37.  endif
38.  if VAL $RQ{$$} EQ 4
39.  RequestChoice >ENV:RQ{$$} "Pest Delay" "NB: Times and
    Delays don't mix*n*nSpecify a delay in minutes
    only*n*nEvent occurs in minutes after most recent
    reset*n*n(Dates/Days can be specified)" "CLI" "Times"
    "Dates" "Delays" "OK"
40.  endif
41.  skip again back
42. endif
```

```
43. lab DoItNow
44. if "$ArgDay" EQ "Mon-Fri"
45.  Set Day "Mon|Tue|Wed|Thu|Fri"
46. endif
47. if "$ArgDay" EQ ""
48.  Set Day "Sun|Mon|Tue|Wed|Thu|Fri|Sat"
49. else
50.  Set Day $ArgDay
51. endif
52. if "$ArgTime" EQ ""
53.  Set EventTime "<Instant>"
54.  Skip NoTime
55. endif
56. setenv edt{$$} "$ArgTime"
57. search >NIL: env:edt{$$} ":"
58. if warn
59.  skip help back
60. endif
61. setenv edt{$$} "$ArgTime"
62. search >NIL: env:edt{$$} (am|pm) pattern
63. if warn
64.  echo >env:edt{$$} "0$ArgTime" len=5
65.  echo >env:hrs{$$} "$edt{$$} " first=1 len 2
66.  echo >env:mns{$$} "$edt{$$} " first=4 len 2
67.  set hrs $hrs{$$}
68.  set mns $mns{$$}
69.  skip 24Hour
70. else
71.  echo >env:edt{$$} " $ArgTime" len=7
72.   echo >env:hrs{$$} "$edt{$$} " first=1 len 2
73.   echo >env:mns{$$} "$edt{$$} " first=4 len 2
74.   echo >env:apm{$$} "$edt{$$} " first=6 len 2
75.   set hrs $hrs{$$}
76.   set mns $mns{$$}
77.   set apm $apm{$$}
78. endif
79. if val $hrs GT 12
```

```
80.   RequestChoice >NIL: "Pest" "Only 12 hours in Am/Pm for-
    mat" "OK"
81.   skip out
82. endif
83. if "$apm" EQ "pm"
84.   if val $hrs NOT GT 11
85.    eval ($hrs + 12) mod 24 to env:hrs{$$}
86.    set hrs $hrs{$$}
87.   endif
88. endif
89. if "$apm" EQ "am"
90.   if val $hrs EQ 12
91.     set hrs 0
92.   endif
93. endif
94. lab 24Hour
95. if val $hrs GT 24
96.   RequestChoice >NIL: "Pest" "Only 24 hours in 24-hour
    clock" "OK"
97.   skip out
98. endif
99. if val $mns GT 59
100.  RequestChoice >NIL: "Pest" "Error: there are only 60
    minutes in an hour!" "OK"
101.  skip out
102.endif
103.set EventTime $hrs:$mns
104.Lab NoTime
105.if "$ArgWait" EQ ""
106. requestchoice >ENV:RQ{$$} "Pest" "Confirming:
    *"$ArgMsg*"*n*nAt $EventTime*n*non/during/every $Day"
    "OK" "Forget It"
107.else
108. requestchoice >ENV:RQ{$$} "Pest" "Confirming:
    *"$ArgMsg*"*n*n$ArgWait mins from
    startup*n*non/during/every $Day" "OK" "Forget It"
109.endif
110.if VAL $RQ{$$} EQ 0
111.  skip out
```

```
112.endif
113.if "$ArgWait" EQ ""
114. echo >>S:PestFile "PEST3:SetPestEvent
    Time=*"$EventTime*" day=*"$Day*" Message=*"$ArgMsg*""
115.else
116. echo >>S:PestFile "PEST3:SetWaitEvent Wait=*"$ArgWait*"
    day=*"$Day*" Message=*"$ArgMsg*""
117.endif
118.lab out
119.unset EventTime
120.unset Day
121.unset ArgDay
122.unset ArgWait
123.unset ArgTime
124.unset hrs
125.unsetenv hrs{$$}
126.unset mns
127.unsetenv mns{$$}
128.unset apm
129.unsetenv apm{$$}
130.unsetenv RQ{$$}
131.echo "TTFN - from the Pest!"
132.quit
133.lab interactive
134.set ArgWait ""
135.set ArgTime ""
136.set ArgDay ""
137.Ask "Use default message?"
138.if warn
139. echo "Using default message..."
140. Set ArgMsg $Msg
141.else
142. echo "Enter the message"
143. PEST3:GetArgs Msg
144.endif
145.Ask "*e[32mDo you require a timed event?*e[31m"
146.if warn
147. echo "Enter time (format HH:MM)"
```

```
148.  PEST3:GetArgs Time
149.else
150.  Ask "*e[32mIs this a delay event?*e[31m"
151.  if warn
152.    echo "Enter minutes to wait"
153.    PEST3:GetArgs Wait
154.    set ArgTime ""
155.  else
156.   echo "Event will occur upon restart."
157.  endif
158.endif
159.Ask "*e[32mIs this event for a specific date?*e[31m"
160.if warn
161.  echo "Enter date (format DD-MMM-YY) partial dates
    accepted (see Help)"
162.  PEST3:GetArgs Day
163.else
164.  echo "Event scheduled for daily use."
165.  Ask "*e[32mDo you wish to specify a day or days?*e[31m"
166.  if warn
167.   echo "Enter day or days using first three
    letters.*nSpecial formats are accepted - see help"
168.   PEST3:GetArgs Day
169.  else
170.   echo "Event scheduled for every day."
171.  endif
172.endif
173.PEST3:AddPestEvent Go=Now
```

Pest 3: ChangePestMessage

Synopsis: [[Event=]Event #] [[Message=]"Text"]

Template: Event,message

Path: SYS:Pest3

Requires: V3+

See also: AddPestEvent, DeletePestEvent, KillPestEvent, StartPest, GetArgs, DeletePestEvent, SetPestEvent, SetWaitEvent, ListPestEvents

Type: IconX script

Brief: Changes the message attached to a pest event

Description

ChangePestMessage: was provided because the problem was there. Although I have found little use for it, I suspect someone will like it. Essentially you are provided with a list of all the current events (those already running) and are given the opportunity to change the message attached to any one.

```
List Events © No Batteries Required
Pest active: Friday 07-May-93 18:23:47
Event     Time      Status        Message
0.        5:M       ++Active++    Boing... AmigaDOS Pe
1.        5:M       ++Active++    Hello There!
Enter event number to change:2
Enter new message for event 2:This is a new Message
```

Pest Change Message

The script will usually be called from Workbench (using IconX) but it doesn't have to be. If called from Shell without arguments, it behaves like the Workbench version, otherwise you can supply an event number and a new message, viz:

```
1>ChangePestMessage 3 "I've changed this message!"
```

Important: ChangePestMessage does not alter "PestFile" it only affects a current event.

Line-By-Line

1. Defines a simple template for the script. Note that neither of the arguments are required. This makes it possible for the script to be executed from IconX safely.

2-3. Change the default < and > markers to { and }.

4. Checks if an event number has been supplied. If not, control continues at Step 5; otherwise it is transfers to Step 12. This provides the interactive feature for the event number.

5. Calls ListPestEvents to show the current events with their attached event number. This can be handy from Shell if you don't know which event number is attached to a particular event.
6. Provides a simple message. Note that the newline character is suppressed.
7. Sets the local ArgEvt to some arbitrary value. This prevents any blobs if the user doesn't supply some value at Step 8.
8. Calls GetArgs asking it to return a value in ArgEvt. See the description of GetArgs to see how this is done.
9. This calls ChangePestMessage recursively. Note the event number is set by ArgEvt on this call.
10. When the recursion unwinds, this line clears the value in ArgEvt.
11. Jumps to the end of the script (Step 27) and exits cleanly.
12. Terminates the IF...ENDIF construct opened at Step 4.
13. Tests if a message has been supplied. If the message is empty, control continues at Step 14; otherwise it jumps to Step 20.
14. Displays a newline plus a message asking for the new message. Note the final newline is suppressed to keep the input looking logical.
15. Sets a dummy value for ArgMsg (the message string). This prevents any blobs if the user doesn't supply some value at Step 16.
16. Gets a new message from the user and returns the result in ArgMsg. Note that if no value is supplied, the message defaults to "+++".
17. Calls ChangePestMessage recursively with all the required information.
18. When the script unwinds, this clears the ArgMsg variable...
19. ...and exits this level cleanly by jumping to Step 27.
20. Terminates the IF...ENDIF construct opened at Step 13.
21. Tests for an event running using the supplied Pest event number. This will be a resident copy of WAIT numbered according to its event number. See SetPestEvent and SetWaitEvent to see how this is achieved.
22. STATUS sets the WARN condition if it could not find the requested command. This checks the return and branches to Step 23 if the event was present or Step 24 if it wasn't.
23. Changes the indirection variable "PM+Event #" to the new message. See SetPestEvent to see how this is used.
24. If control reaches here from Step 23 it jumps to Step 26;

otherwise it continues at Step 25.

25. This gives a puzzled error message if the event number could not be found. Typically this means you have not supplied a correct event number, either interactively or at the command line. (This applies if you don't supply something to GetArgs too.)
26. Terminates the IF...ELSE...ENDIF construct opened at Step 22.
27. Marks the end of the script: primarily for the recursive jumps.

Listing

```
1.  .key Event,message
2.  .bra {
3.  .ket }
4.  if "{Event}" EQ ""
5.   PEST3:ListPestEvents
6.   echo "*nEnter event number to change:" noline
7.   set ArgEvt "<Non Existant>"
8.   PEST3:GetArgs Evt
9.   ChangePestMessage Event=$ArgEvt
10.  unset ArgEvt
11.  skip end
12. endif
13. if "{Message}" EQ ""
14.  echo "*nEnter new message for event {Event}:" noline
15.  Set ArgMsg "+++"
16.  PEST3:GetArgs Msg
17.  ChangePestMessage Event=$ArgEvt Message="$ArgMsg"
18.  unset ArgMsg
19.  skip end
20. endif
21. status >NIL: command=Wait{Event}
22. if not warn
23.   setenv PM{Event} "{Message}"
24. else
25.   Echo "Error: That event does not seem to exist?"
26. endif
27. lab end
```

Pest 3: DeletePestEvent

Synopsis: Usually executed from Workbench

Template: exits)

Path: SYS:Pest3

Requires: V3+

See also: AddPestEvent, KillPestEvent, StartPest, GetArgs, ListPestEvents, ChangeMessage, SetPestEvent, SetWaitEvent

Type: IconX script

Brief: Removes a Pest 3 event: permanently

Description

DeletePestEvent permanently removes any pest event from the system and should be used with care. It works along similar lines to KillPestEvent but this one actually lists the main "Event" script (Pestfile). Enter the number of the line to remove or "0" to escape. If executed from the Shell, this program is best used without arguments to prevent any errors.

```
    1 PEST3:SetPestEvent Time="16:55" day="Mon|Tue|Wed|Thu|Fri" Message="Star
 Trek on Sky 1 in 5 mins" quiet
    2 PEST3:SetWaitEvent Wait="5" day="Sun|Mon|Tue|Wed|Thu|Fri|Sat" Message="
Boing... AmigaDOS Pest calling!"
    3 PEST3:SetWaitEvent Wait="3" day="Mon|Wed|Sat" Message="Master Scripts w
ith this amazing book!"
Enter event number to remove (0 exits):
```

Pest Delete Event

Line-By-Line

1. DeletePestEvent uses a interactive, recursive technique to get its command line option. Part of the prompt text is "(0 exits)" which is why the strange bracket "exits)" appears here.

2-3. Define the angle brackets as braces.

4. If the user enters 0 at the command line, control continues at Step 5; otherwise it jumps to Step 6.

5. Stops the script and returns back to AmigaDOS. (Quit cannot be called from a nested script by the way.)

6. Terminates the IF...ENDIF construct opened at Step 4.

7. Checks that some input has been made. If not, control continues at Step 8.

8. Types the current event program with line numbers.

9. Displays the main part of the interactive prompt: note there's an extra space before the final closing quote – this ensures the message is constructed correctly.
10. Calls DeletePestEvent recursively with interactive mode (?). This displays the last part of the prompt and waits for the user to enter something.
11. When the recursion unwinds to this point, it jumps to the end and exits cleanly.
12. Closes the IF...ENDIF construct at opened at Step 12.
13. Decrements the value of the input and stores the result in the global, "Line".
14. Tests if the value of "Line" is less than 1. If it is, control resumes at Step 15; otherwise it jumps to Step 16.
15. Creates a simple EDIT macro to delete the first line EDIT encounters.
16. If control reaches here from Step 15, it jumps to Step 18; otherwise it continues at Step 17.
17. Creates a simple EDIT macro to jump forward "Line" number of lines then delete the next one.
18. Closes the IF...ELSE...ENDIF construct opened at Step 14.
19. Deletes the requested line by editing the current PestFile (event program) with the macro created by Step 15 or 17. Note, this number is not range checked and if a large number is used, EDIT will stop and complain. You might like to experiment with some error checking here. Hint: See the DataBase program for more information.
20. Marks the "out" point for bailout.
21. Clears the global variable, Line.
22. Removes the EDIT macro created above. All scripts should do this, but very few people ever bother (including me) because this takes time and is rarely necessary.

Listing

```
1.  .key exits)
2.  .bra {
3.  .ket }
4.  if "{exits)}" EQ "0"
5.   quit
6.  endif
7.  if "{exits)}" EQ ""
```

```
8.    type s:pestfile number
9.    echo "*nEnter event number to remove (0 " noline
10.   PEST3:DeletePestEvent ?
11.   skip out
12.  endif
13.  eval {exits)} - 1 to=env:Line
14.  if VAL $Line NOT GE 1
15.   echo >T:AE{$$} "d"
16.  else
17.   echo >T:AE{$$} "$Line(n);d;"
18.  endif
19.  edit s:PestFile with t:AE{$$}
20.  lab out
21.  unsetenv Line
22.  delete >NIL: T:AE{$$}
```

Pest 3: GetArgs

Synopsis: [EXECUTE] GetArgs <[Argnum=]#> [lc=private] [[Arg=]->]

Template: ArgNum/a,lc/K,Arg/F

Path: SYS:Pest3

Requires: V3+ (for Pest 3)

See also: AddPestevent, DeletePestEvent, KillPestEvent, StartPest, DeletePestEvent, ChangeMessage, SetPestEvent, SetWaitEvent, ListPestEvents

Type: Script

Brief: Argument retrieval for the Pest v3

Description

A central part of Pest 3 is its ability to interactively read a line of text from the user and return the result to another script. A deceptively short script achieves this with a minimum of fuss, GetArgs works like this. You send it a variable name (or number) and it returns a local variable (Arg<Name>) containing the result. It is similar to the BASIC command:

```
LINE INPUT A$
```

For instance, the command "GetArgs Time" returns its result in "ArgTime". The word "Arg" is appended to ensure there are no clashes with existing variables. The script operates with a recursive algorithm which is enough to make your brain itch until you get the hang of the idea. Here's how it works – don't panic if you don't get the idea right away:

Line-By-Line

1. Defines the argument template with ArgNum required and two optional arguments: note ARG is Final (/F). We'll see their function shortly – but you might like to consider, this script will work equally well if "LC" was a switch (/S) by changing Line 5 slightly. Before you read the rest of this, try to predict why.

2-3. Redefines BRA and KET to my favourite versions.

4. Opens an IF...ENDIF construct to check if an argument has been supplied *or* the script is being run for the second time. If either is true, execution jumps to Step 6; otherwise it continues at Step 5.

5. This line recursively calls GetArgs again – the argument

number is passed back (it's required) and LC (last chance) is turned on. LC could be a switch in the template in which case the "=ON" would not be required, either way works as well. More importantly, this command places the argument parser into interactive mode and sinks the argument template to NIL:. The result of this is to give the user somewhere to type without printing a useless message. Whatever they type is passed directly back to "ARG" in the second recursive invocation of GetArgs. Since this argument is Final, everything including any spaces is passed into the argument.

6. Terminates the IF...ENDIF construct opened at Step 4. This line is only reached when the script has done one complete recursive loop.

7. Defines a local environmental variable ARG<ArgNum> with the value held by Arg. The script then terminates unless the script was called from within itself (Step 5) in which case execution resumes effectively at Step 6 in the original.

Listing

```
1.  .key ArgNum/a,lc/K,Arg/F
2.  .bra {
3.  .ket }
4.  if "{Arg}{lc}" EQ ""
5.   PEST3:getargs >NIL: ArgNum={ArgNum} LC=ON ?
6.  endif
7.  Set Arg{Argnum} {Arg}
```

Pest 3: KillPestEvent

Synopsis: [EXECUTE] KillPestEvent <[Event=]#> [[Message=]Text] [[Sys=]private]

Template: Event/a,message,sys

Path: SYS:Pest3

Requires: V3+ (as part of Pest 3)

See also: AddPestevent, DeletePestEvent, KillPestEvent, StartPest, GetArgs, DeletePestEvent, ChangeMessage, SetPestEvent, SetWaitEvent, ListPestEvents

Type: Script

Brief: List the current events tracked by the Pest

Description

Removes a current running event from the list. This command is designed to be used from Workbench and may be used without arguments to trigger its interactive mode. The Message argument is reserved for use by the Pest system although you can supply one if you wish. When an event is removed, the "++Active++" string is replaced by the contents of the message argument. In any case, when KillPestEvent terminates an event, it echoes the process slot used (check STATUS). Examples:

```
1>KillPestEvent 1
Running as process:10
Bang! Event 1 bites the dust
1>KillPestEvent
Pest active Monday 27-Apr-93 11:30:23
Event Time Status  Message
0.   12:59 ++Active++   Time for lunch!!
1.   14:47 +Deceased+
Enter event number to delete:
```

Line-By-Line

1-3. Define the argument template and re-define the bra and ket characters.

4. Sets the default message parameter. This occurs when KillPestEvent is called by a user – SetPestMessage sends its own message.

5. Tests if an event number has been supplied. If not, control

continues at Step 6, otherwise it jumps to Step 12. This is primarily to keep things interactive from Workbench.

6. Calls ListPestEvents to display the current events. Note that ListPestEvents shows "dead" or completed events too. You can't kill these since they have already been removed from the system.
7. Prints the prompt for GetArgs to use. The NOLINE option is used to suppress the extra linefeed.
8. Gets the number of the event to delete from the user and returns the result in the local, ArgEvt.
9. Calls KillPestEvent (itself) recursively but this time with the correct event number inserted at the command line.
10. Clears the ArgEvt variable.
11. Skips to the end of the script and exits when the recursion unwinds.
12. Terminates the IF...ENDIF construct opened at Step 5.
13. Checks the status of a WAIT command numbered by the event number. If this exists it is sent to the file "T:Kill", if not the WARN condition is set.
14. Checks if the WARN flag was clear (if the WAIT exists). If it is, execution continues at Step 15, otherwise it jumps to Step 19.
15. Displays the first part of a progress message with the newline character suppressed so...
16. ...the process number appears correctly. This feature is not strictly necessary but it makes things look professional.
17. Uses an interactive break to stop the WAIT event checked at Step 5. Re-direction to NIL: prevents BREAKS argument template appearing and messing up the display.
18. This confirms the event has been deleted.
19. Execution arrives here if the WAIT event was not found and continues...
20. ...here, where it prints an error. (This error is not displayed by SetPestEvent, even though it occurs.)
21. Checks if the script was called by the Pest system (SYS<>"") and if so, execution jumps to Step 23. Otherwise it continues at Step 22.
22. Forces execution to jump to the label at Step 29.
23. Terminates the IF...ENDIF construct opened at Step 21.
24. Closes the IF...ELSE...ENDIF construct opened at Step 14
25. Writes an EDIT script – here it is in longhand:

```
1.  F/{Event}./
2.  PA /:/
3.  PA /[I/
4.  15#
5.  B//{Message}/
```

or

```
1.  Find the line starting with the event number.
2.  Move the cursor after the “:” in the event time.
3.  Move the cursor after the next TAB.
4.  Delete 15 characters
5.  Insert the message at the current position plus a
    tab.
```

26. Checks to make sure the Event global is available.
27. Replaces the "++Active++" message in the global with the message defined at the command line. SetPestEvent sends "Completed" by default.
28. Clears the message attached to the requested event number.
29. Terminates the IF...ENDIF construct opened at Step 26.
30. This marks an exit point if something has gone wrong earlier in the script. It is ignored otherwise.
31. Deletes the temporary file and frees up some memory.

Listing

```
1.  .key Event,message,sys
2.  .bra {
3.  .ket }
4.  .def Message "+Deceased+"
5.  if "{Event}" EQ ""
6.   PEST3:ListPestEvents
7.   echo "*nEnter event number to delete:" noline
8.   PEST3:getargs Evt
9.   PEST3:KillPestEvent $ArgEvt
10.  unset ArgEvt
11.  skip end
12. endif
13. status >T:Kill{$$} command=Wait{Event}
14. if not warn
15.  echo "Running as process:*e[32m" noline
```

```
16.  type T:Kill{$$}
17.  break <T:kill{$$} >NIL: all ?
18.  echo "*e[31mBang! Event: {Event} bites the dust"
19. else
20.  echo "Error: That event has not been set?"
21.  if "{sys}" EQ ""
22.   skip end
23.  endif
24. endif
25. echo >T:Kill{$$} "F/{Event}./;pa/:/; pa/[I/;15#;
    B//{Message}/;"
26. if exists env:PV{Event}
27.  edit env:PV{Event} with T:Kill{$$}
28.  unsetenv PM{Event}
29. endif
30. lab end
31. delete >NIL: T:kill{$$} quiet
```

Pest 3: ListPestEvents

Synopsis: Usually used from Workbench

Template: QUICK/S

Path: SYS:Pest3

Requires: V3+

See also: AddPestevent, DeletePestEvent, KillPestEvent, StartPest, GetArgs, DeletePestEvent, ChangeMessage, SetPestEvent, SetWaitEvent

Type: IconX script

Brief: List the current events tracked by the Pest

Description

This function is provided to list information on all the current events. The QUICK switch is available from the Shell only and is used to suppress the date heading and message output.

Pest List Events

ListPestEvents is used to list the current events and their status. Note: this utility only affects *running* events scheduled for that day. They can be reset by re-booting the machine – see DeletePestEvent for a more permanent solution. Three general states are possible:

- Active: event is running and waiting to execute.
- Deceased: event has been removed by the user (with KillPestEvent) before completion.
- Complete: event has already timed out normally.

An event may be removed by calling KillPestEvent with a special message. In this case the "Status" code will reflect that.

Line-By-Line

1-3. Defines the argument template as described above and sets the angle brackets to braces.

4. Tests if the QUICK option was supplied. If it was control jumps to Step 6; otherwise it continues at Step 5.

5. Prints the "QUICK" header for ListPestEvents.
6. If control reaches here from Step 5 it jumps to Step 9; otherwise it continues at Step 7.
7. Displays the current date and time as a part of the header message.
8. Displays the full "message-included" event header.
9. Terminates the IF...ELSE...ENDIF construct opened at Step 4.
10. Creates a global variable called "Count#" and sets it to zero. ({$$} shown as # here is the current process number.)
11. Marks the start of a loop.
12. Attempts to get the value of the Pest internal variable, suffixed by the loop counter, Count#. Events start at 0 and count up from there. The actual value of the variable is sent to NIL: at this point, we want to test for the presence of the variable. For example, if three events were (or had been) set, there would be three PVs: PV0, PV1 and PV2. When the number of current events is exceeded, GETENV returns WARN.
13. Tests if the WARN flag was present and...
14. ...jumps out of the loop and the script (to Step 30).
15. Closes the IF...ENDIF construct opened at Step 13.
16. Uses "`" (expand command) to display the current value of PVn: where n is the number held in "Count". (It has to be done this way: PVCount{$$} will confuse the dollar parser.)
17. Tests if the QUICK switch was used at the command line. If it was, control continues at Step 18; otherwise it jumps to Step 19.
18. Displays a blank line.
19. If control reaches here from Step 18 it jumps to Step 26; otherwise it continues at Step 20.
20. Tests for the presence of a message variable using the technique described at Step 12. Completed messages do not have a message attached to them. (The message is removed by KillPestEvent.)
21. Tests if the variable was valid. If it was control jumps to Step 24; otherwise it continues at Step 22...
22. ...and displays another blank line (finishing off the event details).
23. Jumps directly to Step 27, does not pass GO and does not collect £200.
24. Terminates the IF...ENDIF construct opened at Step 21.
25. Retrieves the message from the appropriate variable (see Step16) and displays the first 20 characters of it. This stops long messages from wrapping on lines and messing up the display.
26. Terminates the IF...ELSE...ENDIF construct opened at Step 17.

27. Marks the entry point for the next loop of the script...
28. ...where the counter is incremented by 1.
29. Jumps back to the start of the loop (at Step 11) and goe through the whole process again.
30. When the script needs to bail-out, it jumps to this point.

Listing

```
1.  .key QUICK/S
2.  .bra {
3.  .ket }
4.  if "{quick}" NOT EQ ""
5.   echo "Event*e[ITime*e[IStatus"
6.  else
7.   echo "Pest active: `date`"
8.   echo "Event*e[ITime*e[IStatus   *e[IMessage"
9.  endif
10. setenv count{$$} 0
11. lab loop
12. getenv >NIL: PV$Count{$$}
13. if warn
14.  skip all_done
15. endif
16. echo "`getenv PV$Count{$$}`" noline
17. if "{QUICK}" NOT EQ ""
18.  echo ""
19. else
20.  getenv >NIL: PM$Count{$$}
21.  if warn
22.   echo ""
23.   skip next
24.  endif
25.  echo "`getenv PM$Count{$$}`" first=1 len=20
26. endif
27. lab next
28. eval $Count{$$} +1 to ENV:Count{$$}
29. skip loop back
30. lab all_done
```

Pest 3: SetPestEvent

Synopsis: [EXECUTE] [[Time=]time|date] [[Message=]"Msg"] [Day=Dayname] [QUIET]

Template: time/a,Message,day/k,QUIET/S

Path: SYS:Pest3

Requires: V3+

See also: AddPestEvent, DeletePestEvent, KillPestEvent, StartPest, GetArgs, DeletePestEvent, ChangeMessage, SetWaitEvent, ListPestEvents

Type: Pest support script

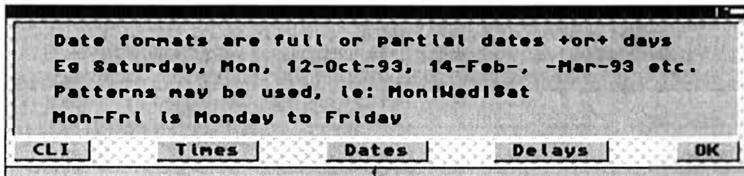
Brief: Sets timed events for Pest v3

Description

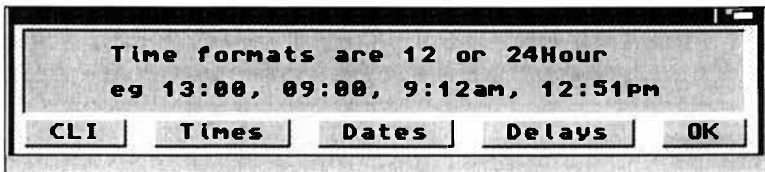
This command sets the normal time/date events and is normally executed by StartPest (via the PestFile). You can call it directly to set an immediate event which does not require a re-boot however. This command is stripped down for speed and times should be entered in 24-hour clock only! If the message contains spaces it should be enclosed by quotes. You can use this command with care to add events directly to S:PestFile if you wish. Example:

```
1>SetPestEvent Time=13:00 Message="Time for a break"
```

The QUIET switch is used from the Shell or PestFile directly to suppress "override" messages when an event has been missed.



Pest Date Help



PestTime Help

Line-By-Line

1-3. Sets the input key and set the bracket characters to { and }.

Day Fragment (4-10)

4. {Day} is an optional argument supplied by the user as defined above which carries the day(s) *or* date(s). This test determines if an argument has been supplied by checking if "{day}" and "" are *not* the same. (AmigaDOS pre-parses the script by replacing every argument name with any values supplied for it.) Two possible ways this line can be expanded are:

```
A.  if not eq ""
```

or

```
B.  if Mon|Wed|Fri not eq ""
```

When such a test evaluates TRUE, execution continues at Step 5. If, on the other hand, the test returns FALSE, execution jumps to the matching ENDIF (Step 10). In this first case, things are slightly more complex because the test is reversed by "NOT". The unknown string on the left is compared for inequality to the empty string on the right: Test A evaluates FALSE; B evaluates TRUE.

5. This sends the current time and date to a temporary file. The name is arbitrary, but use of the "T:" assignment ensures the file is written to RAM for speed.

6. Here is the practical version of the search described in AddPestEvent. Although this line looks complex, all that has been added is the day/date argument collected from the command line. When AmigaDOS expands this it might look like this:

```
search >NIL: t:TPToday "(Mon|Wed|Fri)" pattern
```

(Brackets and quotes are added here to prevent the command becoming confused if the date argument contains spaces.) If the date file, TPToday, contains one of the sub-strings: Mon, Wed or Fri, SEARCH returns OK; otherwise it returns WARN.

7. Checks if the SEARCH set the WARN flag – the date/day sub-string was *not* located in today's date. If WARN was found, execution continues at Step 8, otherwise it jumps to Step 9.

8. Immediately transfers control to the end of the script and exits to save time.

9. Marks the closing point for the IF...ENDIF construct at Step 7. Execution continues at Step 10.

10. Closes the IF...ENDIF construct from Step 4 and allows execution to continue with the remainder of the script.

Instant Fragment (11-14)

11. Checks if the variable "time" is equal to the special string "<Instant>". This is a private string usually set by AddPestEvent.

12. If an instant event has been selected, this presents a requester titled "Pest" with the message string and a single "OK" button. The return from RequestChoice is sent to the global, RQ although it is never actually used in this context. NIL: could be used here if preferred.

13. Jumps to the end of the script and exits quickly.

14. Terminates the IF...ENDIF construct from Step 11. Control only arrives here if a non-instant event is being set.

Time Fragment (15-31)

15. Writes the date and time to a global environmental variable.

16. Constructs an EDIT macro to extract the time from the current date. This is quite involved, so let's examine it in more detail. The actual macro file can be split into separate commands like this:

    ```
    2(dta/ /)
    pa/:/
    pb/:/
    3#
    ```

    Recall how the date is actually written:

    ```
    Monday 19-Apr-93 10:57:03
    ```

    The first part of the macro deletes everything up to the time by searching and deleting everything up to the second space inclusive. This leaves us with:

    ```
    10:57:03
    ```

    Next the edit "pointer" is placed after the colon in hours and before the colon separating the minutes and seconds. If this seems odd, it's just the way EDIT works. Finally the second colon and the last two digits are deleted leaving us with:

    ```
    10:57
    ```

17. Edits the variable TimeNow{$$} directly using the macro just described.

18. Compares the requested event time to the actual time. This test returns TRUE if the event time is less than the actual time; in other words the time has already passed.

19. Tests for the QUIET switch. This is an extra not supported by AddPestEvent and determines if a warning for a missed

(timed) event should be shown. If the QUIET switch is active control skips to Step 30.

20. If the test at Step 18 is true, this presents at requester indicating what has happened and provides some options as what to do. The user's response is sent to the global "RQ" acted later in the script. Results are as follows.

```
Yes=1; Show=2; Cancel=0.
```

21-23. If the user presses Cancel after a "Missed event" warning, this code causes the script to exit immediately.

24-29. If the user replies "Show" to the request at Step 20, this code displays the message and exits the script.

30. Closes the IF...ELSE...ENDIF construct opened at Step 19.

31. Closes the IF...ELSE...ENDIF construct opened at Step 18.

Event Creation (32-36)

32. Creates or increments the global variable, PestEvent. This variable is used internally to track the process numbers attached to any given event.

33. Loads a resident version of WAIT with the name determined by WAIT + the current Pest event number. Note: this is an internal name and has no bearing on the actual process number running the event. Each WAIT is ADDed to the resident list to allow multiple WAIT processes and to ensure the correct one is unloaded when the event completes.

34. Displays a confirmation that the event has been set. The event number shown is Pest's internal event number, and is not related to the process: which at this stage remains to be launched. Strictly, speaking this confirmation should not appear until after the task has been started but a programming consideration prevents this.

35. Creates a variable (PV + the event number) with details of the event. This information is used by ListPestEvents.

36. Creates a variable (PM + the event number) containing the entire event message.

Event Starting (37-38)

Most of the remaining script, up to Step 41, looks like a single command and forms the complete event process. This part sets up the new process, presents the completion message, removes the WAIT command and clears the event. It's important to note that these lines form an "asynchronous process" and do not have to complete in order for the script to finish. These lines launch the resident part of the event and

leaving it hanging around in memory. Because of this, SetPestEvent can be called many times in very rapid succession.

37. Creates the new process which will wait until the requested time is reached. Note that a specific resident copy of WAIT is called – the one numbered by this event. The "+" symbol at the end of this line ties it to step 38.

38. When the WAIT started at Step 37 "times out" the request appears. The time is inserted in the title string using command expansion and the message is inserted using manual expansion of GETENV. This allows the message to be changed at any time via indirection. See the discussion of RESCALC for a simpler example using this technique. The "+" ties this step to step 39.

Event Removal (39-41)

39. Kills (removes) the event and marks it complete. This information is used by ListPestEvents. The "+" ties this step to step 40.

40. Removes the WAIT command from the resident list and discards it from memory. The "+" ties this line to step 41.

41. This is the last line in the "RUN +" group and actually starts the process running. When control reaches here at some future time (determined at Step 37) this removes the Pest message.

42. Marks the bail-out point for the script.

43-44. Clear some variables that we don't want to leave hanging around.

Listing

```
1.  .key time/a,Message,day/k,QUIET/S
2.  .bra {
3.  .ket }
4.  if {day} not eq ""
5.   date >T:TPToday
6.   search >NIL: t:TPToday "({day})" pattern
7.   if warn
8.    skip out
9.   endif
10. endif
11. if "{time}" EQ "<Instant>"
```

```
12.  RequestChoice >env:RQ{$$} "Pest" "{Message}" "OK"
13.  skip out
14. endif
15. date >env:TimeNow{$$}
16. echo to T:EdTime "2(dta/ /);pa/:/;pb/:/;3#"
17. edit env:TimeNow{$$} with T:EdTime
18. if $TimeNow{$$} GT "{Time}"
19.  if "{QUIET}" EQ ""
20.   RequestChoice >env:RQ{$$} "Pest" "Requested event time:
    {time} has already passed.*nShould I wait until tomor-
    row?" "Yes" "Show" "Cancel"
21.   if $RQ{$$} EQ "0"
22.    skip out
23.   endif
24.   if $RQ{$$} EQ "2"
25.    RequestChoice >NIL: "Pest Override Message"
    "{Message}" "OK"
26.    skip out
27.   endif
28.  else
29.   skip out
30.  endif
31. endif
32. eval $PestEvent+1 to env:PestEvent
33. resident name=wait$PestEvent file=c:wait add
34. echo "Event $PestEvent set at {time}"
35. echo <NIL: >ENV:PV$PestEvent
    "$PestEvent.*e[I{time}*e[I*e[32m++Active++*e[31m*e[I"
    noline
36. echo <NIL: >ENV:PM$PestEvent "{Message}"
37. run <NIL: >NIL: wait$PestEvent until {time} +
38. RequestChoice >NIL: "Pest (active `date`)" "`getenv
    PM$PestEvent`" "OK" +
39. PEST3:KillPestEvent $PestEvent "+Complete+" sys=QUIT +
40. Resident Wait$PestEvent remove +
41. unsetenv PM$PestEvent
42. lab out
43. unsetenv RQ{$$}
44. unsetenv TimeNow{$$}
```

Pest 3: SetWaitEvent

Synopsis: [EXECUTE] SetWaitEvent [[Wait=]time] [[Message=]"event message"] [Day=Dayname]

Template: wait,Message,day/k

Path: SYS:Pest3

Requires: V3+

See also: AddPestevent, DeletePestEvent, KillPestEvent, StartPest, GetArgs, DeletePestEvent, ChangeMessage, SetPestEvent, ListPestEvents

Type: Pest support script

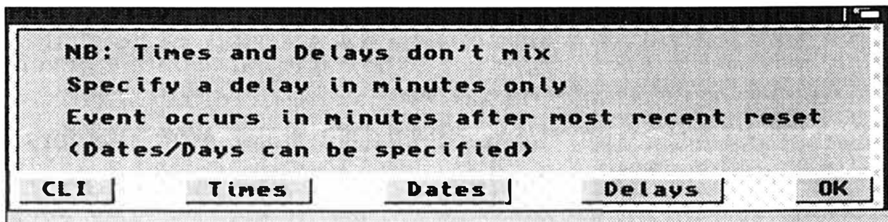
Brief: Sets the delayed events for Pest v3

Description

SetWaitEvent is the similar to SetPestEvent, except that the time is a delay in minutes. Example:

```
1>SetWaitEvent Time=5 Message="Five minutes have elapsed..."
Event 3 set in 5 minutes
```

Note that a QUIET switch has not been supplied with this command; you might like to add one for yourself – see SetPestEvent for more details.



Pest Delay Help

Line-By-Line

1-3. Defines a simple argument template and set up the brace characters.

4. Tests if some day names (see AddPestEvent) have been supplied. If not, control jumps to Step 10; otherwise it continues at Step 5.

5. Creates a file containing the current time and date as a string.

6. Checks the day name supplied matches "today": if not a WARN condition is returned. (See SetPestEvent for more information.)

7. Checks for a WARN condition from Step 6 and...

8. Leaves the script immediately. This speeds things up considerably.
9. Closes the IF...ENDIF construct opened at Step 7.
10. Closes the IF...ENDIF construct opened at Step 4.
11. Increases the current Pest "event" number and stores the result in PestEvent.
12. Adds a resident copy of WAIT to the resident list and names it as WAIT + the current event number. This is used to track the process attached to individual events.
13. Confirms the event and event number.
14. Creates the main event variable (used by ListPestEvents).
15. Creates the event message variable. This is stored separately from the event so that the message can be changed later using variable indirection.
16. Creates the event by RUN-launching WAIT to wait for a specific time. This line is attached by "+" to Step 17 and is part of the same process. (Note: WAIT events are controlled by the most recent reset.)
17. Sets the time-out requester using REQUESTCHOICE. The message is retrieved when this command is actually activated using `GETENV PM$PestEvent`. If a simple "$" was used, the message would be inserted when the script was interpreted and could not be changed. This line is attached to Step 18 by "+".
18. Removes the event from the active list using KillPestEvent. This line is attached to Step 20 by "+".
19. Removes WAIT from the resident list returning the memory it used to the system. This line is attached to Step 20 by "+".
20. Removes the event message from the system. The event proper is left hanging around so that ListPestEvents knows that it has completed. This line actually triggers the RUN process and physically starts the event.
21. Marks the bail-out point for non-starting events.
22. Removes the excess date file from the system.

Listing

```
1.  .key wait,Message,day/k
2.  .bra {
3.  .ket }
4.  if {day} not eq ""
5.   date >T:TPToday{$$}
```

```
6.   search >NIL: t:TPToday{$$} "({day})" pattern
7.   if warn
8.    skip out
9.   endif
10. endif
11. eval $PestEvent+1 to env:PestEvent
12. resident name=wait$PestEvent file=c:wait add
13. echo "Event $PestEvent set in {Wait} mins"
14. echo <NIL: >ENV:PV$PestEvent
    "$PestEvent.*e[I{wait}:M*e[I*e[32m++Active++*e[31m*e[I"
    noline
15. echo <NIL: >ENV:PM$PestEvent "{Message}"
16. run <NIL: >NIL: wait$PestEvent {wait} mins +
17. RequestChoice >NIL: "Pest (active `date`)" "`getenv
    PM$PestEvent`" "OK" +
18. PEST3:KillPestEvent $PestEvent "+Complete+" sys=QUIT +
19. Resident Wait$PestEvent remove +
20. unsetenv PM$PestEvent
21. lab out
22. delete >NIL: T:TPToday{$$}
```

Pest 3: StartPest

Synopsis: Workbench only

Template: none

Path: SYS:WBStartup

Requires: V3+ (as part of Pest 3)

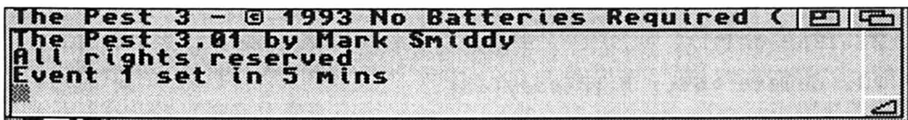
See also: AddPestevent, DeletePestEvent, KillPestEvent, GetArgs, DeletePestEvent, ChangeMessage, SetPestEvent, SetWaitEvent, ListPestEvents

Type: Startup script

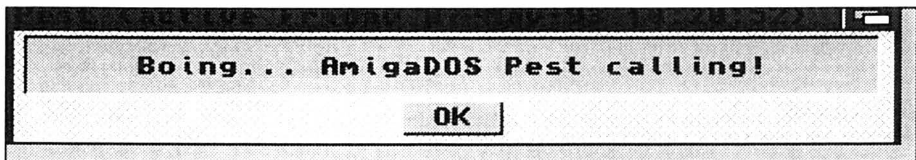
Brief: Starts Pest from Workbench's WBStartup drawer

Description

See "Pest 3: AddPestEvent" for a full description of The Pest. This is the only part of Pest to live in the WBStartup drawer and allows users to enable or disable the entire system by simply dragging the icon in and out of the drawer.



Pest Starting



Pest Calling

Line-By-Line

1-10. Make various Pest system commands resident. This makes Pest operate faster under most conditions and prevents having to swap back to the Workbench disk when an event times-out.

11. Copies the entire contents of the Pest3 system directory to a new directory in the Ram Disk. This allows Pest to execute its internal system commands without having to fiddle with paths, and worse, swapping disks.

12. Announces the Pest. This is version 3.01 some of the bugs present in the original have now been fixed and it is easier to use on a floppy-based system.

13. Creates the PEST3: assignment. This directory is used to reference the special Pest scripts.
14. Executes the Pest event program, PestFile. This always lives in the S: assignment and can be edited directly if you prefer.

Listing

```
1.  resident c:Break add
2.  resident c:Date add
3.  resident c:Delete add
4.  resident c:Edit add
5.  resident c:Eval add
6.  resident c:Execute add
7.  resident c:RequestChoice add
8.  resident c:Search add
9.  resident c:Status add
10. resident c:Type add
11. copy Sys:Pest3/-(#?.info) ram:Pest3 quiet
12. echo "The Pest 3.01 by Mark Smiddy*nAll rights reserved"
13. assign PEST3: RAM:Pest3
14. execute S:PestFile
```

PFIND

Synopsis: PFIND <file> <start directory>
Template: na
Path: na
Requires: 1.3+
See also: FFIND
Type: Alias
Brief: Find a file with automatic patterns
Definition: ALIAS PFIND SEARCH SEARCH=#?[]#? FILE ALL

Description:

This alias is almost identical to FFIND described earlier, but this version automatically includes a pattern. You might not always want to do this because the results can sometimes be unpredictable. Note that the directory always comes after the file you're looking for. This is the reverse of the natural settings for SEARCH and is necessary to get the required effect.

```
1>PFIND Pest SYS:
Workbench3:WBStartup/StartPest
Workbench3:WBStartup/StartPest.info
Workbench3:Pest3/DeletePestEvent.info
Workbench3:Pest3/ChangePestMessage.info
Workbench3:Pest3/AddPestEvent.info
Workbench3:Pest3/KillPestEvent.info
Workbench3:Pest3/ListPestEvents.info
Workbench3:Pest3/DeletePestEvent
Workbench3:Pest3/ChangePestMessage
Workbench3:Pest3/AddPestEvent
Workbench3:Pest3/KillPestEvent
Workbench3:Pest3/ListPestEvents
Workbench3:Pest3/SetPestEvent
```

QFF

Synopsis: QFF <DRIVE #> [NOICONS] [FFS] [QUICK]

Requires: V1.3+

See also: QF

Type: Alias

Brief: Quick format any floppy disk device

Definition: ALIAS QF FORMAT DRIVE DF[]: NAME Empty

Description:

This is a variation on the QF theme for those who despise the long winded format of the FORMAT command. Called QF – Quick Format – it takes a single parameter (the drive number) and formats a disk called Empty. The trashcan can be suppressed by adding NOICONS.

Examples:

```
1>QFF 0
1>QFF 0 NOICONS
```

QF

Synopsis: QF <DRIVE> [NOICONS] [FFS] [QUICK]

Requires: V1.3+

See also: QFF

Type: Alias

Brief: Quick format any floppy disk device

Definition: ALIAS QF FORMAT DRIVE [] NAME Empty

Description:

Quick Format takes a single parameter (the drive number name) and formats a disk called Empty. The trashcan can be suppressed by adding NOICONS. Be careful when using this with hard disks!

Example:

```
1>QF DF0:
1>QF DH1: NOICONS
```

RCD2

Synopsis: [EXECUTE] RCD2 [[number=] # | dir | pat]
[SAVE | LOAD]

Template: number,SAVE/S,LOAD/S

Path: S:

Requires: V2+

See also: FCD, RCD

Type: Script

Brief: Store recent directory changes in RAM as a menu

Description

RCD2 is the same as RCD with one important change: the directory list is global. It is compatible with RCD and FCD but will create a gobal directory list that can be accessed from within any Shell.

RCD Working

Line-By-Line

The listing here is the same as RCD with the following changes:

Lines: 6, 11, 15, 16, 23, 25, 32, 33, 35, 43; the directory list T:CDS{$$} has been replaced with a global list T:CDS. This ensures the list is available to all Shells.

Line: 18 the recursive call is changed to call RCD2. You MUST change this if you change the name of the script.

```
1.  .key number,SAVE/S,LOAD/S
2.  .bra {
3.  .ket }
5.  if "{LOAD}" EQ "LOAD"
6.   copy >nil: s:cds t:cds
7.  endif
8.  if "{SAVE}" EQ "SAVE"
9.   ask "This will replace your current file*nAre you sure
    N/y?"
```

```
10.  if warn
11.   copy >nil: t:cds s:cds
12.  endif
13. endif
14. if "{number}" EQ ""
15.  if exists t:cds
16.   type t:cds number
17.   echo "Enter directory, pick a number, any " noline
18.   execute s:rcd2 ?
19.   skip out
20.  else
21.   echo "Loading/making default file"
22.    if exists s:cds
23.     copy >nil: s:cds t:cds
24.    else
25.     echo >t:cds "*"`cd`*""
26.     skip out
27.    endif
28.  endif
29. endif
30. if VAL "{number}" EQ 0
31.  cd "{number}"
32.  echo >t:cd "*"`cd`*""
33.  join t:cd{$$} t:cds AS t:cd0{$$}
34.  echo >t:ed{$$} "9n;d"
35.  edit t:cd0{$$} with t:ed{$$} to t:cds ver=nil:
36. else
37.  eval >env:usr{$$} {number} -1
38.  if val $usr{$$} NOT GE 1
39.    echo >t:ed{$$} "n;9d"
40.  else
41.    echo >t:ed{$$} "$usr{$$} d"
42.  endif
43.  copy >nil: t:cds env:cd{$$}
44.  edit env:cd{$$} with t:ed{$$} ver=nil:
45.  cd $cd{$$}
46. endif
47. lab out
```

RCD

Synopsis:	[EXECUTE] RCD [[number=] # | dir | pat] [SAVE | LOAD]
Template:	number,SAVE/S,LOAD/S
Path:	S:
Requires:	V2+
See also:	FCD
Type:	Script
Brief:	Store recent directory changes in RAM as a menu

Description

This command is very useful if you have a hard disk. It stores a list of the last ten directory changes in RAM and allows you to pick one by selecting it from a numbered menu. You can choose to save your current list at any time, or load a pre-built one from disk. Every path feature available to CD, including patterns, may be used. The command line is sensitive to arguments so that the script can completely replace CD (using an ALIAS) if you prefer. Several modes are available:

• Called without arguments. The script shows the current list and prompts you to interactively select an existing entry, load or save the list or enter a new directory. Note: you can enter LOAD or SAVE at this prompt. Example:

```
1>RCD
        1. "Workbench3.0"
        2. "Workbench3.0:Fonts"
        3. "Apps:"
        4. "Workbench3.0:Fonts"
        5. "Workbench3.0:Devs/Keymaps"
Enter directory, pick a number, any number,SAVE/S,LOAD/S:
```

• Called with a new directory path: FCD selects the directory (if available) and adds its full path to the menu. (The oldest directory is removed.) Example:

```
1>RCD SYS:
```

• Called with a number from the directory menu. The directory is selected from the list and changed. Example:

```
1>RCD 4
```

• Asked to save the FCD/RCD preferences to disk.

```
1>RCD SAVE
```

This will replace your current file

Are you sure N/y?"

Line-By-Line

1-4. Comprise a standard header. Note that there are no required arguments for this script.

5. If the LOAD switch was specified, control continues at Step 6, otherwise it skips to Step 7.

6. Loads the default FCD/RCD file from disk to the private Shell RCD file, CDS#.

7. Terminates the IF...ENDIF construct opened at Step 5.

8. If the SAVE switch was specified, control continues at Step 9, otherwise it jumps to Step 13. Note: the arrangement of this sequence LOAD->SAVE ensures that even if *both* switches are supplied, the script does not overwrite a working file: it is simply loaded then saved again.

9. Displays a warning that you are about to write a CDS file to disk. A WARN condition is set if "Y" or "Yes" is entered at the prompt and cleared otherwise.

10. If Y was entered at Step 9, control continues at Step 11; otherwise it jumps to Step 12.

11. Saves the current CDS preferences file to disk.

12. Terminates the IF...ENDIF construct opened at Step 10.

13. Terminates the IF...ENDIF construct opened at Step 8.

14. If a number was entered at the command line (or from interactive mode) control is transferred to Step 29: otherwise it continues at Step 15.

15. Checks if a CDS# file exists. If found, control continues at Step 16; otherwise it's transferred to Step 20.

16. Displays the menu of currently saved directories with menu numbers. (The numbers are supplied by TYPE.)

17. Displays the first part of the prompt...

18. ...and calls RCD recursively to put it into interactive mode.

19. When the recursive call returns, this jumps to the bail-out point.

20. If control reaches this point from Step 19, it jumps to Step 28; otherwise it continues at 21

21. Control reaches here if a private history (CDS#) file could not be found. This displays a progress message to confirm this.

22. Checks if a RCD/FCD history file is already present on disk...
23. ...and loads it if it was found.
24. If control reaches here from Step 23, it jumps to Step 27; otherwise it continues at Step 25.
25. Creates a default private history file with the first entry being the current directory. Note that the directory is enclosed in quotes to avoid confusing CD with spaces.
26. Jumps to the bail-out point. You'll have to run the command again in this case.
27. Terminates the IF...ELSE...ENDIF construct from Step 22.
28. Terminates the IF...ELSE...ENDIF construct from Step 15.
29. Terminates the IF...ELSE...ENDIF construct from Step 14.
30. Checks if the value of the entry made for number was 0. This is the case if a text entry – a directory path – was made. If text was entered, control continues at Step 31; otherwise it jumps to Step 36.
31. Attempts to set the new directory. If this command fails because the directory cannot be found (or more than one directory matches, for patterns) the script stops. Normally, the directory is made current.
32. Creates a temporary file with the new current directory name enclosed in quotes.
33. Joins the new current directory to the existing list and saves the resulting file as t:CD0#.
34. Creates a simple edit macro thus:

    9n Move down nine lines (to line 10).

    d Delete the current line.
35. Uses the macro created at Step 34 to hack off the last entry in the file. Note if there are less than ten entries (directory paths) in the file, this macro has no effect. This macro therefore, only trims off the oldest entries. Changing the line count at Step 34 affects how many lines are stored in history. More than about 25 is getting silly and less than 3 is pointless. (If you increase this number, you will have to make changes later in the script too.)
36. If control reaches here from Step 30, it branches to Step 46; otherwise it continues at Step 37.
37. Subtracts 1 from the menu entry and stores the result in the global, Usr#.
38. Tests if the value of Usr# is less than 1 and if it is, control continues at Step 39; otherwise control jumps to Step 40.

39. Writes a simple macro to skip the first line of a file (n) and delete the next 9 lines (9d).
40. If control gets here from Step 38 if jumps to Step 42; otherwise it continues at Step 41.
41. Writes a simple macro to delete the first "Usr#" lines of a file.
42. Closes the IF...ELSE...ENDIF construct opened at Step 38.
43. Creates the new directory variable from the saved directory list.
44. Edits the history file with the macro created at Step 39 or 41 and creates a global, CD# using that information. Note that the contents of this variable can be 2 or more lines, but only the first line will be read by $CD#.
45. Changes to the selected directory.
46. Terminates the IF...ELSE...ENDIF construct opened at Step 30.
47. Marks the bail-out point for the recursion.

```
1.  .key number,SAVE/S,LOAD/S
2.  .bra {
3.  .ket }
4.  ;
5.  if "{LOAD}" EQ "LOAD"
6.  copy >nil: s:cds t:cds{$$}
7.  endif
8.  if "{SAVE}" EQ "SAVE"
9.  ask "This will replace your current file*nAre you sure
    N/y?"
10. if warn
11. copy >nil: t:cds{$$} s:cds
12. endif
13. endif
14. if "{number}" EQ ""
15. if exists t:cds{$$}
16. type t:cds{$$} number
17. echo "Enter directory, pick a number, any " noline
18. execute s:rcd ?
19. skip out
20. else
21. echo "Loading/making default file"
```

```
22. if exists s:cds
23. copy >nil: s:cds t:cds{$$}
24. else
25. echo >t:cds{$$} "*"`cd`*""
26. skip out
27. endif
28. endif
29. endif
30. if VAL "{number}" EQ 0
31. cd "{number}"
32. echo >t:cd{$$} "*"`cd`*""
33. join t:cd{$$} t:cds{$$} AS t:cd0{$$}
34. echo >t:ed{$$} "9n;d"
35. edit t:cd0{$$} with t:ed{$$} to t:cds{$$} ver=nil:
36. else
37. eval >env:usr{$$} {number} -1
38. if val $usr{$$} NOT GE 1
39. echo >t:ed{$$} "n;9d"
40. else
41. echo >t:ed{$$} "$usr{$$} d"
42. endif
43. copy >nil: t:cds{$$} env:cd{$$}
44. edit env:cd{$$} with t:ed{$$} ver=nil:
45. cd $cd{$$}
46. endif
47. lab out
```

RecDemo

Synopsis: [EXECUTE] RECDEMO [[dir=]dir|pattern] [d=private]

Template: dir,d

Path: S:

Requires: V1.3+

See also: Tree

Type: Script

Brief: To demonstrate recursion

Description

You may be finding the concept of recursion is a beggar to grasp. Don't worry, this is an area many "real" programmers avoid at any cost! As a more practical example here's a script modified to tell you where it is in the directory hierarchy, how deep the recursion is and when it starts to wind and unwind. Important: This script must not be multi-tasked!

Line-By-Line

1. The argument template shown here has two variables, but only one of them is for direct use. The other is passed back to the program during recursion. An environmental variable could be used here on later versions. See if you can figure out how.

2-3. The compulsory re-setting of the brackets to braces.

4. Sets the current directory private variable "d" to the current directory.

5-6. You've already met this kind of calculation. It uses EVAL's interactive mode to calculate the depth of the recursion. For the purposes of this demonstration only, the variable "depth" must be set before the script starts. It could be tested with IF EXISTS... but this is wasteful – at most this test would fail once on the first run. A better solution would be to call the recursive script from another (main) script. In AmigaDOS 2 and higher, this calculation can be written:

```
eval $depth + 1 to env:depth
```

7-8. Gives a progress message. The recursion depth will be printed on the same line, so the NOLINE switch is used on ECHO and ther variable TYPEd at Step 8. In AmigaDOS 2+, this couplet may be written thus:

```
echo "*nEntering: {d} ({dir}) - depth now: $depth"
```

9. This creates a list of directories in the current directory. We mention it because this causes the script to act slightly differently on the second and successive runs. This returns two arguments to the script – the parent path (%s%s) and the sibling directory name (%s). In this way the sibling name can be displayed without the extra complications of the path.

10. Heavens to betsy! This is where the script calls a script to call itself. If the LISTed script (called L here) is empty – that is there are no more sub-directories to search – execution continues at 5 and the script unwinds.

11-12. Again nothing new here. This just calculates the new nesting depth as the script unwinds. In AmigaDOS 2 this pair can be replaced by:

```
eval $depth - 1 to env:Depth
```

13-14. ...and this prints the sub-directory the script is leaving and the new depth. You'll notice that this number will tend to alternate between values when the script is searching sub-directories of directories like FONTS: this is quite normal. By the time you've run this script a few times you should be starting to get to grips with the idea. This couplet can also be re-written for AmigaDOS 2 thus:

```
echo " Leaving: {d} - depth now: $Depth"
```

Listing

```
1.  .key dir,d
2.  .bra {
3.  .ket }
4.  .def d {dir}
5.  eval <env:depth >nil: op=+ value2=1 to env:tmp ?
6.  copy env:tmp env:depth
7.  echo "*nEntering: {d} ({dir}) - depth now: " noline
8.  type env:depth
9.  list >T:L "{dir}" dirs lformat "execute recdemo
    *"%s%s*"*"%s*""
10. execute T:L
11. eval <env:depth >nil: op=- value2=1 to env:tmp ?
12. copy env:tmp env:depth
13. echo " Leaving: {d} - depth now: " noline
14. type env:Depth
```

RemAlias

Synopsis: [EXECUTE] <[name=]Alias name> <[string=]"command"> [comment="remarks"]

Template: name/a,string/a,comment/k

Path: S:

Requires: V2

See also:

Type: Script

Brief: Add remarks to or fix aliases against extra parameters

Description

This is an unusual use of I/O re-direction to force the comment character (;) as part of a command line – ; is read as a comment and normally ignored by the command parser. Why? ALIAS allows the user to type the alias and add additional parameters. The first is substituted at the '[ ]', the rest are tacked onto the end. However, there are times when it is useful to protect an alias against these additional parameters. Also, you might want to comment some of your aliases for future reference. In a multi-user setup you might want to prevent someone typing NEWCLI. This ALIAS seems to work:

```
1>ALIAS NEWCLI ECHO "Access denied"
```

In practice, although this works, if the user adds any additional parameters such as a window description ECHO fails. Here's what happens:

```
1>NEWCLI AUX:
Argument line invalid or too long
```

The error changes with different versions of AmigaDOS but this is confusing. The alias is interpreted thus:

```
ECHO "Access denied"AUX:
```

What's needed is a comment to stop the extra parameters having any effect. In other words, we want to make AmigaDOS interpret the alias like this:

```
ECHO "Access denied" ; AUX:
```

Because this works, it is not possible to add the semi-colon (;) to the alias definition directly like this:

```
ALIAS NEWCLI ECHO "Access denied";
```

because everything beyond the quote is truncated!

This script solves the problem. It forces ALIAS to pick up the comment and anything after it. By doing this you can add comments to aliases in a way no one ever thought possible. Provided the aliases only require one argument each, you can add comments to them which will appear in the alias list. For instance, how about this:

```
1>ALIAS
NEWCLI ECHO Access denied ; Prevent use of NEWCLI
```

Sharp eyed readers will be thinking: "That's not possible, he hasn't got quotes around the echoed string!" Quite right, there's a nice little trick involved here which makes this possible.

ECHO understands white space to mean either a tab or a space character. It doesn't know about ALT+Spacebar which also generates white space (strictly speaking it's a non-break space)! The example above was generated using this command line – the caret symbol (^) shows where to type ALT+Spacebar:

```
1>RemAlias NEWCLI "ECHO Access^denied" COMMENT "Prevent
 use of NEWCLI"
```

Line-By-Line

1: The argument key consists of two required arguments and a keyword. The name is going to become the alias name and the string will be the alias itself. This mirrors the normal ALIAS command with one difference – the string is a required argument. Also, as noted above, it should be surrounded by quotes. There is a way around this which we'll show later – it's a bit long winded to include here. The comment keyword allows you to optionally supply a comment to add to the end of the alias. This will appear in the alias list.

2-3: Re-define the bracket characters to { and }.

4: This is where the clever bit is done. ALIAS won't pick up a semi-colon on a command line but it will include it from interactive mode. This line uses ECHO to create a file which contains just such a simulated command line...

5: ...and this line picks it up using ALIAS's interactive mode. Once again re-direction to NIL: is used to make sure interactive output (the help template) is sent to oblivion.

Note: ALIAS in 1.3 doesn't have interactive input, so this technique can't be used. In fact, there's no need to either! You can just type the alias with a comment tacked on the end. AmigaDOS reads everything to the end of the line – directly equivalent to the /F argument in release 2.

Listing

```
1    .key name/a,string/a,comment/k
2    .bra {
3    .ket }
4    echo >env:Alias{$$} "{string}; {comment}"
5    alias <env:Alias{$$} >nil: {name} ?
```

Remt-Chat

Synopsis: [EXECUTE] Remt-Chat

Template: none

Path: S:

Requires: V1.3

See also: ...

Type: Script

Brief: Read piped message from host terminal

Description

This script is a companion to HOST-CHAT and is fully described there.

Listing

```
1.  lab start
2.  type pipe:B
3.  skip start back
```

RemoteRead

Synopsis: [EXECUTE] RemoteRead [time]

Template: time

Path: S:

Requires: V1.3+

See also: HostRead, Mail-2-Host, Mail-2-Remote

Type: Script

Brief: Read messages for the host machine

Description

This command is a part of matched pair of scripts. See HostRead for a full description.

Listing

```
1.  .key time
2.  .bra {
3.  .ket }
4.  .def time 30
5.  Lab Start
6.  list >T:ItsForMe{$$} T:#?.hst lformat "TYPE %s%s*nDELETE
    %s%s*n"
7.  if exists env:StopItNow
8.   quit
9.  endif
10. run execute T:ItsForMe{$$}
11. wait {time} mins
12. skip Start BACK
```

REN

Synopsis: REN <name or pattern> [AS|TO] <destination>

Template: na

Path: na

Requires: V1.3+

See also: DEL

Type: Alias

Brief: Short name for RENAME

Definition: ALIAS REN RENAME

Description:

This alias is not included for padding (as it might seem to be) it has a very serious us. REN is the MS-DOS command for RENAME and such users will feel much more at home if the command works like this. It's also shorter to type.

ResCalc

Synopsis: [EXECUTE] ResCalc <[First=]colour 1> <[Second=]colour 2> <[Multiplier=]multiplier> [[Tolerance=]Tolerance]

Template: First/a,Second/a,Multiplier/a,Tolerance

Path: S:

Requires: V2+

See also: ...

Type: Script

Brief: Calculate the value of a resistor using a standard colour code

Description

If you're not an electronics hobbyist, this script is really for fun although the unusual use of ECHO and global variables are quite notable. On a cleverness factor, this script is one of the most ingenious in the book. See if you can figure out how it works before reading the Line-By-Line description.

For as long as I can remember, carbon resistors (a type of electronic component) have used a standard colour code to indicate value. Experienced users memorise this code very early on and can spot a value just by practise; some resort to resistor code calculators and that's what this script does. Given an unknown resistor, you enter the colours and this script calculates its value. Note: the script is not very intelligent and will not check against a list of preferred values. It will generate an error if a colour code is not recognised though.

The tolerance band is optional and you don't have to enter the complete colour names. ResCalc uses the conventions to represent the ohms symbol as "R" and replaces the decimal point with the value indicator. For example:

```
1>rescalc yellow violet yellow
Resistor is: 4K7 (±????%)
1>rescalc brow red brow
Resistor is: 120R (±????%)
1>ResCalc red red yel gol
Resistor is: 220K (±   5%)
```

This script is unlike most of the others in the book in that it requires an extra data file. This must be entered with line numbers

exactly as shown or the program will not work. Electronics fans might like to expand this program to work with Polyester "candy-striped" capacitors which use a similar code.

The standard resistor colour code.

Colour	*Number*	*Multiplier*	*Tolerance*
Black	0	x 1	–
Brown	1	x 10	1%
Red	2	x 100	2%
Orange	3	x 1000	–
Yellow	4	x 10,000	–
Green	5	x 100,000	.5%
Blue	6	x 1,000,000	.25%
Violet	7	–	.1%
Grey	8	–	–
White	9	–	–
Gold	–	–	5%
Silver	–	–	10%
None	–	–	20%

Line-By-Line

1. Defines the command template. Note that the main three arguments are required: if one was missing the result would be undefined.

2-3. Redefine < and > to { and }.

4. Set the default value of Tolerance to x.

5-6. Makes SEARCH and EVAL resident. Note use of the ADD switch to allow more than one copy of these commands. This must be used to ensure the script can remove them safely later without affecting anything.

7. Sets the value of the global variable "OK" to 0.

8. Searches the "colourcodes" file for the a string matching the value entered for FIRST and places the entire line in the global, "F". If FIRST=RE, then F would read:

```
2   $A**K$B   Red        2%
```

9. Takes the result code from SEARCH, adds it to the global OK and stores it back in OK. This value will be used to check for any errors later on. If the colour code is not found, SEARCH returns 5, so a non-zero result in OK at the end of the script

will flag an error. (The result is not checked at this stage for speed and simplicity.)

10-11: As 8 and 9 for the second colour code. The result is stored in the global, "S". If SECOND=Vio, then F would read:

```
7  $A$B**OOMViolet  0.1%
```

12-13: As 8 and 9 for the multiplier. The result is stored in the global, "M". If the multiplier was "Red" then M reads:

```
2  $A**K$B  Red       2%
```

14. As 8 for the tolerance. The result (if there was one) is stored in the global, "T". No error checking is applied to this optional value.

15. The variable "F" is expanded, the first character extracted and stored in the global, "A".

16. The variable "S" is expanded, the first character extracted and stored in the global, "B".

17. The variable "M" is expanded, a special string extracted and stored in the global, "C". This string is the key to how ResCalc works. You'll see it in operation shortly. At this stage, the extracted string is (assuming Red): "$A**K$B".

18. The variable "T" is expanded and the tolerance string stored in the global, "D".

19-21. Check if the first number was 0 and (Black) and if this is the case, blanks the value to prevent a leading zero being included in the output. So, for example, Black-Brown-Black reads "1R" rather than "01R" which looks messy.

22-24. Provides simple error checking by testing the cumulative value in "OK" is still zero. If not, one or more of the colours entered was not found in the data table and the code (or data) must be wrong.

25. This is the really smart bit. The variable "$D" is expanded to the tolerance value directly, but the variable "$C" is expanded by an ECHO statement embedded in the same printed string. This is a process called "indirection" where the contents of the variable "$C" is determined by the contents of the variables contained within it. Confused?

Well imagine that "$C" contains "$A**K$B". When ECHO is called it expands the variables "$A" and "$B": the positions of which are determined by the multiplier's colour (Steps 12-13). If "A" and "B" contained "2" and "7" respectively, ECHO expands this to the string: "2K7". (The asterisks are there to separate the variables and prevent the parser from getting confused.) This string is inserted in the output of the main

ECHO statement and presto, the value appears like magic. Clever, isn't it!

26-28. Clean up the script and exit. SEARCH and EVAL are removed from the resident list to save memory.

ResCalc

```
1.  .key First/a,Second/a,Multiplier/a,Tolerance
2.  .bra {
3.  .ket }
4.  .def tolerance "x"
5.  resident c:search add
6.  resident c:eval add
7.  setenv OK 0
8.  search >ENV:F s:colourcodes {first} nonum
9.  eval $RC+$OK TO ENV:OK
10. search >ENV:S s:colourcodes {second} nonum
11. eval $RC+$OK TO ENV:OK
12. search >ENV:M s:colourcodes {multiplier} nonum
13. eval $RC+$OK TO ENV:OK
14. search >ENV:T s:colourcodes {tolerance} nonum
15. echo to env:A "$F" first=1 len=1
16. echo to env:B "$S" first=1 len=1
17. echo to env:C "$M" first=3 len=9
18. echo to env:D "$T" first=19 len=4
19. if val $A EQ 0
20.  setenv A ""
21. endif
22. if VAL $OK NOT EQ 0
23.  Echo "unknown code {First} {Second} {Multipler}"
24. else
25.  echo "Resistor is: `echo $c` (±$D%)" ; ± is ALT+Z
26. endif
27. resident search remove
28. resident eval remove
```

S:Colourcodes – numbers *must* be entered

```
0  $A$B**R   Black  ????%
1  $A$B**OR  Brown     1%
2  $A**K$B   Red       2%
3  $A$B**K   Orange ????%
4  $A$B**OK  Yellow ????%
5  $A**M$B   Green   0.5%
6  $A$B**OM  Blue   0.25%
7  $A$B**OOMViolet   0.1%
8  $A$B      Grey   ????%
9  $A$B      White  ????%
?  $A**R$B   Gold      5%
?  OR$A$B    Silver   10%
?  ??????    None     20%
```

SAFE

Synopsis:	SAFE [File|Pattern]
Template:	...
Path:	...
Requires:	V1.3+
See also:	UnSafe
Type:	Alias
Brief:	Protect files against deletion
Definition:	ALIAS Safe SPAT PROTECT [] -d

Description:

This simple alias makes files "safe" by clearing the deleteable flag. SPAT is used to allow patterns, examples:

```
1>Safe C:LIST
1>Safe DEVS:#?
```

SlideshowWB

Synopsis: SlideshowWB

Template: none (uses IconX

Path: Depends on Icon

Requires: V1.3+ (+ VILBM) or Workbench 3

See also: SlideShowWB

Type: Script

Brief: Show a list of pictures as a slideshow

Description

SlideshowWB uses VILBM (or Multiview if you have Workbench 3) to display pictures from the root directory of a disk as a slideshow. Like the Shell version, this script only works with dual drive systems and VILBM must be on the Workbench disk. More information on this script will be found under the description of SlideShow.

Line-By-Line

1: This is a dummy "key" variable which is not used by IconX - but it must be supplied for BRA and KET to work.

2-3: Change the brackets character from "<" and ">" to "{" and "}".

4: This prints a welcome message, waits for you to insert a disk and press Return.

5: This is the guts of the script – the meat in the sandwich. It uses the list command to create another script in T: (on the Ram Disk) called PIXn – where "n" is an arbitrary process number. It looks for files in the root directory of the disk in DF1: with the extension .PIC and creates a program something like this:

```
echo "Now showing Picture1.PIC"
VILBM "Pictures:Picture1.PIC"
echo "Now showing Picture2.PIC"
VILBM "Pictures:Picture2.PIC"
```

I'll be looking at how this sort of thing works later in the series, for now you might like to experiment with it and see for yourself. This is a script which actually writes new scripts. If you have Workbench 3, this line can be changed like this:

```
list >T:Pix{SS} df1:#?.PIC lformat "echo *"Now showing
    %s%s*"*nVILBM *"%s%s SCREEN *""
```

6: If you want to use this script from Workbench, you'll have to create a project icon for it and set the default tool to IconX. I'll leave that creativity to you.

Slideshow (IconX) version

```
1.  .key dummy
2.  .bra {
3.  .ket }
4.  ask "Slideshow*nPut the pictures disk in DF1: and press
    <Return>"
5.  list >T:Pix{$$} df1:#?.PIC lformat "echo *"Now showing
    %s%s*"*nVILBM *"%s%s*""
6.  execute T:Pix{$$}
```

Slideshow

Synopsis: [EXECUTE] Slideshow <[Drive=]drive number> [[ext=]extension]

Template: Drive/a, ext

Path: S:

Requires: V1.3+ (+ VILBM) or Workbench 3

See also: SlideShowWB

Type: Script

Brief: Show a list of pictures as a slideshow

Description

This uses VILBM to display pictures from the root directory of a disk as a slideshow. (VILBM, written by Sculpt 3D designer Eric Graham, is in the public domain and widely available.) Much the same idea could be used to play music tracks as a jukebox and so on. To keep Slideshow simple, this only works with dual drive systems and VILBM must be on the Workbench disk.

This version of Slideshow is essentially exactly the same as the Workbench version but with improvements to take account of command line options. Therefore, to use this program you would enter the drive number 0, 1, 2 or 3 and optionally supply the extensions used on the picture files - say PIC or IFF. You do not have to enter the full name, say DF0: or .IFF. the script adds those bits for you. If you have a hard disk, the drive description in line 6 should be changed to reflect this. (I use an assignment called SHOTS: for all my screen dumps - that keeps them all in one place.) Typical examples:

```
1>SlideShow 1

1>SlideShow 1 PIX
```

1: The command .KEY used here defines the command line as having one required argument (drive) and one optional argument (ext). These will ensure the drive number is always supplied and the extension is picked up when required.

2-3: Define the left and right-hand bracket characters as { and } respectively.

4: This defines the extension as .PIC in case one is not supplied. Extensions are not vital, but they are a good way of organising disks. In any case, this allows the script to separate picture files from, say dot-info files.

5: Like the Workbench version of the program, this defines the

startup message which prompts you to insert a disk. Note how the drive argument is included as part of the printed statement.

6: This line creates the program in the same way as the previous example, only this time some of the arguments sent to list are determined by the user options set at the command line. Users with Workbench 3 can change this line to read:

```
list >T:Pix{$$} df{drive}:#?{ext} lformat "echo *"Now
showing %s%s*"*nMutliview *"%s%s*" SCREEN"
```

7: And finally, run the program.

Listing

```
1.  .key drive/a,ext
2.  .bra {
3.  .ket }
4.  .def ext .PIC
5.  ask "Slideshow by Mark Smiddy*nPut the pictures disk in
    DF{drive}: and press <Return>"
6.  list >T:Pix{$$} df{drive}:#?{ext} lformat "echo *"Now
    showing %s%s*"*nVILBM *"%s%s*""
7.  execute T:Pix{$$}
```

STOP

Synopsis: [EXECUTE] STOP <[command=]process name>

Template: command/a

Path: S:

Requires: V1.3+

See also: Halt

Type: Script

Brief: To stop an asynchronous (RUN launched) process

Description

This script is relatively simple – it serves to show just what can be done in a few short lines. The idea is to stop a command once it has been started. You can do this either from the current CLI (if you used RUN) or from another CLI if you started the process directly.

Line-By-Line

1-3. This gets the command name from the user and re-sets the brackets to braces. The use of COMMAND/A makes certain that the script gets its argument.

4. The meat starts here. STATUS locates the process running as "command" and sends the process number to a file. An environmental variable is used to keep things neat and tidy. The actual file name is "stopper" with the calling CLI's number tacked on the end. For instance, if you started this from CLI #1, the filename would be "stopper1". The "{$$}" script variable does this. Note however, this only works if a .KEY is specified.

5. Uses BREAK's "interactive" BREAK to get the process number which will receive the stop code. The process number is retrieved from the environmental variable "stopper". The command would normally send its "help template" to the screen – >NIL: prevents this. The "?" at the end of the line is the crucial part. This is what puts the command into interactive mode, allowing it to read its input from the "stopper" file. We've used interactive mode extensively in these examples, so you should make sure that you understand it.

Listing

```
1    .key command/a
2    .bra {
3    .ket }
4    status >env:stopper{$$} com={command}
5    break <env:stopper{$$} >nil: all ?
```

SubDemo

Synopsis: [EXECUTE] SubDemo

Template: none

Path: S:

Requires: V1.3+

See also: ...

Type: Script

Brief: To demonstrate the implementation of simple subroutines

Description

As batch languages go, AmigaDOS has one of the best. However, it lacks the ability to jump to subroutines. Subroutines are used extensively by programmers to perform simple tasks because, in theory, a good program should be just a set of small subroutines.

Each subroutine can be called from anywhere in the program any number of times. A subroutine saves repeating the same section of code many times in the same program. By passing one or more parameters, a subroutine can be adjusted slightly at run time – subroutines of this type are more usually called procedures. These can be implemented using support scripts – the AskEm example is a good demonstration of this.

To a limited degree, AmigaDOS is capable of jumping to subroutines. Although the technique is not widely known and very rarely used. They are not quite as easy to implement as in other languages – but possible if you really need them. The advantage of a subroutine is it's faster than executing a completely separate script. We should stress that this is a contrived example meant purely for the purposes of demonstration – there are other means of achieving the same effect – but the script's output is not on trial here. The script's nature has necessitated a slight deviation from the usual analysis, however.

Line-By-Line

1. These points set up a return "address" as a variable. The variable and address names are arbitrary – however, the address name should be the same as that used at point 3. If it isn't, the script will not return to the correct point. Unfortunately there is no (easy) way to force the label name.
2. The script jumps to the subroutine at these points. Due to the nature of the implementation, the subroutine(s) must come after the main body of the script.

3. This marks the "return address". When the subroutine finishes, it will come directly back to this point. As we've already said, the label name must be the same as the return address defined at point 1.
4. This marks the start of the subroutine. The nature of AmigaDOS means that you call subroutines by name rather than line numbers. This forces you to write structured code.
5. This is the clever bit. It uses SKIP's interactive mode to retrieve the return address from the variable and jump back to the main body of the script. This leaves the technique open to errors because the return address can be altered at almost any point!

Advanced programmers can use this to their advantage however; gaining full control over the script's unconditional branch instructions. If the label does not exist, SKIP fails with an Return Code of 10 which can be tested with IF.

Listing

```
     .key dummy
     .bra {
     .ket }
1    echo >T:return{$$} "1"
2    skip DatePrint
3    lab 1
     wait 10 secs
1    echo >T:return{$$} "2"
2    skip DatePrint
3    lab 2
     wait 15 secs
1    echo >T:return{$$} "3"
2    skip DatePrint
3    lab 3
     echo "I am here"
     quit
     ; This is the subroutine
4    lab DatePrint
     echo "The time is:" noline
     date
5    skip <T:return{$$} >nil: back ?
```

SX

Synopsis: [EXECUTE] SX <[File=]scriptname>

Template: file/a

Path: S:

Requires: V2+

See also: WX

Type: Script

Brief: Execute an IconX file from Shell

Description

The idea for this script came late one night while I was driving home from a friend's house. I make this distinction for two reasons.

1: In spite of what people might think, writers do have time for things other than computers.

2: I was quite fatigued. Which is, more to the point, why the following happened.

My addled brain formulated the idea that not everyone was going to get the disk accompanying this book. (At this stage I have no way of knowing if the disk will be extra or be bound in with the cover at extra cost to you). This got me thinking, since some of the featured scripts need icons to run, a lot of people are going to be left wondering what do. Aha!, I thought, all I have to do is extract the "WINDOW=" tooltype information from the respective icons and run the script using that information.

So I did, and here, some 45 minutes later is the result. Of course, it does exactly what I predicted. It runs IconX configured project icons from the Shell: but (in case you haven't got it yet) you still need the icon! This does not defeat the object of what turns out to be a useful script, however, but it also gave rise to WX to be found later, which actually does run an IconX script without an Icon!

Using SX is simple, you just give it the name of the IconX icon you want to run (without the dot-info part) and let it do the rest. For example, to run AlarmClock from Shell you would enter:

```
1>SX AlarmClock
```

Since a new shell is opened by this script it has to be closed when the script exits (or is stopped) so this script also adds a close box to the window description. The DELAY= tooltype is not supported but the script issues a warning if this is found.

Line-By-Line

1-3. Comprise a standard header. Note that the file (a script) is required.

4. Creates a simple edit macro which just the deletes the first 8 characters of any file it is applied to.

5. Searches the "dot-info" file attached to the requested script for a WINDOW= description. This string should be present (although IconX does not require it) if the file is going to be processed by SX. The result is sent to global variable, WIN#. Early versions of SEARCH cannot search binary files, so this script is limited to AmigaDOS 2+.

6. If the WINDOW string could not be located the WARN condition is set and tested here. If found control resumes at Step 7; otherwise it jumps to Step 9.

7. Displays an error message.

8. Leaves the script by the back door.

9. Terminates the IF...ENDIF construct opened at Step 6.

10. Searches for the DELAY tooltype in the attached script and clears the WARN flag if it's found.

11. If the DELAY tooltype is present, control continues at Step 12, otherwise it branches to Step 13.

12. This error is probably not serious: and will not affect the vast majority of IconX scripts, so it is reported as a warning and the script is allowed to proceed.

13. Terminates the IF...ENDIF construct opened at Step 12.

14. Uses an edit macro to massage the unwanted "gunk" from the window description found by SEARCH at Step 5.

15. Places quotes and the extra "/CLOSE" tool around the window description. Quotes are necessary to avoid confusing NEWSHELL if the window's title includes spaces.

16. Creates a simple two line script which, assuming a script called AlarmRun, will end up looking something like this:

    ```
    echo "*e[0;0H*e[J" noline
    execute AlarmRun
    ```

    When this script is called it clears the current console window and positions the cursor at the start of the first line. (Unlike the CLEAR alias supplied by Commodore.)

17. Opens a new Shell executing the script at created at Step 16 and using a window defined by Steps 5, 14 and15. It has to be done like this because the NEWSHELL execute (FROM=) does not work the same way as a normal execute.

```
1.  .key file/a
2.  .bra {
3.  .ket }
4.  echo >t:ed1{$$} "8#"
5.  search >env:win{$$} {file}.info "WINDOW" NONUM
6.  if warn
7.   echo "Error: No window description?"
8.   quit
9.  endif
10. search {file}.info "DELAY" NONUM QUIET
11. if not warn
12.  echo "Warning: delay option not supported!"
13. endif
14. edit env:win{$$} with t:ed1{$$}
15. setenv win{$$} "*"$win{$$}/CLOSE*""
16. echo >t:ax{$$} "echo *"**e[0;0H**e[J*" noline*nexecute
    {file}"
17. newshell from=t:ax{$$} window=$win{$$}
```

TD

Synopsis: TD <marker>

Template: na

Path: na

Requires: 1.3+

See also: LD

Type: Alias

Brief: Mark the current directory for LD

Definition: ALIAS TD ASSIGN DIR_[]: ""

Description:

This is one of those clever little tricks that you will use once and wonder forever how you ever managed without it! TD (This Directory) memorises the current directory path and assigns it a simple name – it could be a number as suggested here. Later, you use LD to get back to this directory. It's a bit like PCD (supplied by Commodore) but a lot more versatile. There is no restriction on the number of directory changes you can store with this command and you can freely move around volumes too!

When you call this alias it expands thus:

```
1>TD 1
1>ASSIGN DIR_1: ""
```

In other words, it creates a directory assignment pointing to the current directory. Since the assignments are stored in a list there is no limit to the number you can have! The "ListD" script can be used to list the current assignments made with this alias.

Examples:

```
1>CD SYS:  ; change directory to root
1>TD 0   ; mark root as directory 0
1>CD Code:LC/Examples/Headers/Include/Devices
1>TD 1   ; mark this as directory 1
1>LD 0   ; go back to SYS:
1>CD
Workbench3.0:
1>LD 1   ; go back to 1
1>CD
Code:LC/Examples/Headers/Include/Devices
1>CD
Workbench3.0:Fonts
1>LD 1
1>CD
Code:LC/Examples/Headers/Include/Devices
```

TreeStart

Synopsis: [EXECUTE] TreeStart <[pat=]directory]

Template: pat

Path: S:

Requires: V2+

See also: Tree

Type: Script

Brief: Start the directory tree drawing program. Tree

Description

This script is only used to initialise the variables used by Tree. Something of a sledgehammer approach really, but the Tree script is complex enough itself without complicating things further. This intialisation speeds the main script by performing the once only intialisation that would otherwise have to be skipped in the script.

Listing

```
1.  .key pat
2.  setenv Depth 1
3.  setenv LastDepth 1
4.  execute Tree "<pat>"
```

Tree

Synopsis:	None (called by TreeStart)
Template:	dir,d
Path:	S:
Requires:	V2+
See also:	TreeStart
Type:	Script
Brief:	To draw the directory hierarchy

Description

Take heart, AmigaDOS 2+ owner, this is just for you. It's conceivable that this script could be written in AmigaDOS 1.3 but the extra work involved does not bear thinking about – it's large enough as it stands, and even on 25Mhz 68030 machines it isn't outstandingly fast.

Much the same effect can be gained from using DIR with the ALL and DIRS switches. Nevertheless, the output from this program is far more attractive and it serves to illustrate the techniques behind this type of programming. The backward links drawn by this script do not reflect the real structure of the disk – serving instead as a visual aid.

There are two scripts – the first just sets up a couple of variables and calls the meaty bit: TREE. Just to show how powerful AmigaDOS 2 is, this script doesn't pull any punches and uses every new facility it can.

Line-By-Line

1-3. Define the standard header. Note that "d" is a private variable only used for the recursive part of the algorithm.

4. Sets the default for the internal variable, d.

5. This acts as a safety-net for the edit macros defined below. All the actions undertaken by EDIT in these macros must execute at least once – 0 can cause some queer effects. The value of "Depth" must be at least one or control jumps to Step 27.

6. This test determines if "Depth" and "LastDepth" are the same. These values are affected later in the script (or by previous runs). If Depth=LastDepth this means that the directory level has not changed. Or, put another way, the directory about to be displayed is at the same level in the hierarchy as the previous directory. If "Depth" <> "LastDepth", control jumps to

Step 12 and continues from there.

7-11. Control arrives here when "Depth" equals "LastDepth". This means that the nesting level has not changed. These lines construct an edit macro to output a vertical bar (|) above the directory name – both indented to the current directory level (depth):

```
|
System
```

7. `"b//|"`

   Inserts the "|" character at the start of the first line. (b// = before the first character)

8. `"$Depth b/|/ "`

   Inserts Depth * 3 spaces before the | symbol. Remember, Depth is expanded as the string is appended to the macro file. The number is used by EDIT to repeat the action of the "B" command.

9. `"n; $Depth b// "`

   Moves to the next line in the file (expanded to the directory name) and indents it by Depth * 3 spaces.

10. Creates a file consisting of a 1 blank line, the directory name highlighted by changing the print colour, and finally the current values of Depth and LastDepth in brackets.

11. Produces the output file ready to be printed by editing the string created at 10 with the macro created at 7-9 Note: errors are re-directed to NIL:.

12. Marks the jump point from 6. If control arrives here from 11 it continues at Step 30.

13. When Control arrives here if:

    a: Depth > LastDepth – control continues at 14.

    b: Depth = LastDepth – control branches to 20.

14. Control gets here if "Depth" did not equal "LastDepth". It determines if "Depth" is greater than "LastDepth". These values are affected later in the script (or by previous runs). If Depth > LastDepth the directory level has increased. Put another way, the directory about to be displayed is a child of the last one displayed. Since control has reached this point, we need an edit macro to indent the directory and draw a line from where it was to where it is now using the current depth. On screen this looks something like this:

```
l___Garnet
|
```

14. Just like Step 7, this inserts a string at the beginning of the first line in the file. Incidentally, lower case "L" was used because it gives the best effect on the Amiga's screen in Topaz 8. You may like to experiment with different (fixed width) fonts.

15. Indents "l___" by Lastdepth * 3 spaces. At this point LastDepth equals Depth-1 so this makes "l___" line up with the bottom of the this directory's parent.

16. Moves to the next line of the file. Then inserts Depth * 3 spaces before the "|" character.

17. Moves to the next line in the file (expanded to the directory name) and indents that by Depth * 3 spaces.

18. This creates a file consisting of two blank lines and the expanded directory name highlighted in a different colour. The Depth and LastDepth variables are printed also.

19. Produces a file ready to be printed by editing the string created at 18 with the macro created at 14-17. Note: Errors are redirected to NIL:.

20: When control arrives here if:

```
a: Depth <= LastDepth – control continues at 21
b: Depth > LastDepth – control branches to 30
```

21. If control arrives at this point, there is only one possibility left: the program has unwound one or more directory levels. This calculation works out how many levels were unwound (the difference between "Depth" and "LastDepth") and stores the result in the environmental variable "Back". This will be used later to determine the length of the joining line.

22-27. These lines create the edit macro for the most complex job of all. That is, the one where the directory hierarchy drops back to a previous level. The catch is, because of the way recursion works, this drop might be 2, 3, or more levels in one jump! The result looks something like this on screen:

```
______|
|
Utilities
```

22. As in the previous cases, this inserts the "|" character at the start of the first line.

23. Indents "|" by LastDepth * 3 spaces.

24. This is the crucial part of this macro. The first command just moves to the next line in the file. Then inserts Back * 3 overscore characters AFTER the "|". On UK keyboards this is the SHIFT + ALT + N key combination.

25. This indents the whole line just defined by Depth * 3 spaces.
26. Finally this moves to the next line in the file, then indents it too by Depth * 3 spaces.
27. Creates the temporary file to be edited using the macro just defined.
28. Produces a file ready to be printed by editing the string created at 27 with the macro created at 22-26. Note: errors are re-directed to NIL:
29. Marks the end of the IF construct opened at 13.
30. Marks the end of the IF construct opened at 6.
31. Marks the end of the IF construct opened at 5.
32. Sets the variable LastDepth to the current value of Depth. This will be used on the next run.
33. Calculates the new value of Depth. Depth increases by one every time a new directory is entered. ie, when the script calls itself recursively.
34. Displays the file created at 11, 19 or 28 – drawing the next part of the directory tree.

35-36. Perform the file-based recursion.

37. Reduces the value of Depth by one. This happens each time the script unwinds one level of recursion.

Listing

```
1.  .key dir,d
2.  .bra {
3.  .ket }
4.  .def d {dir}
5.  if $Depth GT 0
6.   if $Depth EQ $LastDepth
7.    echo >t:ed0 "b//|"
8.    echo >>t:ed0 "$Depth b/|/ "
9.    echo >>t:ed0 "n; $Depth b// "
10.   echo >t:ed2 "*n*e[32m{d}*e[31m($Depth,$LastDepth)"
11.   edit t:ed2 t:ed3 with t:ed0 ver=nil:
12.  else
13.   if $Depth GT $LastDepth
14.    echo >t:ed1 "b//l___"
15.    echo >>t:ed1 "$LastDepth b/l/ "
16.    echo >>t:ed1 "n; $Depth b/|/ "
```

```
17.     echo >>t:ed1 "n; $Depth b// "
18.     echo >t:ed2 "*n|*n*e[32m{d}*e[31m($Depth,$LastDepth)"
19.     edit t:ed2 t:ed3 with t:ed1 ver=nil:
20.    else
21.     eval $LastDepth - $Depth to env:back
22.     echo >t:ed2 "b//|"
23.     echo >>t:ed2 "$LastDepth b// "
24.     echo >>t:ed2 "n; $Back a/|"/___"
25.     echo >>t:ed2 "$Depth b// "
26.     echo >>t:ed2 "n; $Depth b// "
27.     echo >t:ed4 "*n|*n*e[32m{d}*e[31m($Depth,$LastDepth)"
28.     edit t:ed4 t:ed3 with t:ed2 ver=nil:
29.    endif
30.   endif
31.  endif
32.  setenv LastDepth $Depth
33.  eval $Depth +1 to env:Depth
34.  type t:ed
35.  list >T:L "{dir}" dirs lformat "execute Tree *"%s%s*"
     *"%s*""
36.  execute T:L
37.  eval $Depth -1 to env:depth
```

UNSAFE

Synopsis: UNSAFE [File|Pattern]

Template: ...

Path: ...

Requires: V1.3+

See also: Safe

Type: Alias

Brief: Enable deletion for a file or files

Definition: ALIAS UnSafe SPAT PROTECT [] +d

Description:

This simple script is a direct compliment to SAFE and sets the deleteable flag on the file (or files) specified. SPAT is used to allow patterns. Example:

```
1>UnSafe C:WAIT
1>UnSafe C:#
```

VLS

Synopsis: VLS

Template: na

Path: na

Requires: V2.0+

See also: VOLS, DLS, DVS

Type: Alias

Brief: Check an assignment without removing it

Definition: ALIAS VLS ASSIGN VOLS

Description:

This short alias performs a similar function to the VOLS script, although it displays both mounted and available volumes. Example:

```
1>VLS
Ram Disk   [Mounted]
Wordworth
Workbench3.0 [Mounted]
```

VOLS

Synopsis: [EXECUTE] VOLS

Template: none

Path: S:

Requires: V1.3

See also: DEVS

Type: Script

Brief: List all mounted volumes using ASSIGN

Line-By-Line

1. Gets the assignment list and sends it to a file in the T directory. Note the use of <$$> to allow multi-tasking.
2. Displays a simple message to show what's going on...
3. ...and searches the output from ASSIGN for the "]" character. This only appears on mounted volumes, so that's what is displayed.

Listing

```
1.  ASSIGN >T:temp<$$>
2.  ECHO "Mounted volumes:"
3.  SEARCH T:temp<$$> "]" nonum
```

WD

Synopsis: LD

Template: na

Path: na

Requires: V1.3+

See also: TD, LD, ListD

Type: Alias

Brief: Show a memorised directory

Definition: ALIAS WD ASSIGN DIR_[]: "" EXISTS

Description:

WD is the ALIAS compliment of ListD. It simply checks which directory a particular label is assigned to and displays it. The label is not cleared by this action.

```
1>CD Code:LC/Examples/Headers/Include/Devices
1>TD 1  ; mark this as directory 1
1>WD 1
Code:LC/Examples/Headers/Include/Devices
1>WD 5
Code:LC/Examples
```

Who

Synopsis: Who <Command name>

Template: na

Path: na

Requires: V1.3+

See also: ...

Type: Alias

Brief: Search the process list for a command

Definition: ALIAS WHO STATUS COM=[]

Description:

This might seem a strange alias, but making the best use of commands is just what ALIAS is all about. This little program makes it possible to discover the process number currently running a named command. If more than one process is running the command, the first is listed (all commands from 3.0). NOTE: The WARN flag is set if the process is not found.

Example:

```
1>WHO WAIT
 3
```

WX

Synopsis: [EXECUTE] WX <[file=]scriptname>

Template: file/a

Path: S:

Requires: V2+

See also: SX

Type: Script

Brief: Execute a script in its own window (like IconX)

Description

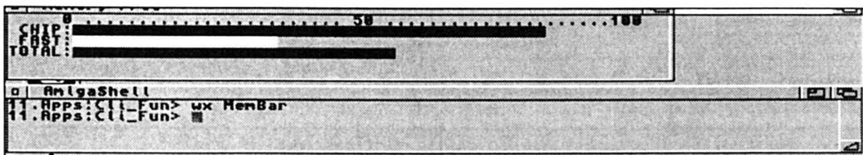
This script is what SX should have been! If I hadn't been so tired at the time, then it might just have been this! This script is a very handy way of creating scripts that can run in their own windows on a Workbench without all the fiddling around with Icons, IconX and NEWSHELL. It does it all for you – all you have to supply is a special window description somewhere in the file! Typically, you can call WX like this:

```
1>WX AlarmClock
1>WX Clock
Sorry, Clock is not configured for WX
```

In the second example, the script had not been prepared for WX. All you have to do is supply a console (RAW, CON: etc.) description as you would to IconX: but preceed it with a special string: ";WX:WINDOW=". The use of a comment character is quite deliberate: this allows you to incorporate the line anywhere in the script – even straight after the header. For example:

```
key dummy
.bra {
.ket }
;WX:WINDOW=con:0/0/500/40/Script window/CLOSE
echo >t:ed1{$$} "11#"
```

This line is completely ignored by AmigaDOS, but interpreted by WX; so its simple to incorporate and easy to use. If you include more than one window description only the first one is used.



WX Starting Membar

Line-By-Line

1-3. Comprise a standard header.

4. Creates a simple EDIT macro to remove the special WX window identifier from the window description. It just deletes the first 11 characters of any file its applied to.

5. Attempts to locate the WX window description marker in the specified file. If found, the result is sent to a global variable, WIN#; the WARN flag is set if it could not be found.

6. If the WARN flag is not set (the marker was found) control continues at Step 7; otherwise it jumps to Step 11.

7. Trims the excess grunge from the window description (held in "WIN#") using the edit maro defined at Step 4.

8. Places quotes around the window description and adds the CLOSE option to make the window easier to get rid of.

9. Creates a simple script, "ax#" which will be used to clear the new window and execute the required script proper.

10. Starts the new Shell process using a the supplied window parameters and special script.

11. If control reaches here from Step 10, it jumps to Step 13; otherwise it proceeds...

12. ...here and informs the user that the script does not have a proper window description or the marker is badly formed.

13. Closes the IF...ELSE...ENDIF construct opened at Step 6.

Listing

```
1.  .key file/a
2.  .bra {
3.  .ket }
4.  echo >t:ed1{$$} "11#"
5.  search >env:win{$$} {file} ";WX:WINDOW=" NONUM
6.  if not warn
7.   edit env:win{$$} with t:ed1{$$}
8.   setenv win{$$} "*"$win{$$}/CLOSE*""
9.   echo >t:ax{$$} "echo *"**e[0;0H**e[J*" noline*nexecute
    {file}"
10.  newshell from=t:ax{$$} window=$win{$$}
11. else
12.  echo "Sorry, {file} is not configured for WX"
13. endif
```

X

Synopsis: X <script> [options]
Template: ...
Path: ...
Requires: V1.3+
See also: ...
Type: Alias
Brief: Short name for EXECUTE
Definition: ALIAS X EXECUTE

Description:

This must be the simplest alias in the book (and probably the one I use the most). I use X all the time when I'm developing scripts because it saves typing EXECUTE. You could give this alias a longer name, but why bother!

XCD

Synopsis: [EXECUTE] XCD [[pat=]pattern]

Template: pat

Path: S:

Requires: v1.3 – 1.3.3

See also: SubDemo

Type: Script

Brief: To add pattern matching and history to CD

Description

The introduction of AmigaDOS 2 sees some remarkable changes and upgrades to commands like CD and LIST and the pattern matching algorithms. Both have been extensively revamped and enhanced. There's no longer any need to use CD to move between assignments or directories, because the Shell implies them. CD is only used when pattern matching of the directory specification is required. Implementing the same thing in AmigaDOS 1.3 is not easy – but something similar is possible.

Let's face it, if you want the features of AmigaDOS 2, you'll just have to upgrade. XCD works like Commodore's PCD but doesn't fall over when it encounters spaces in directory names. Unlike PCD, it adds single pattern matching. Multiple patterns are not possible in release 1.3 – a limitation of LIST.

Line-By-Line

1-3. Sets the key and, wait for it, resets bra and key to something else.

4. If a pattern has been supplied, control jumps to Step 15; otherwise it continues at Step 5.

5-6. Prints the current directory on a single line.

7. Saves the current directory setting to a file...

8. ...and this retrieves the new directory specification from another file.

9-11. These are support lines for the "subroutine" call. Each one defines the return address for the subroutine to branch back to when it exits. In most languages this part is automatic and invisible – in AmigaDOS the simulation of subroutines requires us to do some of the work.

12. Copies the history to the history variable. The source file in this case is created by the QUOTES subroutine.

13-14. Show the current directory.

15. Control usually gets here if a pattern was supplied, if not it jumps to Step 27.

16. Forms the list of directories on the specified path in the format. All pathnames are enclosed in quotes to stop CD complaining about invalid command lines.

17-26. It is known – and highly likely – that one or more directories will match the same pattern. If "#?" is used, all directories in that path will be listed. SEARCH is used here to produce two effects:

- Ensures at least one directory is found and generates a WARN if not. It does this by scanning for a quote character which is inserted during the list phase.
- Displays all the directories found during the list phase. This is default for this command but it lets the user know which directories were located. The first in this list is entered and this can help to debug the pattern.

20. Here CD uses another curious and very useful feature of interactive mode which is this: The command only receives input from the source file up to the first carriage return. This is usually taken for granted since interactive files are usually only one line long. However, it means you can drag a single argument from the top of a list of many possible arguments. It's a bit like getting a pack of cards and using the one from the top of the pile.

29-33. Are the QUOTES subroutine. In brief, this short routine takes a single line file and surrounds it with quotes. A slightly more complex approach would be required for multi-line files – which are not passed to this subroutine. This is important, never use 20 lines to do something you can do just as well in two. Use whatever you need to make it great – but keep it simple too. In this case a facility to quote an entire multi-line file is overkill. The main script body handles that part during the list phase.

29. This marks the start of the QUOTES "subroutine". It's a good idea to leave at least one blank line between the end of the script and the start of any subroutines.

30. Labels the subroutine for AmigaDOS's SKIP command.

31. Creates a two line EDIT auto which expands as follows:

```
cl/"/
```

Appends a quote to the end of the line by concatenating (joining together) the end of the current line, a quote and the next line. Note: the next line is empty because of the carriage

return present at the end of the original file. The truth is, I'm foxing EDIT to make it do something it shouldn't – it just works that way.

```
b//"
```

This appends a quote to the start of the line. The end result could look something like this:

```
"RAM DISK:clipboards"
```

Remember, these quotes must be present or CD will fail. This is because it will interpret the space between "RAM" and "DISK" as a delimiter – thinking that the new directory is RAM and the rest of the command line is a user error.

32. This creates the previous directory history variable "LastDir{$$}" using the edit macro just defined...
33. ...and this leaves the subroutine returning to the calling point.

Listing

```
1.  .key pat
2.  .bra {
3.  .ket }
4.  if "{pat}" EQ ""
5.   echo "Old directory: " noline
6.   cd
7.   cd >T:Last{$$}
8.   cd <T:lastDir{$$} >nil: ?
9.   setenv Return 1
10.  skip QUOTES
11.  lab 1
12.  copy T:LastDir{$$} T:Last{$$}
13.  echo "New directory: " noline
14.  cd
15. else
16.  list >T:cdt{$$} {pat} dirs lformat "*"%s%s*""
17.  search T:cdt{$$} "*'"
18.  if not warn
19.   cd >T:Last{$$}
20.   cd <T:cdt{$$} >nil: ?
21.   setenv Return 2
22.   skip QUOTES
```

```
23.   lab 2
24.  else
25.   echo "No files appear to match: {pat}"
26.  endif
27. endif
28. quit
29. ; The subroutine starts here...
30. lab QUOTES
31. echo >T:cdt{$$} "cl/*"/*nb//*""
32. edit T:Last{$$} T:LastDir{$$} with T:cdt{$$}
33. Skip <env:Return >nil: back ?
```

Mastering Amiga Guides

Bruce Smith Books are dedicated to producing quality Amiga publications which are both comprehensive and easy to read. Our Amiga titles are being written by some of the best known names in the marvellous world of Amiga computing. Below you will find details of all our currently available books for the Amiga owner.

Titles Currently Available

Brief details of the titles currently available along with review segments are given below. New publications and their contents are subject to change without notice. If you would like a free copy of our catalogue and to be placed on our mailing list then phone or write to the address below.

Our mailing list is used exclusively to inform readers of forthcoming Bruce Smith Books publications along with special introductory offers which normally take the form of a free software disk when ordering the publication direct from us.

Bruce Smith Books,
PO Box 382,
St. Albans, Herts, AL2 3JD
Telephone: (01923) 894355
Fax: (01923) 894366

Note that we offer a 24-hour telephone answer system so that you can place your order direct by 'phone at a time to suit yourself. When ordering by 'phone please:

- Speak clearly and slowly
- Leave your name and full address and contact phone number
- Give your credit card number and expiry date
- Spell out any unusual names

Note that we do not charge for P&P in the UK and endeavour to dispatch all books within 24-hours.

Buying at your Bookshop

All our books can be obtained via your local bookshops – this includes WH Smiths which will be keeping a stock of some of our titles, just enquire at their counter. If you wish to order via your local High Street bookshop you will need to supply the book name, author, publisher, price and ISBN number.

Overseas Orders

Please add £3 per book (Europe) or £6 per book (outside Europe) to cover postage and packing. Pay by sterling cheque or by Access, Visa or Mastercard. Post, Fax or Phone your order to us.

Dealer Enquiries

Our distributor is Computer Bookshops Ltd who keep a good stock of all our titles. Call their Customer Services Department for best terms on 021-706-1188.

Compatibility

We endeavour to ensure that all general *Mastering Amiga* books are fully compatible with all Amiga models and all releases of AmigaDOS and Workbench.

Mastering AmigaDOS 3 Volume One – Tutorial by Mark Smiddy

ISBN: 1-873308-20-5, Price £21.95, 384 pages.

The place to begin if you want to learn about and effectively use AmigaDOS. Covering both AmigaDOS 2 and 3, the tutorial guide assumes no previous knowledge of AmigaDOS. From formatting a disk to pipes and multitasking, even multi-user, this volume will turn the novice into an expert with its practical approach and many fascinating examples. The disk which accompanies the book contains all the examples and many other useful AmigaDOS tools.

Mastering AmigaDOS 3 Volume Two – Reference by Mark Smiddy

ISBN: 1-873308-18-3, Price £21.95, 416 pages.

Following on from the best selling Mastering AmigaDOS 2 volumes, Mastering Amiga DOS 3, Volume Two is a complete A to Z reference to DOS commands covering versions 2.04, 2.1 and 3. The action of each command is explained and examples to try are provided. Chapters on AmigaDOS error codes, viruses, the Interchange File Format (IFF), the Mountlist and the new hypertext system, AmigaGuide, complete this valuable guide.

Also:

Mastering AmigaDOS 2 Volume One (Tutorial)
by Mark Smiddy and Bruce Smith
ISBN: 1-873308-10-8, price £21.95, 416 pages.

Mastering AmigaDOS 2 Volume Two (Reference)
by Mark Smiddy and Bruce Smith
ISBN: 1-873308-09-4, price £19.95, 368 pages.

These two volumes follow the same great layout and comprehensive content as this *Mastering AmigaDOS 3* book, but cover the older versions of DOS, namely 1.3, 2.04 and 2.1.

AmigaDOS Insider Guide by Mark Smiddy

ISBN: 1-873308-37-X, Price £14.95, 256 pages.

If you've worn out your Workbench and are excited by the possibilities offered by AmigaDOS itself, then this Insider Guide is for you! It introduces all aspects of AmigaDOS from beginners level with plenty of examples in the now famous *Insider Guide* format. This is the guide to the practical use of AmigaDOS without any technical sidetracks or jargon. Take control of your Amiga.

Mastering Amiga Workbench 2

ISBN: 1-873308-08-6, Price £19.95, 320 pages.

The most comprehensive guide to Workbench 2 available. Every aspect of Workbench 2 is explained through tutorials and step by step guides. If you need a guide to Workbench 2 then this is the way to master it first time! No stone is left unturned.

Mastering Amiga Workbench 3

ISBN: 1-873308-31-0, Price £19.95, TBA pages.

The most comprehensive guide to Workbench 3 available, it follows the popular and easy to understand format of the best-selling Workbench 2 book. Every aspect of Workbench 3 – including 3.1 – is explained through tutorials and step by step guides. If you need a guide to Workbench 3 then this is the way to master it first time!

Workbench 3 A to Z Insider Guide by Bruce Smith

ISBN: 1-873308-28-0, Price £14.95, 256 pages.

Every aspect of the Amiga Workbench is documented with screen shots and examples of usage. Once you've become familiar with Workbench techniques, this alphabetical reference proves invaluable when you need to find a file, remember a menu operation or...how do you run that Commodity? Owners of A500 Plus and A2000/3000 upgrading to Workbench 3 will find this an essential add-on to their manuals.

Amiga A1200 Insider Guide by Bruce Smith

ISBN: 1-873308-15-9, price £14.95, 256 pages.

Assuming no prior knowledge, it shows you how to get the very best from your A1200 in a friendly manner and using its unique Insider Guide steps. Configuring your system for printer, keyboard, Workbench colours, use of Commodities and much much more has made this the best-selling book for the A1200.

As well as easy to read explanations of how to get to grips with the Amiga, the book features 55 of the unique Insider Guides, each of which displays graphically a set of step by step instructions. Each Insider Guide concentrates on a especially important or common task which the user has to carry out on the Amiga. By following an Insider Guide the user learns how to control the Amiga by example. Beginners to the A1200 will particularly appreciate this approach to a complex computer.

The disks which come with the A1200 contain a wealth of utilities and resources which allow you to configure the computer for your own way of working. The step by step tutorials take you through using these point by point, anticipating any problems as they go. There are also fully fledged programs such as MultiView and ED which can seem impenetrable for the new user but which become clear when observed in use over the shoulder of author Bruce Smith.

Great new features such as the colour wheel, Intellifonts, using MSDOS disks with CrossDos and configuring sound are dealt with in detail. A useful appendix acts as a file locater so that any of the many files on the Amiga disks can be quickly found.

Amiga A1200 Next Steps by Peter Fitzpatrick

ISBN: 1-873308-24-8, Price £14.95, 256 pages.

For those who have mastered the very basics of the A1200 this book is the ideal companion to our Amiga A1200 Insider Guide. Leaving the basics of the Workbench and AmigaDOS behind this book takes you the next step and shows you how to get the very most out of your A1200, using both the software supplied and other material readily available.

For example, learn how to use MultiView to write your own adventure game and edit a picture! Create your own fully recoverable Ram disk, get better results when you print out, recover deleted files. We even show you how to add your own hard disk and copy software onto it! This is only the tip of the iceberg. Amiga A1200 Next Steps is worth its weight in gold!

Amiga Assembler Insider Guide by Paul Overaa

ISBN: 1-873308-27-2, Price £14.95, 256 pages.

The Amiga Assembler Insider Guide has been written for the new user who wishes to learn to write programs in the native code of the Amiga computer – assembly language. The approach taken to this often daunting subject is designed to achieve practical results with short examples which demonstrate different programming skills. Each program in the book can be assembled and run in under one minute so the beginner need have no fear of long impenetrable listings. This is programming for the novice, made all the easier though the mini Insider Guides which summarise important operations and fundamental concepts.

Possible stumbling blocks and areas which regularly cause beginners problems are taken head on. No extra software is required to run the examples provided. After reading the book, the user will be able to confidently type in and edit source code, assemble it, debug it and and run it.

The book is compatible with all the main assemblers on the market. A support disk is available from the publisher which contains the A68K assembler, all the listings in the book, additional utilities and examples (cost £2.00 P&P). With the Amiga Assembler Insider Guide learning assembler on the Amiga has never been easier.

Amiga Disks and Drives by Paul Overaa

ISBN: 1-873308-34-5, Price £14.95, 256 pages. FREE Utilities disk.

Just what do you do when all your valuable data is locked in your computer? How do you copy files and install software? What do you do when you can't find a file on the Workbench screen? This book has all the answers!

Paul Overaa teaches you how to use and care for all types of disk drives in order to minimise the risk of problems, to get a better understanding of how they work and what you can do if things go wrong. Packed with practical topics, it's step by step guides are invaluable to novice and advanced users alike. Applicable to all Amigas.

Amiga A1200 Beginners Pack

ISBN: 1-873308-30-2, Price £39.95 plus £3 p&p, one-hour Workbench basics video and two books (A1200 Insider Guide and Amiga Next Steps) plus 4 disk of essential software.

Combining the Amiga A1200 Video, the Amiga Next Steps Insider Guide and the Amiga A1200 Insider Guide this bumper pack is the perfect gift for somebody you know taking their first tentative steps along the wonderful road of Amiga computing.

The disks of software contain the most sought after programs every beginner should have, including a database, a wordprocessor, a clip art selection, the OctaMed music sampler, a virus checker, a file recovery package and a disk compression utility.

If you already have one part of the pack then telephone us for an upgrade price.

Amiga Workbench 3.1 Booster Pack

ISBN: 1-873308-41-8, Price £39.95 plus £3 p&p, one and a half hour Workbench video, two books *(Workbench 3 A to Z* and *Amiga Disks & Drives)* a Quick Reference Card and a disk of essential software.

Already over 400,000 A1200 and CD[32] owners enjoy the power and versatility of Workbench 3. Now the million plus owners of A500, A2000 and other recent machines can enjoy the same power with a simple chip upgrade.

The *Amiga Workbench 3 Booster Pack* provides the most comprehensive support for such new users. The *Workbench 3 A to Z* book and the 90 minute *Amiga A1200 – A Deeper Look* video provide the complete guide in both tutorial and reference material to Workbench 3. The *Amiga Disks and Drives Insider Guide* goes on to take the new user to intermediate levels, showing how to optimise the use of their machines in both speed, capacity and security. All this, a disk of essential software *and* the Quick Reference card make it an essential purchase.

If you already have one part of the pack then telephone us for an upgrade price.

Introduction to the Amiga A1200 Video by Wall Street Video/BSB

BSBVIDAMI001, Price £14.99, one-hour Workbench basics video.

New from Bruce Smith Books in association with Wall Street Video – Australia's leading Amiga training company – the perfect video introduction to using your Amiga A1200 and a perfect companion for the world's best selling A1200 book, Bruce Smith's classic Amiga A1200 Insider Guide. This one hour video provides a basic tutorial on how to set up and run your Amiga A1200 by using great animations and split screens to increase your understanding or the concepts being explained. Re-examine those tricky grey areas by instantly rewinding the video.

Applicable to both hard and floppy disk users the Amiga A1200 Video may also by used to understand the Amiga A4000 and at £14.99 represents outstanding value.

Introduction to the Amiga A1200 – A Deeper Look Video by Wall Street Video/BSB

BSBVIDAMI002, Price £24.99, 90 minutes video.

The follow-up to the best-selling Introduction to the A1200 from Australia's Wall Street Video. Applicable to any Workbench 3 Amiga, this video goes beyond the first steps of using your machine to comprehensively tackle all the features of Workbench 3.

Mastering Amiga Beginners by Bruce Smith and Mark Webb

ISBN: 1-873308-17-5, Price £19.95, 320 pages. FREE Games disk.

Mastering Amiga Beginners is the book for the growing number of novice computer users who turn to the Amiga as the natural computer for home entertainment and self-education.

The authors have built up a wide experience of beginners' requirements and the problems they encounter and now this vast knowledge of the subject has been distilled into 320 pages of sensible advice and exciting ideas for using the Amiga.

Mastering Amiga System by Paul Overaa

ISBN: 1-873308-06-X, Price £29.95, 398 pages. FREE disk.

Serious Amiga programmers need to use the Amiga's operating system to write legal, portable and efficient programs. But it's not easy! Paul Overaa shares his experience in this introduction to system programming in the C language. The author keeps it specific and presents skeleton programs which are fully documented so that they can be followed by the newcomer to Amiga programming. The larger programs are fully-fledged examples which can serve as templates for the reader's own ideas as confidence is gained.

Mastering Amiga Printers by Robin Burton

ISBN: 1-873308-05-1, Price £19.95, 336 pages. FREE Programs disk

After reading Mastering Amiga Printers, any Amiga owner will be able to choose effectively the ideal printer for his or her requirements. The Amiga's own printer control software is pulled apart and explained from all points of view, from the Workbench to the operating system routines. Individual printer drivers are assessed and screen-dumping techniques explained.

Mastering Amiga AMOS by Phil South

ISBN: 1-873308-12-4, Price £19.95, 320 pages.

AMOS has very quickly developed into one of the most exciting and accessible programming languages on the Amiga. Its easy to use interface and familiar BASIC structure are augmented by powerful libraries for games and graphics programming. Mastering Amiga AMOS is ideal for anyone investing in AMOS, EasyAMOS or AMOS Professional. Full of hints, tips and shortcuts to effective and spectacular AMOS programming, this book also contains many useful routines and program design ideas.

Mastering Amiga Assembler by Paul Overaa

ISBN: 1-873308-11-6, Price £24.95, 416 pages. FREE disk.

The big brother to the Amiga Assembler Insider Guide, this book explains the use of assembly language to write efficient code within the unique environment of the Amiga, doing so without duplicating standard 68000 material in over 400 pages. Instruction is achieved by short code examples amidst discussion of the issues involved in using machine code for various purposes. Subjects covered include cooperation with the System software, custom chips and the C language. All the popular Amiga assemblers are supported by the many code examples in this book.

Mastering Amiga C by Paul Overaa

ISBN: 1-873308-04-6, Price £19.95, 320 pages.

FREE Programs Disk and NorthC Public Domain compiler.

C is one of the most powerful programming languages ever created with much of the Amiga's operating system written using C. The introductory text assumes no prior knowledge of C and covers all of the major compilers, including the charityware NorthC compiler supplied with this book when ordered direct from BSB. It is ideal for anyone using their Amiga to catch up on computer studies!

Mastering Amiga ARexx by Paul Overaa

ISBN: 1-873308-13-2, Price £21.95, 336 pages. FREE disk.

Now a standard part of Commodore's software strategy and readily available to Workbench 2 and 3 users, ARexx has been much admired by the programming community and is now available to all as a third party product. This book is an ideal companion to the ARexx documentation, explaining ARexx's main features, how it controls other programs, its built-in functions and support libraries, methods for creating well structured ARexx programs and much, much more.

Mastering Amiga Programming Secrets by Paul Overaa

ISBN: 1-873308-33-7, Price £TBA, TBA pages. FREE Programs Disk.

All the tricks and tips for programming your Amiga for graphics, animation and sound. Programs in assembler and C are provided with full documentation together with step by step tutorials to teach you how to program. Paul Overaa has saved up his best routines for what proves to be a dazzling guided tour around the best in Amiga programming techniques.

Amiga BASIC – A Dabhand Guide by Paul Fellows

ISBN: 1-873308-87-9, Price £17.95, 560 pages.

FREE Disk with ACE Freeware BASIC compiler.

BASIC is the computer programming language devised for beginners and now a standard on most computers. The Amiga doesn't usually come with a BASIC as standard but we provide one with the book so you have a head start. A number of commercial BASICs are available including HiSoft BASIC 2, True Basic and FBASIC. AMOS is also BASIC-like in its structures and keywords. This book is a substantial introduction to the language and is peppered with some of the cleverest routines around. Paul Fellows is a leading software author in his own right and his programming experience shines through in this easy to read guide. If you want to learn about programming in BASIC then this is the place to start.

Amiga Gamer's Guide by Dan Slingsby

ISBN: 1-873308-16-7, Price £14.95, 368 pages.

The latest book for the discerning Amiga owner is this highly illustrated guide to your favourite games. From sports sims to arcade adventures, Amiga Gamer's Guide author – and CU Amiga magazine editor – Dan Slingsby gives you the hints and tips, hidden screens and puzzle solutions you are looking for. Completed by a massive A to Z of tips and tricks for over 300 games, this is the most masterful of Amiga games guides yet published.

Secrets of Frontier Elite by Tony Dillon

ISBN: 1-873308-39-6, Price £9.95, 128 pages.

If you want to become Elite, or just incredibly rich, then get this book. This is the ultimate guide to the ultimate space trading game. Learn how to move up the ranks of the military, how to choose the best ships and weapons, how to trade and mine to the top. Games editor Tony Dillon has researched the game and included many of the hints and tips which have come his way. Find out how to gain control of the secret Mirage ship and how to become Elite, by the back door.

Disk Order Form

Please rush me a copy of Mastering AmigaDOS Scripts Disk.

I enclose a Cheque/Postal Order* for £2.00.

Name. ....

Address. ....

....

.... Post Code ....

Contact phone number. ....

*Cheques payable to *Bruce Smith Books Ltd.*

Send your order to:

**MAD Disk, Bruce Smith Books Ltd, PO Box 382,
St. Albans, Herts, AL2 3JD**

Please note that unless otherwise requested we will add you to our mailing list. This mailing list is currently only used to mail out to our readers details of new and forthcoming books. This includes our catalogue *Mastering Amiga News.*

Please take the time to answer the following questions:

How did you find out about Mastering AmigaDOS Scripts?

Where did you purchase your copy?

What other titles would you like to see in the Mastering Amiga range of books?

Mark Smiddy

Mastering AmigaDOS – Script Programs

AmigaDOS is the software built into your Commodore Amiga. It lets you write and run programs called *scripts. Mastering AmigaDOS Scripts* contains over one hundred ready-to-run script programs. There are script programs for AmigaDOS versions 3.x, 2.x and 1.x so this book is applicable to all Amigas, including the Amiga A1200, A600, A500 Plus, A500, A4000, A3000 and A2000 microcomputers.

The script programs are fully documented line by line so that you can learn from them, picking up the new techniques and programming twists which AmigaDOS guru Mark Smiddy has devised. Beginners will find the scripts easy to load and run, providing handy off-the-shelf utilities and full programs such as database and diary.

This is the third volume in the Mastering AmigaDOS series, which provides complete coverage of the Commodore Amiga's built-in software. Volume One – Mastering AmigaDOS Tutorial – is for Beginners, Volume Two – Mastering AmigaDOS A-Z Reference – documents every command with examples.

– Bruce Smith Books –

Publishers of the World's Best Selling Amiga Books

ISBN 1-873308-36-1

9 781873 308363

£19.95

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC-BY-SA) License. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866,
Mountain View, CA 940942, USA

Copyright:
Mark Smiddy 1994

Released under Creative Commons 2018