Adventures in Programming: init cdev

I had always wanted to create a cdev. But I never had good reason to do so. After all, a cdev is supposed to be some sort of enormously useful thing which you always want conveniently available, and too many cdevs cause a crowding of the control panel. Therefore it took some time until I decided upon a cdev which met my criteria of worthiness.

The cdev in point is, of course, “init cdev”. It is so called because it manages INIT, cdev and RDEV files. Such files are used by the “INIT 31 Mechanism” as occasionally referenced in Inside Macintosh on pages IV-256 and V-352.

The simple fact is that files of types ‘INIT’, ‘cdev’ and ‘RDEV’ (being Startup, Control Panel and Chooser files respectively) are examined, in alphabetical order, for any INIT resources contained within them. If such a resource exists, it is loaded and executed.

The only way to stop an INIT from loaded was to take the file out of the system folder. However, this always meant having to keep them somewhere temporarily and it was a nuisance to move files in and out of increasingly large system folders. That's where init cdev steps in.

The simple way of activating and deactivating INITs is to change their type. Therefore, INIT becomes, say, xNIT; cdev becomes xdev; RDEV becomes xDEV. This is ideal because INIT 31 ignores them, their programming code and resources are untouched, and the operation is easily reversible.

Such a technique was utilised in Aask, by CE Software. However, the one in general distribution is, in fact, an illegal ßeta copy and therefore does not actually exist. It was rather slow, especially when there were many files in the system folder.

init cdev manages to speed the process of searching for INIT resources by keeping a separate list of names of files to ‘exclude’ during the search. These files are typically cdevs which contain no INIT resources, such as the ‘Mouse’ cdev. By keeping this list up to date, init cdev will perform more efficiently.

The concept of making init cdev into a cdev as well as an INIT was to enable files to be activated or deactivated without restarting the machine. Of course, the changes only take effect *after* the next restart. Much of the programming code in the INIT and cdev is identical.

But enough of the background, and into the real adventure material. There is always a story behind a program, and ours is about to begin…

Having been the most complex INIT I had written to date, the program obviously caused me no end of trouble. The need to create a window from within an INIT is requires special initialisation of the Macintosh. While Inside Macintosh would have you believe it is necessary to call InitGraf, InitFonts, InitWindows, InitMenus and InitDialogs, the truth is that only InitWindows and InitDialogs is necessary (as determined by a painstaking process of trial and error). Be careful when other INITs load before yours, because they may have done some of the initialisation before you and your INIT may not work correctly on another system without those INITs.

The problem which I always run up against, however, was that the machine would

bomb at the most unexpected times whenever windows were in use. The actual point at which the crash occurred would also change on each restart. After extreme mental breakdown and massive heart failure, I finally discovered that it is of upmost importance to **lock all INIT code resources**. This prevents the INIT from being (re)moved whenever you do a memory-intensive operation such as creating a window. The Macintosh creates the window, then returns to your program which is no longer where the Macintosh thinks it should be. Result --> BOMB!

Another major disappointment came from closing the window. To quote Malcolm Fraser, past Prime Minister of Australia, “Life wasn't meant to be easy”. Whenever a DisposeWindow was issued, the machine would bomb (unimplemented instruction). Even a CloseWindow would bomb. AND a HideWindow. The problem was eventually traced to the global variable **DeskHook**.

Whenever the Macintosh needs to redraw the desktop, yet before it jumps to the normal routine to do so, it examines the **DeskHook** global variable for the address of a routine which wishes to override this process. The guys who made Macintosh wanted it to be an extremely flexible system. However, they never foresaw INITs at the time. As Mr Murphy would suggest, during the INIT phase this variable is somewhat undefined. The Macintosh pulls out an incorrect address, jumps to it and promptly dies.

To avoid this inconvenience, simply clear the variable. I simply wrote a routine to do it with standard code used in other window-using INITs. It is:

```
procedure ClrDeskhook;
inline  $42B8, $0A6C;
```

Once the windows were working, the actual programming could now take place. And more pearls of wisdom arose from this operation, too.

The list of INITs presented to the user is generated with the help of the **List Manager** (Inside Macintosh IV-259). It is a very free-form tool which does most of the hard work in creating and displaying lists. The default list routine handles text only. The text is passed via a pointer and a length field.

From Pascal, it is necessary to provide a pointer to standard text which is stored in Str255 format. This is relatively simple, and may be done so:

```
LSetCell(Ptr(ord(@Name) + 1), length(Name), theCell, theList);
```

The hard part comes in retrieving text from the List Manager, which is necessary if a private list is not kept by the program (which is poor usage of memory). While a pointer and length indicator is provided, just how can it be put back into Str255 format?

The text is easily retrieved via:

```
LGetCell(Ptr(ord(@Name) + 1), theLength, theCell, theList);
```

But how to set the length byte stored as the first byte in the Str255 variable? It is necessary to address it as a **byte** variable, but because it is only one byte long, in difference to the normal method of storing a byte in the low word of two bytes (got that?), as is the case with Lightspeed Pascal, a fake identity must be assigned to the string.

```
type
  fakeLengthType = packed array[1..2] of byte;
var
  fakeLengthPtr : ^fakeLengthType;
…
  fakeLengthPtr:=@Name;
  fakeLengthPtr^[1] := theLength;
```

This bit of code creates a pointer to the first byte, as a true byte because it is packed, and enables it to be dereferenced and changed. If there is any simpler method for doing this, I'd appreciate hearing about it.

As an aside, it might be worth mentioning how to outline default buttons. The Dialog Manager is nice enough to hit the default button when Return or Enter is pressed, but it is necessary to tell the user which button actually *is* the default. This information is, in fact, given within Inside Macintosh, but is located such that it is always impossible to locate when it is actually needed. But now the truth is revealed. Page I-407 says:

```
PenSize(3, 3);
InsetRect(displayRect, -4, -4);
FrameRoundRect(displayRect, 16, 16);
```

This draws the correct outline around the button having displayRect as its box. Please use it always.

Time for another problem… The list is presented in a dialog box. Lists are not standard items in a dialog box. Therefore, the ModalDialog command does not handle lists. What to do? You could totally avoid ModalDialog, but then you lose the very nice auto-highlighting of the default button and correct updating of the dialog.

A fake item, such as a UserItem, can be created and ModalDialog will report whenever it is clicked within. However, ModalDialog does not actually return the co-ordinates of the click, just the item hit. This is just fine for buttons, but what about lists?

It is therefore necessary to capture the event and get the MouseDown co-ordinates before ModalDialog gets at it. This can be done by performing EventAvail until a suitable event occurs. Keep a copy of the event handy, then call ModalDialog. If ModalDialog says a MouseDown happened in the UserItem, check the event record previously captured. Don't forget that the co-ordinates are global, and the List Manager requires local co-ordinates.

Another tip… if you are ever waiting for the user to click the mouse button to continue, such as when displaying “About”-like information in a cdev, don't forget to call GetNextEvent and probably SystemTask while waiting, otherwise things such as menu clocks and MultiFinder won't work. A good routine to use is:

```
repeat
   SystemTask
until GetNextEvent(mDownMask, theEvent) and (theEvent.what <> nullEvent);
```

This will wait until a MouseDown occurs, yet still repeatedly call SystemTask and GetNext Event. Even the control panel updates its cursor shape correctly during this repetition!

And now another sad tale, but with a happy ending. Just when I thought the program was finished (ain't it always so?), a terrible bug was pointed out. Whenever the scroll thumb was clicked in the INIT's list, the Macintosh bombed. And it wasn't my code because exactly the same code worked in the cdev. After much frustration and eager reading of Inside Macintosh, the problem was discovered to be another naughty global variable.

The variable **DragHook** performs much like DeskHook. Whenever something is dragged, such as a window or a scroll thumb, the routine pointed to by DragHook is called. Needless to say, it is also undefined at startup. It can be fixed by storing zero in DragHook, or:

```
procedure ClrDragHook;
inline $42B8, $09F6;
```

So ends the adventure of “init cdev”. I hope you enjoy it as much as I hated it.

— John Rotenstein

Identification:

Adventures In Programming #3: init cdev
Date: 13 November 1988

John Rotenstein
PO Box 165
Double Bay NSW 2028
AUSTRALIA.