Chapter 2 Code Resources (Inits,Cdevs,VBLs,...)

From: russotto@eng.umd.edu (Matthew T. Russotto)
Subject: Re: INITs and such...
Keywords: mac INIT

In article <7429@ur-cc.UUCP> kellogg@prodigal.psych.rochester.edu.UUCP (Lars Ke:

>Would some one please describe to me how to get an INIT to hang around in
>memory and do something WITHOUT having to patch an OS routine?

Sure-- just DetachResource and HNoPurge yourself (you can depend on A0 being a handle to your code, or, more safely, get a pointer to the beginning of the code and RecoverHandle it) I prefer to instead copy my code to the system heap, that way I don't need to copy installation code.

>For instance, continually check a specific folder for a specific file, and
>notify the user if it appears - I know this has been done, but it serves as
>an example. Can this be done without patching something?

I think so-- use asynchronous file manager calls, with a completion routine that re-queues the call if it failed (be sure to put a delay in though-- you may need a VBL task which re-queues the call to do this). Use the Notification manager to inform the user. (I think it's safe to do from completion routines)

>On a slightly different subject, is it even possible to patch an OS routine
>without using assembly language?

Sure, it can be done with stack-based routines with either C or PASCAL.

●●●

From: jackiw@cs.swarthmore.edu (Nick Jackiw)
Subject: Re: INITs and such...

kellogg@prodigal.psych.rochester.edu.UUCP (Lars Kellogg-Stedman) writes:

> Would some one please describe to me how to get an INIT to hang around in
> memory and do something WITHOUT having to patch an OS routine?

You can get it to hang around all you want by putting it into the system heap, either by _BlockMoving the relevant code from your INIT resource into a _NewPtr,SYS block or by loading + detaching the resource directly into the heap. Unfortunately, while hanging around, it won't *do* anything, because no part of the regular functioning of the Macintosh has been instructed to execute it periodically.

While the most common method of ensuring your INIT's code gets executed is by patching a trap (especially a trap the invocation of which is related to the purpose of your code--e. g. patch _MenuSelect if you want to preprocess menu displays), if you need periodic invocation regardless of which traps are invoked by the user, you could consider installing a VBL task. These are called by hardware interrupts, and are discussed in Inside Mac vol I, Chapter 11, and in vol V, chapters 24 and 29. Be sure your VBL task is installed in the system heap.

Unfortunately, VBL tasks must assume a very restricted environment, and therefore, cannot perform certain operations (allocating and disposing memory, for instance). Conversely, trap-patches can operate in whatever environment the trap is documented as assuming, which--in some cases--is quite flexible.

> For instance, continually check a specific folder for a specific file, and
> notify the user if it appears - I know this has been done, but it serves as
> an example. Can this be done without patching something?

I am the author of ChainMail, an INIT/cdev which polls a given folder for the existence of a specific file or the existence of any files. I designed ChainMail as a trap-patch (to _GetNextEvent) for the following reasons:

- Accessing the disk is difficult, if not impossible, from a VBL task.

- My task takes a noticeable amount of time. _GetNextEvent is not called in the middle of time-critical bits of code, and is generally assumed to mean that the application has little better to do than waste time.

  > On a slightly different subject, is it even possible to patch an OS routine
  > without using assembly language?

One is strongly advised against it. On the otherhand, patching a trap takes a *minimum* of assembly, and can be done in line. The actual code executed by the patch may be written in any language you chose, if your compiler can generate code resources.

> ~ ~ | Lars Kellogg-Stedman | "Software rots if not used"

Nick Jackiw jackiw@cs.swarthmore.edu "Just break out the

●●●

From: Lawrence D'Oliveiro
Subject: Re: INITs and such...

Just to add my bit to everybody else's comments on this.

There are two mechanisms for queueing an operation to occur at a specific time: installing a VBL task, or putting an entry into the Time Manager queue. In both cases, your code would get called at interrupt level, so it can't do anything that would involve the memory manager (file and device I/O is OK).

If you need to do some operation involving allocating/deallocating/locking/ unlocking memory, you can queue it from your interrupt-level code for later execution via the Notification Manager. Remember, you don't actually need to post any notifications to the user if you don't want to: you can use the Notification Manager simply as a safe, clean way of queueing work to be done at Get/WaitNextEvent time.

Lawrence D'Oliveiro

●●●

From:Charles Martin
Subject :Re:How do you write a INIT in PASCAL?

Isn't it amazing? No advice anywhere. Try this in THINK Pascal, the basic idea is to push the old trap address on the stack right before your patch by modifying the header that THINK Pascal puts on your CODE resource. If someone sees any grave errors or omissions (such as, what is the role of the 'sysz' resource, and does it matter that the address of the trap won't be in ToolScratch), please let me know!

(Of course there's a little implicit assembly here, it's inescapable!)

----- beep.p: code resource, CODE 128, locked, system heap, preload

```
unit beep;
interface
 procedure main;
implementation
 procedure main;
   begin
     SysBeep (1)
   end;
end.
```

----- install.p: code resource, INIT 0, locked; use resource file beep

```
unit install;
interface
 procedure main;
implementation
 const
   YourTrapGoesHere = $A9xx;
```

```
type
  long_hdl = ^longptr;
  long_hdl = ^long_ptr;
  long_t = array [0..0] of longint;
procedure main;
```

```
   var beep: Handle;
       long: long_hdl;
       addr: longint;
   begin
     beep := Get1Resource ('CODE', 128);
     DetachResource (beep);
     long := long_hdl (beep);
     addr := GetTrapAddress (YourTrapGoesHere);
     long^^ [5] := $2F3C; { MOVE.L <addr>,-(A7) }
     long^^ [6] := addr;
     SetTrapAddress (longint (beep^) + 22, YourTrapGoesHere)
   end;

end.
```

Charles Martin // martin@gargoyle.uchicago.edu
In-reply-to: mxmora@unix.SRI.COM (Matt Mora)

In article <11608@unix.SRI.COM>, mxmora@unix (Matt Mora) writes:

>What I am really looking for is any examples in PASCAL.

Sorry, there were a couple of typos. The correct lines are:

```
   long_hdl = ^long_ptr;
     long^^ [5] := $2F3C; { MOVE.L <addr>,-(A7) }
     long^^ [6] := addr;
```

Charles Martin // martin@gargoyle.uchicago.edu

●●●

From: Scott A. Mason
Subject: .c2.How to write a vbltask(with code)

In article <604@mit-amt.MEDIA.MIT.EDU> adam@mit-amt.MEDIA.MIT.EDU (Adam Glass) :

>I've been having the darndest time trying to make a program to do a VInstall.
>I can't seem to be able to get the PASCAL variant records/C unions to work.
>Could some kind soul mail me a procedure/quickie program to do it? You could
>also post it, as I'm sure other people will find it interesting.

Ok, here's a snippet of my code written in LSC. (It doesn't do much, but should help a lot of people getting things to work.)

```
#include <VRetraceMgr.h>

VBLTask vblTask;

void VBL_Routine ()
{
       SetUpA5();               /* set up for globals */
       vblTask.vblCount = 30;  /* reset the count so we run next time */
/* your code to do something real goes here */
       RestoreA5();             /* reset the globals stuff */
}

/* This routine installs the given VBLTask */

Set_Interrupts (vblTask)
VBLTask *vblTask;
{
```

```
OSErr err;

vblTask->qType          = vType;
```

```
       vblTask->vblAddr        = (ProcPtr) VBL_Routine;
       vblTask->vblCount       = 30;
       vblTask->vblPhase       = 0;
       err = VInstall ( vblTask );
}
```

Well, there it is in all its simplicity. I must say it was difficult the first time for me as well. The implementation of VBL tasks is not very clear in IM. Hope this helps.
--

Scott A. Mason

●●●

From: sdh@flash.bellcore.com (Stephen D Hawley)
Subject: .c2.How to write WDEFs (with code)

I tried to mail this but it bounced. Here's what you need to do to write a WDEF:
Create a new THINK C Project, set the project type to be WDEF. Set the resource number to something nice for you.

The main should look like this:

```
pascal long main(varCode, theWindow, message, param)
short varCode;
WindowPtr theWindow;
short message;
long param;
{

   /*
    * This is probably the most straight forward approach
    * to implement the function.
    * return a long as appropriate for each part of the
    * window.
    */
   switch (message) {
   case wDraw:
      break;
   case wHit:
      break;
   case wCalcRgns:
      break;
   case wNew:
      break;
   case wGrow:
      break;
   case wDrawGIcon:
      break;
   }
}
```

Fill in the blanks as you need, and compile it into a WDEF with "Build Code Resource".

To debug, write a small project that creates a bunch of windows (using the resource number you specified before, allowing for variation codes etc) and lets you do all the playing with them. Then make sure you WDEF is put into a file called whatevermysmallprojectiscalled.rsrc where "whatevermysmallprojectiscalled" is the name of the small shell application.

When you want to start using it in a real program, use ResEdit to copy it into your project's resource file, or use the Merge option in Think C to merge it into your project.

Check out the Think C manual (4.0 p. 84-86) for more details.

Good Luck!

Steve Hawley

●●●

From: cak3g@astsun7.astro.Virginia.EDU (Colin Klipsch)
Subject: Re: Help! (jGNEFilter, GetNextEvent, events)

In article <14463@reed.UUCP> chaffee@reed.UUCP (Alex Chaffee) writes:

>In article <25ff4e70.643e@polyslo.CalPoly.EDU> rcfische@polyslo.CalPoly.EDU
><in the menu, even when the menu gets re-drawn. Then you might want
><to look at hooking into GetNextEvent so that you can intercept mouse
><downs and see if they're on your icon.
>
>A small suggestion - rather than patching GetNextEvent, take advantage of
>the low-memory global jGNEFilter, which is sort of a built-in tail patch on
>GNE. It's documented in a tech note (I can't remember the number); I could
>give you source code if you're interested.

><Ray Fischer
>--
>Alex Chaffee

Tech Note #85 talks about jGNEFilter. It is a low-memory global which points to code that will be executed upon every event, after the Mac has done some of its own processing. I used to use this method of event filtering for TappyType, and it worked as follows. . .

In your INIT:

• Load your event-filter code, presumably stored as another resource, into the system heap.
• Make sure it's locked. (i.e. set its resLocked bit beforehand)
• Detach it with _DetachResource.
• Take the whatever pointer you find in jGNEfilter and store it somewhere safe in your event-filter.
• Store the address of your event-filter in jGNEfilter.

Now every time an event happens, your event-filter (tucked safely inside the system heap) will be called. Hopefully.

In your event-filter:

• Upon entry, A1 should point to the event record. Do whatever you want to do with the event.
• After you've done your thing, jump to the old value of jGNEfilter that you saved previously.

In effect, you are "splicing" your event-filter into the flow of event processing. If everyone were to do the same thing, then many different event filters could co-exist.

Two Bad Things: Unfortunately, not everyone plays nicely, and thus it may matter when your INIT runs in relation to others. Secondly, jGNEfilter is a low memory global, and thus it is not guaranteed to be around in the long run.

Patching GetNextEvent, as a method of event filtering, won't work except as a tail patch, which is a Thing Not To Be Done.

The scheme I use now is head-patching SystemEvent, which also seems to be called after every event. SystemEvent is "guaranteed" to be supported, and you don't have to share a precarious memory global with random strangers.

Colin Klipsch

●●●

From: shebanow@Apple.COM (Andrew Shebanow)

Subject: Re: Info on Tail patches

Tail patches are trap patchs which do processing after calling the original trap, or modify the stack before calling the original patch. They're called tail patches because the typically look like this:

```
LEA    origTrapAddress,A0
; do preprocessing
JSR    (A0)
; do postprocessing
```

Since the bad part is at the end of the patch, its a tail patch.

A "clean" patch will have a format like this:

```
; do processing
LEA    origTrapAddr,A0
JMP    (A0)
```

Tail patching is bad because of the techniques that Apple uses in its System Software to fix bugs. Sometimes, a trap (lets call it _TrapA) has bugs in it which would be very difficult to fix without rewriting the trap's entire code. Rather than do that, Apple will sometimes patch a different trap (_TrapB) which is called by the broken trap (_TrapA). _TrapB will check the address of the caller on the stack and compare it to the hardcoded address of _TrapA's call, and, if the addresses match, behave in a different manner to fix the bug. This can cause a huge savings in speed and memory usage, but it prevents the use of tail patches. Continuing our example, if you patch _TrapA and then JSR to the original _TrapA code, you will have changed the contents of the stack. And believe it or not, some of the patches are involved enough that they look two or three stack levels high.

Hope this clears it up,

Andy Shebanow   Mr. Clean, MacDTS   Apple Computer, Inc.

●●●

From: parent@apple.com (Sean Parent)
Subject: Re: Tail patches

I have written lots of patches of all kinds in the past and I can tell you that they are not simple. If you are writing an application then there is no good reason to patch a trap that I have been able to determine... so don't. If you are writing an INIT, or some funky piece of psuedo-system software then you are walking a fine compatibility line to begin with and you should try to avoid patches.

Take a close look at your code to see if there is some way around the problem. For example: If you are writing an INIT and you wish to display an alert after the system has started up then don't patch SystemTask or GetNextEvent but use the notification manager. Or, if you just need some time then install a driver and get idle time that way.

If you decide you must patch a trap then try to avoid tail patches for the reasons mentioned previously. Patches to ROM routines do depend on return address (I know, I wrote one that does). With any kind of patch the rule is "take only picture, leave only footprints." Live by it. Restore EVERYTHING you can that you disturb. When chaining to the next routine restore ALL registers to their previous values, this can be done by reserving space on the stack at the start of your routine and moving the address that you will continue to into this space. At the end of your patch then restore all registers and do an RTS. The reason for this is that there are patches that upon checking the return address make assumptions about what is in the registers at that particular moment.

Be aware that:

- Multifinder keeps separate trap tables for each application.
- On 64K ROM machines the patch must be in the first 64K of RAM.
- If you remove your patch someone may have patched it after you so don't remove their patch also.
- Some traps dispatch to the same routine. For example _write async, and _write sync, dispatch to the same routine with the trap number in register D1 so you can determine which call was being made.

- Your patch may NEVER get called if something in the system (like Multifinder) replaces the routine entirely.

If you do write patches be prepared for your software to break even if you write it very carefully. Warn your employer and understand that you need a good update service in place to fix problems that may arise with the next release of system software. Build a maintenance clause into your contract if you are working as a contractor.

And before you write the patch, send DTS a link. They are pretty good at talking people off the roof. Give them a chance.

Sean Parent

●●●

From: tim@hoptoad.uucp (Tim Maroney)
Subject: Re: Menu font list appearing as font (source to MDEF)

In article <32498@ucbvax.BERKELEY.EDU> oster@dewey.soe.berkeley.edu.UUCP David Phillip Oster) writes:

>To have a list of font names, each name in its own font (for example the
>string "New York" in the New york font.) you need your own MDEF. It isn't
>worth writing though, since there are at least two commercial products
>that do this for you, uniformly, at INIT time. MenuFont 2, and Font/DA
>Juggler, I think, are their names. It is much better to do this at INIT
>time, since it is very slow to read in the font data, and create the
>bitmaps. An INIT can pre-render all the fonts, and save them off to disk,
>and only build new ones when there are changes.

This is a good idea; it can't be stressed enough that you should only do the menu rebuilding when there are changes in the available fonts,though. I'd hate to have to wait for an INIT to do this every time I restarted.

I think Claris has the right idea. All their programs with Font menus save them in a common data structure and a common file in the system folder, and each application comes with the necessary code to check the font list stored in that file against the current font list, rebuild the menu if necessary, and display and track the menu. It would be nice if someone would put code to do this in the public domain, or if Apple were to include it in system software. Probably the best way to do it would be to use the Resource Manager to store the data structures. The menu itself could be stored in a PICT, and the rectangles for each font name would be stored in an 'nrct'; a 'STR#' would hold the alphabetized list of font names.

Here's an old PICT MDEF I wrote. The extension to variable-sized elements should be pretty easy. Anyone else have useful code they can contribute to this cause?

```
#define ItemHeight 16

PASCAL void
mdef(message, menuid, r, p, which)
short message;
short **menuid;
Rect *r;
long p;
short *which;
{
	PicHandle pic = GetPicture(**menuid);
	short new;
	Rect invert;
	Point pt;

	switch (message) {
	case 0:
		/* draw the menu */
		DrawPicture(pic, r);
		break;
	case 1:
		/* choose and hilight */
		BlockMove((Ptr)&p, (Ptr)&pt, 4);
		if (PtInRect(pt, r)) {
```

```
/* find the item */
new = ((pt.v - r->top) / ItemHeight) + 1;
```

```
                    if (*which != new) {
                          if (*which != 0) {
                                /* unhighlight which */
                                SetRect(&invert, r->left,
                                      r->top + ((*which - 1) * ItemHeight),
                                      r->right, r->top + (*which * ItemHeight));
                                InvertRect(&invert);
                          }
                          /* highlight the item */
                          SetRect(&invert, r->left, r->top + ((new - 1) * ItemHeight),
                                r->right, r->top + (new * ItemHeight));
                          InvertRect(&invert);
                          *which = new;
                    }
              }
              else if (*which != 0) {
                    /* unhighlight which */
                    SetRect(&invert, r->left, r->top + ((*which - 1) * ItemHeight),
                          r->right, r->top + (*which * ItemHeight));
                    InvertRect(&invert);
                    *which = 0;
              }
              break;
        case 2:
              /* calculate size */
              (*menuid)[1] = (*pic)->picFrame.right - (*pic)->picFrame.left;
              (*menuid)[2] = (*pic)->picFrame.bottom - (*pic)->picFrame.top;
              break;
        }
}
```

>If the INIT is also a
>control panel device, it can give the user a place to say that he wants to
>turn the mechanism off for unusual fonts that don't contain legible
>characters for their own name. Symbol is one such font, but any upper case
>only font with a mixed-case name would qualify.

Hmm. I don't know that this rule is correct. Might there not be symbol fonts that were not upper-case-only? And I confess that I'm mystified by what the mixed-case font name has to do with it.

Also, there are some Script Manager compatibility issues here that aren't clear to me. Perhaps some nice person at Claris could tell us how their applications deal with Symbol font and others of like kind, and how they deal with the international environment.

>I've written this feature into at least two prototype products that never
>saw the light of day. If you must do it, do it as an INIT, make it work
>everywhere, and compete with the other guys.

Yeah, well, I think the last thing we need is more INITs. Especially more overpriced single-function INITs. I'd rather have someone give me a code resource or two that I can just plug into my program and fly with.
--
Tim Maroney, Mac Software Consultant, sun!hoptoad!tim, tim@toad.com

●●●

From: mahboud@kinetics.com (Mahboud Zabetian,Kinetics,4511,9457194,)
Subject: Re: How can a CDEV "talk" to its INIT?

From article <8183@cs.yale.edu>, by kishon-amir@CS.Yale.EDU (amir kishon):

> I would like to write a CDEV that can change the global variables of a

```
> patch to GetNextEvent (the patch was done by an INIT that I wrote).
> How can I have both the CDEV and the patch (INIT) point to the same
> global variables, so the CDEV can change those values?  Or are there
> any other ways to do that? namely, to keep some static global
```

> parameters referenced both by a CDEV code resource and an INIT code
> resource? I am working with Think C 4.0. HELPPPPPPPPPPPPPPPPPPPPPPP
>
> Thanks,
>
> -amir

Make the first instruction in your INIT be a JMP to the third instruction. Make the second instruction be a JMP to code that will set up your globals for you(these globals will probably be imbedded in your patch).

Now when you change the settings in the cdev, load in your INIT resource, and jump to 2(A0) where A0 is the beginning of the code.

Or if you can wait until reboot before changing things around, then just store that info in a resource and read it on the next reboot.

Mahboud Zabetian mahboud@kinetics.com

●●●

From: jholt@adobe.COM (Joe Holt)
Subject: Re: How do I install a driver with an INIT? (with Code)
Keywords: INITs driver

Here's a small code resource INIT I use to open a driver at startup time. I write the guts of my "INIT" as a driver, then paste this resource into it, and I'm off. With Think C, it even handles multi-segmented drivers. If you have any questions, just ask.

-------- CUT HERE --------

```
/**

 OPEN DRIVER INIT

 Think C 4.0

 This bit of code should be compiled as a Code Resource, Type INIT, ID 0 (or
 whatever you like).  Change the name DRIVER_NAME to the name of the driver
 you want opened at start up.  Place this compiled INIT resource in the
 driver and change the driver's file type to INIT.

 When this starts up, it finds an empty slot in the driver unit table and
 sticks your driver there.  It then opens your driver.  You should write the
 driver to handle whatever ref num it's assigned.

**/


/**--------------------------------------------------------------------------
**
**    Compilation Flags
**
**/

/**

 MULTI_SEGMENT generates code which is needed to INIT a multi-segmented
 driver.  Single-segment drivers can be inited also, so there is no logical
 requirement to ever turn this off, except where code size is a concern.

**/
```

```
#define MULTI_SEGMENT    1
```

```
/**-----------------------------------------------------------------------------
**
**    Include Files
**
**/

   /* none */


/**-----------------------------------------------------------------------------
**
**    Private Macros
**
**/

/**

 UNIT_ENTRIES_NEEDED and FIRST_POSSIBLE_UNIT are used when determining the
 driver reference number for your driver.  This assigns the driver reference
 number dynamically, looking at FIRST_POSSIBLE_UNIT and on up for an unused
 number.  It increases the unit table size to UNIT_ENTRIES_NEEDED if
 there aren't that many unit numbers.  This occurs on Mac Pluses and Mac SEs
 with pre-System 6.0, where the unit table defaults to forty-eight entries,
 which doesn't leave any room for custom drivers like ours.

**/

#define UNIT_ENTRIES_NEEDED     (64)
#define FIRST_POSSIBLE_UNIT     (48)

#define DRIVER_NAME             "\p.your name here"


/**-----------------------------------------------------------------------------
**
**    Private Functions
**
**/

int main(void);

main()
{
   static char     driverName[] = DRIVER_NAME;
   int             theID;
   IOParam         ioPB;

   asm {
           movem.l D3-D4/A2-A4, -(A7)
           movea.l A0, A4


/**

 Change the resource attributes of our DRVR so that it will be loaded into
 the System Heap and locked.

**/

           clr.b   -(A7)
           _SetResLoad

           clr.l   -(A7)
```

```
        move.l  #'DRVR', -(A7)
        pea     driverName
        _GetNamedResource
```

```
           move.l  (A7)+, D3
           beq     @abort

           move.l  D3, -(A7)
           move.w  #resSysHeap + resLocked, -(A7)
           _SetResAttrs
           move.l  D3, -(A7)      ; so that the res handle will be in SysHeap!
           _ReleaseResource

           move.b  #1, -(A7)
           _SetResLoad


/**

 Pick an unused driver reference num for our DRVR and renumber the resource
 to use this new reference number.  We first increase the number of entries
 in the unit table if less than UNIT_ENTRIES_NEEDED, then begin looking at
 FIRST_POSSIBLE_UNIT.

**/

           clr.l   -(A7)
           move.l  #'DRVR', -(A7)
           pea     driverName
           _GetNamedResource
           move.l  (A7)+, D3
           beq     @abort

           move.l  D3, A3
           move.l  A3, -(A7)
           _DetachResource
           move.w  #UNIT_ENTRIES_NEEDED, D1
           sub.w   UnitNtryCnt, D1
           ble.s   @1

           bsr     @updrivers
           bne     @abort

@1:         movea.l UTableBase, A1
           move.w  #FIRST_POSSIBLE_UNIT * 4, D4
           move.w  #FIRST_POSSIBLE_UNIT, D3
@2:         cmp.w   UnitNtryCnt, D3
           bgt     @abort

           tst.l   0(A1, D4.w)
           beq.s   @3

           addq.w  #4, D4
           addq.w  #1, D3
           bra.s   @2

@3:         move.w  D3, D0
           not.w   D0
           movea.l (A3), A0
           dc.w    0xA43D              ; _DrvrInstall 4
           bne     @abort

           movea.l UTableBase, A1
           movea.l 0(A1, D4.w), A0
           _HLock
           movea.l (A0), A2
           move.l  A3, (A2)
```

```
movea.l (A3), A3
move.w  (A3), 4(A2)
bset    #6, 5(A2)
```

```
/**

 Now set the resources which THINK C uses for drivers to load into the System
 Heap as locked, also.  We especially want to lock down any DCOD resources
 because the THINK C segment loader does a MoveHHi() on em when first loaded,
 which is not a good idea in the System Heap at INIT time.

**/

           clr.b   -(A7)
           _SetResLoad

           clr.l   -(A7)
           move.l  #'DATA', -(A7)
           move.w  #1, -(A7)
           _Get1IndResource
           move.l  (A7)+, D3
           beq     @abort

           move.l  D3, -(A7)
           move.w  0x18(A2), D0
           not.w   D0
           lsl.w   #5, D0
           or.w    #0xC000, D0             ; DATA resource is always ID 0
           move.w  D0, -(A7)
           clr.l   -(A7)
           _SetResInfo
           move.l  D3, -(A7)
           move.w  #resSysHeap + resLocked, -(A7)
           _SetResAttrs
           move.l  D3, -(A7)
           _ReleaseResource


#if MULTI_SEGMENT

           subq.w  #2, A7
           move.l  #'DCOD', -(A7)
           _Count1Resources
           move.w  (A7)+, D4
           beq.s   @DCODout

DCODtop:    clr.l   -(A7)
           move.l  #'DCOD', -(A7)
           move.w  D4, -(A7)
           _Get1IndResource
           move.l  (A7)+, D3
           beq     @abort

           move.l  D3, -(A7)
           pea     theID
           clr.l   -(A7)
           clr.l   -(A7)
           _GetResInfo

           move.l  D3, -(A7)
           move.w  0x18(A2), D0
           not.w   D0
           lsl.w   #5, D0
           move.w  theID, D1
           and.w   #0xF01F, D1
```

```
        add.w   D1, D0
        move.w  D0, -(A7)
        clr.l   -(A7)
```

```
           _SetResInfo
           move.l  D3, -(A7)
           move.w  #resSysHeap + resLocked, -(A7)
           _SetResAttrs
           move.l  D3, -(A7)
           _ReleaseResource
           subq.w  #1, D4
           bne.s   @DCODtop

#endif    /* if MULTI_SEGMENT */

DCODout:    move.b  #1, -(A7)
           _SetResLoad

           bra.s   @open


;**
;
;    Increase the number of available entries in the unit table.
;
;**

@updrivers: move.w  UnitNtryCnt, D0
           add.w   D1, D0
           mulu    #4, D0
           _NewPtr SYS+CLEAR
           bne.s   @9

           move    SR, -(A7)
           move    #0x2600, SR
           movea.l A0, A1
           movea.l UTableBase, A0
           move.w  UnitNtryCnt, D0
           mulu    #4, D0
           _BlockMove
           _DisposPtr
           move.l  A1, UTableBase
           add.w   D1, UnitNtryCnt
           move    (A7)+, SR
           moveq   #0, D0
@9:         rts
   }
open:
   ioPB.ioNamePtr = (StringPtr)driverName;
   ioPB.ioPermssn = 0;
   PBOpen(&ioPB, FALSE);
abort:
   asm {
           movem.l (A7)+, D3-D4/A2-A4
   }
}
```

•••

From: t-alexc@microsoft.UUCP (Alex CHAFFEE)
Subject: Re: cdev - INIT Data Exchange
Keywords: cdev, INIT

In article <2695@cooper.cooper.EDU> joseph@cooper.cooper.EDU (Joe Giannuzzi) writes:

```
>       I am writing a cdev that contains an INIT. I would like to
>       exchange data between the two, but so far my attempts have
>       failed. Does anyone have any recommendations about the best
>       way to do this? Also, I have one DITL item that is an ICON.
```

> What is the best way for me to change the icon that it
> displays? Thanks.
>
> Joseph -> joseph@cooper.cooper.edu OR cmcl2!cooper!joseph

My favorite way is to have the INIT write a resource of type 'ADDR' into the preferences file (which should be buried safe in the system folder). The contents of this resource is just a longword pointing to the location of the shared data -- it can be a handle, or a pointer, or the proper value of A4. This is done once at startup, which is fine because the INIT code and data should be locked in memory for the duration..

This sounds kludgey, but it's actually quite clean and compatible. The INIT is guaranteed a writable system folder, even if it's been launched from a server. The existence and validity of the resource provides a quick way for the cdev to tell if the INIT's been loaded. The preferences file is going to keep the same name, even if the user renames the actual cdev/INIT file, so it'll be easy to find. Plus you don't have to walk through the system heap map, or install a driver, or any of the 1001 nasty alternative methods I've heard of...

As for your ICON, can't you just set the itemHandle to the new ICON resource (after GetResourceing it in from disk)? I thought that was how the Dialog Manager dealt with ICONs, but I'm not sure.

- Alex Chaffee
chaffee@reed.bitnet or t-alexc@microsoft.uucp

•••

From: lsr@Apple.COM (Larry Rosenstein)
Subject: Re: cdev - INIT Data Exchange

In article <1990Jun5.142604.11826@asterix.drev.dnd.ca> louis@asterix.drev.dnd.ca (Louis Demers) writes:

> What happens if you disable the INIT and it is not loaded. Your
> resource is still there pointing to some area it has no right to

I would place a magic series of bytes in memory and have the cdev check that the resource points to the expected magic bytes.

I've used this technique before, but users complained that the file in their system folder was modified each time they booted, so it was always being backed up.

Larry Rosenstein, Apple Computer, Inc.

•••

From: cak3g@astsun7.astro.Virginia.EDU (Colin Klipsch)
Subject: CDEV/INIT Data Exchange, another way
Keywords: cdev, INIT, system heap

In article <8560@goofy.Apple.COM> lsr@Apple.COM (Larry Rosenstein) writes:

>In article <1990Jun5.142604.11826@asterix.drev.dnd.ca>
>louis@asterix.drev.dnd.ca (Louis Demers) writes:
>> What happens if you disable the INIT and it is not loaded. Your
>> resource is still there pointing to some area it has no right to
>
>I would place a magic series of bytes in memory and have the cdev check
>that the resource points to the expected magic bytes.
>
>I've used this technique before, but users complained that the file in
>their system folder was modified each time they booted, so it was always
>being backed up.

Here's my experience in writing TappyType, for what it's worth. . .

The first technique I used was the one stated above: the INIT, on startup would store a pointer to my system heap variables in a resource of type "TapT", ID#2 (if I remember correctly). This has the advantage of being more "compatible" than just about any other solution I've heard of, but which has the above disadvantage. Plus, if the user is especially mischievous and does things like: locks the CDEV file, temporarily moves it to another folder, deletes the resource, etc., you've got problems. Using an entire auxiliary file for CDEV-INIT communication is subject to the same problems, and more.

Of course, you can argue that anyone pretentious enough to commit any of these crimes against your CDEV deserves what he or she gets.

The next method I used -- briefly -- was searching the system heap for a block of the right size, and which began with a particular magic string. After rereading the Memory Manager chapter and various tech notes, however, I soon realized that this is a BAD idea. It relies completely on the format of memory blocks, which very very very probably will change in the future at arbitrary times. I suppose you could search the system heap for a magic string without paying attention to block boundaries, but this still seems rather iffy.

The current solution is a bit of a hack, but requires no auxiliary resources, nor installing a driver:

The INIT installs the code and variables in system heap memory from a resource (remember to use DetachResource!), and patches a few traps. Among them is _AddResource. For CDEV-INIT communication, I needed a trap which could be called in some way that my patch could uniquely identify as being a call from the CDEV.

My patch looks at every AddResource call. If the call is made with a resource of my particular type, my particular ID, if the name pointer points to four bytes of zeros, and if the return address is near a particular magic string, then my patch assumes the call was made by my CDEV. It then puts a pointer to my system heap variables in the four bytes of zeros pointed to by the name pointer, which my CDEV can pick up and use. The patch ends by jumping to the original AddResource address that the INIT found on startup (with NGetTrapAddress).

Note that adding a resource which already exists is harmless, so the resource I "add" from the CDEV is just my options resource, which my CDEV has previously loaded anyway. If TappyType has not been installed, then my patch won't be there, and the the four bytes of zeros will remain just that after the call. Otherwise, if it's non-zero, it's a pointer to my system heap variables.

To be even more anal-retentive, one could do the following:

- check that the handle being added is a resource, and that it's the right size
- check to see that the current resource file is of type "cdev" and creator "TapT" (using GetFileInfo)
- pass a pointer to four zero bytes as before, but after the four bytes is also a pointer to the magic string; check for it

In fact, I like that last one much better, now that I thought of it. I think I'll rewrite it that way for the next version. . .

Hope this helps. AddResource is undoubtedly not the only trap you could use; it's just the first feasible one I thought of. The time overhead is low, particularly if you write it in assembly (as one should probably).

I invite (constructive) criticism on this method. Anyone from Apple see anything wrong with this? (Anyone from anywhere, for that matter?)

Colin Klipsch

●●●

From: murat@farcomp.UUCP (Murat Konar)
Subject: Re: cdev - INIT Data Exchange
Keywords: cdev, INIT

In article <1990Jun5.142604.11826@asterix.drev.dnd.ca> louis@asterix.drev.dnd.ca (Louis Demers) writes:

```
>t-alexc@microsoft.UUCP (Alex CHAFFEE) writes:
>
>>My favorite way is to have the INIT write a resource of type 'ADDR'
>>into the preferences file (which should be buried safe in the system
>>folder).  The contents of this resource is just a longword pointing
>>to the location of the shared data -- it can be a handle, or a
>>pointer, or the proper value of A4.  This is done once at startup,
[...]
```

```
>   What happens if you disable the INIT and it is not loaded.  Your
>   resource is still there pointing to some area it has no right to
```

> claim. Upon shutdown, Do you remove this resource ? what happens
[...]

I embed a 4 character string (OSType) into my patch code at a known offset from its address and check for it before writing there. Works really well.

--
Murat N. Konar

●●●

From: urlichs@smurf.sub.org (Matthias Urlichs)
Subject: Re: CDEV/INIT Data Exchange, another way
Keywords: cdev, INIT, system heap

In comp.sys.mac.programmer, article <55103@microsoft.UUCP>, benw@microsoft.UUCP (Ben WALDMAN) writes:

< On cdev-INIT communication:

<   The way I do it is to write my what's really my INIT as a driver.
< Then, the actual INIT resource opens the driver by name, putting into the
< system heap. (And the driver's open routine patches the traps I want to
< patch, etc.). The driver also provides a status call, which returns the
< address of its globals.
<
<   The cdev, when it wants to communicate with the INIT, can simply
< look through the unit table (this is described in a tech note), and find
< the driver (by name). Then, the cdev can make a status call to the driver,

When you already know the driver's name, why not do an OpenDriver(name)? That'll get you its refnum much easier and safer (WRT compatibility). Apple specifically recommends not to scan the unit table if at all possible.

< and, voila, the status call returns the info the driver needs. In my case,
< I returned a pointer to the INITs globals (the INIT is locked in memory),
< but you could, of course, return a handle, or an address of a function you
< wanted to call, etc.

If you want the driver to do anything, it'd be much cleaner just to call the driver through _Control and/or _Status calls, no? That way the whole thing will also work if the user boots with version X of your driver, then installs Y (not necessarily greater than X), and opens the control panel...

<   The scheme fails if a dorky user changes the name of the DRVR resource
< with ResEdit, but will still succeed if the DRVR gets renumbered.

You'll have to check in the unit table for a free refnum and install your driver there. You may have to make the unit table bigger; don't forget to zero the new table, record the new size in the appropriate global, don't free the old table because you don't know where it came from, and turn off interrupts while you do it.
--
Matthias Urlichs -- urlichs@smurf.sub.org -- urlichs@smurf.ira.uka.de

●●●

From: jholt@adobe.COM (Joe Holt)
Subject: Informative INITs (code incl.; was: Re: Need help w/ Public Folder)

michael wrote:

> ASIDE: It sure seems to me that INIT's need some way to communicate with
> users other than just X-ing themselves out. I've been thinking about a
> mechanism where if there is a problem the user can read about it in the

```
> Chooser (or Control Panel) when they bring up the normal UI.  The UI
> would say something like, "Couldn't find a Public Folder" or "AppleTalk
> is not turned on."  What do people think of this idea?
```

I ran into the totally uninformative nature of x'ing the INIT icon out at boot time while writing Flash (an AppleTalk file transfer enhancement -- similar to Public Folder -- hello michael, nice to meet you!). I believe there are on the order of forty or fifty different reasons why Flash wouldn't install itself. Everything from "AppleTalk not installed" and "Yer system's too old" to "out of memory" and "Flash is damaged". A lot of help an X is when joe user is trying to track down the problem.

Of course, bringing up an error window at INIT time is not only contrary to guidelines and tough to do but also just plain ugly.

I solved the problem by using Apple's Notification Manager.

If Flash has a problem, it gets a string from a STR# resource which describes the problem and puts it into a pointer in the system heap. It also loads a very small code resource and creates a Notification Manager structure. The code resource is a NM response procedure (read "completion routine") which gets rid of the structure and string and then itself.

Flash sets up the NM structure and posts a NM message. Flash then cleans up its act like a good boy scout and leaves no trace of itself. At this point the only things left around are the string, the code resource (detached) and the NM structure.

The NM message stays dormant until the Mac has finished booting and begins processing events (most likely in the Finder). Then the NM displays the message in an alert and the user gets a nice informative message. (If you have Flash and want to see this, take the file "ADSP" out of your system folder and reboot.)

When the user closes the note, the code stub gets control. It disposes of the string, the NM structure and finally itself, leaving no remains.

It works very well. It requires no disk I/O or temporary files, occupies about 90 bytes of system heap plus the length of your message, and has the advantage of being 100% compatible with current and future systems, without introducing a new concept to the user (e.g. "Go to the Chooser to read startup error messages" -- yuck!).

What follows are the three Think C files StartupError.h, StartupError.c, and StartupError RESP.c. The first two are used within your INIT's project. The third is the code resource which must be compiled separately. If there are any questions that might be of interest to the net, please post them here.

-------- CUT HERE FOR StartupError.h --------

```
/*****************************************************************************
***
*** StartupError.h
***
*** Informative error messages from INITs
***
*** History:
***   jhh 18 jun 90 -- response to news posting
***
***/

#ifndef _H_STARTUP_ERROR
#define _H_STARTUP_ERROR


/*****************************************************************************
**
** Public Functions
**
**/

void
StartupError(int errorNumber);


#endif  /* ifndef _H_STARTUP_ERROR */
```

```
-------- CUT HERE FOR StartupError.c --------


/**-----------------------------------------------------------------------
**
** Include Files
**
**/

#include "StartupError.h"


/**-----------------------------------------------------------------------
**
** Private Macros
**
**/

/**

   T_NMInstall and T_Unimplemented are Mac toolbox trap numbers used to
   test for the existence of the Notification Manager.

**/

#define T_NMInstall          (0xA05E)
#define T_Unimplemented      (0xA89F)

/**

   STARTUP_ERROR_STR_ is the resource ID of the STR# containing the
   error message corresponding to the error number passed to
   StartupError().

   RESPONSE_RESP is the resource ID of the RESP code resource compiled
   separately and stuck in your INIT's resources.

**/

#define STARTUP_ERROR_STR_  (128)
#define RESPONSE_RESP       (128)


/***********************************************************************
***
*** void    StartupError(int errorNumber);
***
*** When your INIT runs into a problem, clean things up, show the X'ed
*** version of your icon and call StartupError() with an error number
*** corresponding to the message you want displayed.
***
*** History:
***   jhh 18 jun 90 -- response to news posting
***
***/

void
StartupError(int errorNumber)
{
   register NMRec  *note;
   register Handle responseCode;
   register long   size;
   register THz    svZone;
   Str255          errorText;
```

```
/**

   Make sure we've got a Notification Manager.

**/

   if (NGetTrapAddress(T_NMInstall, OSTrap) !=
           NGetTrapAddress(T_Unimplemented, ToolTrap)) {

/**

   All of the memory we allocate from here on out is in the System
   Heap.  First create a Notification Manager record and fill it in.
   Note that you can expand this notification method with sounds and
   icons by adding the appropriate code.  See the Notification
   Manager technote #184 for details.

**/

       svZone = TheZone;
       TheZone = SysZone;
       note = (NMRec *)NewPtr(sizeof(NMRec));
       if (!note)
           goto exit;
       note->qType = nmType;
       note->nmMark = 0;
       note->nmSIcon = 0L;
       note->nmSound = (Handle)-1;

/**

   Get the error message corresponding to the error number given.
   For maximum performance, the STR# resource should be tagged
   "Preload" and not "System Heap".  This way, you can be sure
   the messages will be there even if memory space is the cause of
   the error.

   We create a pointer in the System Heap just big enough for the
   string and copy the string into it.  Point the NM record at this
   string.

**/

       GetIndString(errorText, STARTUP_ERROR_STR_, errorNumber);
       size = *(unsigned char *)errorText;
       note->nmStr = (StringPtr)NewPtr(size);
       if (!note->nmStr) {
           DisposPtr(note);
           goto exit;
       }
       BlockMove(errorText, note->nmStr, size);

/**

   The response procedure also must be in a pointer in the System
   Heap.  You need to include the compiled code resource in your
   INIT's resources of type 'RESP'.

   Create a pointer just big enough for it and point the NM record
   at it, also.

**/
```

```
        responseCode = GetResource('RESP', RESPONSE_RESP);
        if (!responseCode) {
            DisposPtr(note->nmStr);
            DisposPtr(note);
            goto exit;
        }
        size = GetHandleSize(responseCode);
        note->nmResp = (ProcPtr)NewPtr(size);
        if (!note->nmResp) {
            DisposPtr(note->nmStr);
            DisposPtr(note);
            goto exit;
        }
        BlockMove(*responseCode, note->nmResp, size);

/**

   Now post the note.  As soon as startup is complete, the NM
   will display the note for the user's edification.  Hurrah.

**/

        NMInstall(note);
exit:
        TheZone = svZone;
   }
}
```

-------- CUT HERE FOR StartupError RESP.c --------

```
/**

   Compile this code in a separate project.  Set the project type to
   Code Resource, type 'RESP' ID 128.  Once compiled, you can build it
   and merge it into your INIT's resource file ("...project.rsrc") or
   build it into a separate file and use ResEdit to copy it in.  This
   code is independent of the INIT, so it can be used as-is for any
   INIT you write.

**/


/**

   ToolScratch is an 8-byte area of low memory used by the Mac toolbox
   and other people as a temporary holding place.

**/

extern long     ToolScratch : 0x09CE;


/************************************************************************
***
*** pascal void    main(QElemPtr nmReqPtr);
***
*** This is the code resource type 'RESP' which you need to compile
*** separately and include among your INIT's resources.
***
*** This code is called when the user closes the Notification Manager's
*** note dialog.  The NM passes to us the address of the NM record which
*** was set up by StartupError().  We use this to clean up and leave.
***
*** This is written is assembly for size.  If you have problems with
*** assembly, I s'pose it could be written in C, but the trick at the
*** end would be hard to duplicate...
***
```

```
*** History:
***   jhh 18 jun 90 -- response to news posting
***
***/

pascal void
main(QElemPtr nmReqPtr)
{
   asm {

/**

   First remove the note from the Notification Manager's notification
   queue.

**/

           move.l      nmReqPtr, A0
           _NMRemove

/**

   Grab the string's address from the NM rec and dispose of it.  Then
   get rid of the NM rec itself.

**/

           move.l      nmReqPtr, A0
           move.l      OFFSET(NMRec,nmStr)(A0), A0
           _DisposPtr
           move.l      nmReqPtr, A0
           _DisposPtr

/**

   Now the tricky part.  This code lives within a small block in the
   System Heap which we want to get rid of.  But it's not safe to
   dispose of the very block you're calling from!  So, we create a
   little bit of code in ToolScratch which does the dispose for us
   and execute it last.

**/

           move.l      4(A7), A1
           move.l      #ToolScratch, A0
           move.l      A0, 4(A7)
           move.l      #0x2040A01F, (A0)   ; movea.l D0, A0 / _DisposPtr
           move.w      #0x4ED1, 4(A0)      ; jmp (A1)
           lea         main, A0
           move.l      A0, D0              ; Pascal clobbers A0 on exit
   }
}
```

-------- END --------

[I apologize for all of the spaces in the posting; I used them instead of tabs because of the funky tabs my terminal emulator gives.]

●●●

From: leipold@eplrx7.uucp (Walt Leipold)

Subject: Re: 24-Hour FKEY?

For anybody who's interested, the following is an FKEY (resource number 6) to toggle the Mac's time display from 12- to 24-hour mode. Install it with ResEdit or FKEY Manager. Have fun...

Disclaimer: I've only tested this FKEY with the American version of the System file.

{ This is a THINK Pascal source file. Compile it as a code resource of type FKEY (resource number [whatever you want], Custom Header option,Purgeable), and link it with Interface.lib & DRVRRuntime.lib }

```
unit FKEY1224;

   interface
       procedure main;

   implementation

       procedure main;
       var
           h: Handle;
           i: Intl0Hndl;
       begin
           h := GetResource('itl0', 0);
           if h = nil then
               SysBeep(10)
           else begin
               i := Intl0Hndl(h);
               if i^^.timeCycle = 0 then begin
                   i^^.timeCycle := 255;
                   ChangedResource(h);
                   UpdateResFile(0);
                   end
               else if i^^.timeCycle = 255 then begin
                   i^^.timeCycle := 0;
                   ChangedResource(h);
                   UpdateResFile(0);
                   end
               else
                   SysBeep(10);
               end;
       end;

end.
```

Walt Leipold

●●●

From: jpab+@andrew.cmu.edu (Josh N. Pritikin)
Subject: Re: VBL tasks Think C 4.0

>I am writing an INIT that patches the JCrsrTask routine with my
>own routine. I am confused about Think's SetUpA4 macro and the
>IM SetUpA5 macro. They both seem to give me access to the globals

In general, SetUpA5 is used in applications and SetUpA4 is used in code resources. Check the manual for specifics. Since you writing an INIT, your globals will be accessed through A4. Read the part in the manual about INITs now, then read the rest of this. I will describe one of many ways to do what you want to do.

On entry, at bootup, D0 contains a pointer to the INIT resource. You should called RememberD0 and SetUpA4 (in addition to movem all the registers your going to change onto the stack). This set of calls will store the ptr in a local global variable (read, "not A4 referenced but it is static") and load A4 with it. If you want your INIT to stick around, you should RecoverHandle on A4, then DetachResource, HLock and HNoPurge (to be safe).

When your INIT is called again from JCrsrTask or the like, you need to SetUpA4, but DON'T RememberD0. The local global variable already has the correct value (the handle is locked down). You can exit normally using RestoreA4, etc. To be safe, you shouldn't change any registers as a side effect of your routine unless you know what your doing.

Always make sure the stack pointer doesn't get screwed up and happy debugging...

/* Josh Pritikin

●●●