patch

COLLABORATORS			
	TITLE : patch		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 18, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

patch

1.1 patch.doc

```
--overview--

disable()

enable()

end()

install()

patched_function()

remove()
```

1.2 patch.m/--overview--

```
                  PURPOSE
To allow easy patching of Amiga library and device functions
with Amiga E code.

 OVERVIEW
The AmigaOS has a facility to change the code called in any of the
functions of a library or device. But due to the way Amiga E
works, it has been difficult to easily use E code as patches.

This object allows you to install patches to the system using
Amiga E code in the most flexible way, via an assembler wedge.

Installation:

You first need to open the library/device that you are going to
patch. You then need to know the Library Vector Offset of the
entry you are going to patch. This is available using the 'lvo'
tool from the Developer Kit, or any assembler include file with
```

'LVO' in the title. You must keep the library/device open for the entire duration of the patch. If you do not, the library may be flushed from memory and loaded somewhere else, indirectly removing the patch and also causing a crash when we try to remove our defunct patch later.

You create a patch object with
install()
, to install a wedge which
calls an E function which you defined like so:

```
PROC patch_code(function, a7,a6,a5,a4,a3,a2,a1,a0,
                          d7,d6,d5,d4,d3,d2,d1,d0)
```

You can give alternate names to the 'register' parameters, but do remember that they will always be passed IN THE ABOVE-STATED ORDER, and writing the parameter's names in a different order will NOT cause the registers to be passed to you in a different order.

See
patched_function()
for more about this.

Your patch in operation:

This E function is called 'as' the function you have patched, by absolutely any task and program that could call the original function. Also, the function may be called by more than one program at once, so use a Semaphore or similar protection when using global variables.

You are given all the registers as set when the patch was called - do NOT modify them unless it is documented that the function does this. That said, you can always modify the 'd0' variable as it is ignored on exit, the return value of your function is returned in D0 rather than the stack-based copy.

Occasionally you can modify the 'scratch register' variables d0, d1, a0 and a1, but you must check the documentation of the function you are patching, to ensure it does not promise to preserve any of these registers.

The result of your function call is always returned in D0, but E's multiple return values (in D1 and D2) are ignored, and restored to their original values.

Calling the original function:

You are also given a pointer to the original function you have patched. In most patches, you will have to call the original function sometime, so you would set up the registers appropriately from the parameters, including A6 as the library/device base, and call the original function. Do NOT do this by writing the E construct "function()", as this will use an unspecified A-register to make the call. Instead, do it yourself with another register that you choose. If you need to pass A0 to A3 as parameters, then you will have to preserve and use A4 or A5.

```
Patch removal:

Your  patch  can be either enabled or disabled. When disabled, the
assembler  wedge  simply  hops  to  the  original function, adding
nothing  to  the  stack. Simply exiting your program at this point
without  ENDing  the  object would free the object instance itself
but  would leave the working wedge in place, consuming 70 bytes of
memory and adding 3 instructions to the function.

Some  people  would  advocate  exiting your program with the wedge
still in place, as the user is unlikely to do that often, but when
they do there will be no problem with exiting correctly.

Others  would always recommend total removal of the patch, even if
that  means  waiting. Unlike other wedges, the removal method used
by  this  object is very safe, only removes the patch if it is not
being  used,  and understands programs like SetFunction Manager or
SaferPatches which allow removal of patches in any order.

My  advice  is to always
              disable()
               the patch, then try to
              remove()
                the  patch. If that fails, ask the user if you should keep  ↩
                   trying,
or just exit.

 WARNING
When your patch is enabled, the combination of assembler wedge and
E code will add a MINIMUM of 80 bytes - THEN extra bytes are added
for the variables in your patch! There is NO stack checking made!

You should avoid local variables at all costs, use the full REG=5
optimisation, and avoid defining STRINGs or LISTs as variables.

More warnings will be pasted in here when I can find the message
about patches that I posted to the E list.
```

1.3 patch.m/disable

```
                 NAME
patch.disable() -- prevent further execution of the patch.

 SYNOPSIS
disable()

 FUNCTION
Stops the patch from being invoked again. All calls to the patched
function  will be passed directly to the original function, not to
your patch.

There  may,  however, still be invocations of the patch running at
the time this call returns.
```

SEE ALSO

enable()

1.4 patch.m/enable

NAME
patch.enable() -- allow execution of the patch.

SYNOPSIS
enable()

FUNCTION
Toggles a switch in the assembler wedge which stops it passing all calls of the patched function to the original function, and starts passing them to your patch.

Your patch should be ready to run at any time from the start of the call to this method.

SEE ALSO

disable()

1.5 patch.m/end

NAME
patch.end() -- Destructor.

SYNOPSIS
end()

FUNCTION
Frees resources used by an instance of the patch class. It will first
disable()
the patch, then it will busy loop until the patch is successfully removed.

SEE ALSO

disable()
,
remove()
,
install()

1.6 patch.m/install

```
                 NAME
patch.install() -- Constructor.

 SYNOPSIS
install(base, offset, patchfunc)
install(base, offset, patchfunc, userdata)

 FUNCTION
Initialises  an  instance of the patch class, and installs a patch
in the system. The patch will not be enabled to begin with, so you
must  call
              enable()
                on  the  patch  for it to start working. The
exception "MEM" will be raised if there is no memory for a wedge.

Read
              patched_function()
               to see how this patch is called.

 INPUTS
base      - the  base  of  the library or device of whose function
            you  will  be  patching. It should remain open for the
            entire  life of the patch, otherwise it may be flushed
            from memory, rendering the patch useless.

offset    - This  is the Library Vector Offset of the function you
            are patching. The appropriate number is available from
            LVO files or the 'lvo' tool.

patchfunc - The address of the Amiga E function which will replace
            the specified device/library function.

userdata  - An optional parameter that can be anything you want it
            to be. It will be passed to your E function as a LONG,
            whether or not you define or use it. The default value
            for this parameter is zero.

 SEE ALSO

              end()
              ,
              enable()
              ,
              patched_function()
```

1.7 patch.m/patched_function

```
 NAME
patched_function() -- how your installed patch is called.

 SYNOPSIS
```

```
result := patched_function(
            original_function,
            a7,a6,a5,a4,a3,a2,a1,a0,d7,d6,d5,d4,d3,d2,d1,d0
          )

result := patched_function(
            userdata, original_function,
            a7,a6,a5,a4,a3,a2,a1,a0,d7,d6,d5,d4,d3,d2,d1,d0
          )
```

FUNCTION

When you install a patch, your patch function will be called instead of the original function. Rather than your function code being installed directly as the new patch, an assembler wedge is instead used. This allows for the patch to be enabled and disabled with ease, and also prepares the correct environment for an Amiga E function to operate.

The wedge prepares a set of arguments on the stack for the patch function, so that it can know everything neccessary to implement the patch.

INPUTS

userdata - this value was chosen by the programmer when installing the patch.

This userdata value is passed whether or not it was requested in the installation, and the way Amiga E defines function parameters allows you to either define your function so that it knows the userdata value, or it doesn't (as shown above).

The userdata value can enable you to write only one function to patch multiple library functions - each patch would have a different userdata parameter, and the patch code would use this to recognise which function it was patching when called.

original_function - this is a pointer to the code that would normally be executed, if your patch was not installed. In most patches, you do not replace the entire functionality of the patch, and therefore have to use this pointer to call the original function. Remember to initialise all required parameters (including A6) before calling this function.

a7, a6, a5, a4, a3, a2, a1, a0, d7, d6, d5, d4, d3, d2, d1, d0 - these are the 68000 registers as set before your patch was called. Some of these will be parameters to your patch, others will be of no use, but you can NOT remove them from your function declaration. You may assign names other than their real register names to them, perhaps to mirror the declaration of the patched function - but be warned that you must

NOT modify these variables, as they will be restored to the registers on exit of your function. The only parameter you can safely alter is the 'd0' parameter, as its contents are ignored on your function exit, instead the function's return value is used for D0.

These parameters are always defined in the same order - from A7 to A0, then D7 to D0.

RESULT

The value you return from your patch function is always returned in D0 by normal E standards. The assembler wedge avoids restoring the original value of D0 it picked up on entry, so this result stays. Some functions declare that they return results in registers other than D0, so you will have to store those other results in the appropriate 'register' parameters that were passed in to your function. On return, those variables will be restored into the appropriate 68000 registers.

1.8 patch.m/remove

NAME

patch.remove() -- attempt to remove the patch.

SYNOPSIS

```
removed := remove()
```

FUNCTION

Attempts to remove the patch from the system, and restore the original function. If the patch is successfully removed, there is nothing you can do but END the patch.

RESULT

removed - TRUE if removal of the patch was successful.

NOTE

You may have a greater chance of successful removal of the patch if you

```
disable()
```

it first.

SEE ALSO

```
end()
,
disable()
```