Subject: widget_control and group_leader
Posted by nrk5 on Fri, 22 Dec 2000 22:06:48 GMT
View Forum Message <> Reply to Message

Lets say I have two widgets, A and B. There are two links between the two:
1) A.top and B.top are eachother's groupleaders, and
2) A uses common blocks and has a variable 'foreign_event_handler' that is set by B.

So, when an event is generated by A and the 'Use Foreign Event Handler' option is set in the widget, events generated by A go to whatever B set 'foreign_event_handler' using:

```
    widget_control, id, event_pro=foreign_event_handler
```

Things to note:
1) A can't be modified at all. Nothing added or changed. (ie. no more variables)
2) B is an object widget and needs to set its structure variables to variables in the events generated by A.
3) In B::init, B.top has a uvalue of self.

The question is, how can I use foreign_event_handler to get to 'B self' from an event generated by A? My thought was:

```
PRO foreign_event_handler, eventFromA
  widget_control, eventFromA.top, get_Group_Leader = BtopID
  widget_control, BtopID, get_Uvalue = objectReferenceToB
  ...
END
```

And now I would be in business. But, is there such as thing as get_group_leader? Is there another way to do this?

I know that not being able to change A doesn't help, else there would be a million solutions, but its not my program. The only minor change I might be able to make is to create a generic variable in A's common block that could be set to whatever, but then I would have to define it as a string or a long, and that would restrict its use.

Thanks much,

-- Nidhi

Nidhi Kalra
nrk5@cornell.edu

Sent via Deja.com
http://www.deja.com/

Subject: Re: widget_control and group_leader
Posted by nrk5 on Sun, 24 Dec 2000 07:44:51 GMT
View Forum Message <> Reply to Message

Oh jeez. I dont know why this code came out so messy. Maybe this will
look a little better.

```
widget_control, top_menu, event_pro = 'topmenu_event'

widget_control, state.draw_widget_id, event_pro = 'draw_color_event'

widget_control, state.draw_base_id, event_pro = 'draw_base_event'

widget_control, state.keyboard_text_id, event_pro = 'keyboard_event'

widget_control, state.pan_widget_id, event_pro = 'pan_event'
```

> --
> -----------------
> Nidhi Kalra
> nrk5@cornell.edu
>
> Sent via Deja.com
> http://www.deja.com/
>

--

Nidhi Kalra
nrk5@cornell.edu

Sent via Deja.com
http://www.deja.com/

Subject: Re: widget_control and group_leader
Posted by John-David T. Smith on Sun, 24 Dec 2000 19:48:08 GMT
View Forum Message <> Reply to Message

Nidhi Kalra wrote:

```
>  Let me paste in a bit of the code from program A. In the event handler
>  that I am mostly concerned with, the user sets the mode. mousemode
>  cases 0-3 were already there, so I added 4 for uniformity. When 4 is
>  selected, events on the draw_widget are sent to the foreign event
>  handler.
>
>  pro a_event, event
>  ; Main event loop for atv top-level base, and for all the buttons.
>  ...
>  widget_control, event.id, get_uvalue = uvalue
>  ...
>  case uvalue of
>     'mode': case event.index of
>         0: widget_control, state.draw_widget_id, $
>                        event_pro = 'atv_draw_color_event'
>         1: widget_control, state.draw_widget_id, $
>                        event_pro = 'atv_draw_zoom_event'
>         2: widget_control, state.draw_widget_id, $
>                        event_pro = 'atv_draw_blink_event'
>         3: widget_control, state.draw_widget_id, $
>                        event_pro = 'atv_draw_phot_event'
>         4: widget_control, state.draw_widget_id, $
>                        event_pro = state.foreign_event_handler $
>                        + '_event'
>         else: print, 'Unknown mouse mode!'
>     endcase
```

This design choice of A (and A's author probably is scratching his head about now), is simply one possiblity among the zillions of ways events flow can be managed. He has decided to redirect events to different procedures, based on which "button" or mode is selected. Naturally, you added yet another mode to this. I guess I haven't looked closely enough at A to speak sensibly about its design choice. But the one point I wanted to make was that there is no need for events to be processed only in 1 place. That is, you could easily have zoom, blink, phot, or color active at the same time B is receiving these event! Wouldn't this be better functionality, unless something B is doing is really making A's default behavior non-intuitive.

>
> The functionality I'm going for is that the user can decide when to use
> external event handlers and when to let program A run 'naturally'. At
> the moment, I have tried to keep foreign_event as general as possible.
> Each B can do whatever it pleases with its own particular foreign_event
> handler. The two things (now) registered with A are the foregn event
> handler to use and a widget_ID to use. Whatever that needs to be.
>
>> What I would recommend in this case is set up a foreign event handler
>> *method*, since the foreign widget is an object. That is, have a

>> routine to sign up for events from A. from within B., like this:
>>
>> a_signup, self, "Handle_A_Events", /Button, /TRACKING
>>
>> or some such. Then, each "foreign" object can sign up for whatever
>> events it wants.
>

> How do I register event/object pairs? Ok. So here I'm a little lost
> (Caution: Newbie IDL-er at work). A
> registers the following event handlers:
>
> widget_control, top_menu, event_pro = 'topmenu_event'
> widget_control, state.draw_widget_id, event_pro = 'draw_color_event'
> widget_control, state.draw_base_id, event_pro = 'draw_base_event'
> widget_control, state.keyboard_text_id, event_pro = 'keyboard_event'
> widget_control, state.pan_widget_id, event_pro = 'pan_event'
>
> And everything in these main bases is differentiated by uvalues (as you
> can see from the above code). So I'm a bit confused about how to go
> about differentiating the "events requested" and how the reigstering
> in "call_method,method,obj, ev" works.

Sorry if I didn't give enough detail. You register it however you want! You don't need to use IDL's builtin event_pro stuff, and in many cases, it's more convenient not to. To make things concrete, consider that you might like A to remain unmodified. You'd make an event handler app "C" (I'd do it as an object):

```
pro Broker::signup, obj, method, REMOVE=rm
  if obj_valid(obj) eq 0 then return
  if ptr_valid(self.signup) then begin
     wh=where((*self.signup).object eq obj,cnt,COMPLEMENT=valid)

     ;; Rid list of obj, if it's already on there
     if cnt ne 0 then begin
        if valid[0] eq -1 then ptr_free, self.signup $
        else *self.signup=(*self.signup)[valid]
     endif
  endif

  ;; Add it to the list, if necessary
  if keyword_set(rm) then return

  list_item={BROKER_SIGNUP,Object:obj,Method:Method}
  if ptr_valid(self.signup) then begin ; append item
     *self.signup=[*self.signup, list_item]
  endif else $                  ;create list with item
```

```
    self.signup=ptr_new(list_item,/NO_COPY)
end

pro broker_handler, ev
  widget_control,ev.top, get_uvalue=self
  self->Handler
end

pro Broker::Handler, ev
  ;; Send it to A
  widget_control, self.A_ID, SEND_EVENT=ev

  ;; Send it to all the B's
  if ptr_valid(self.signup) eq 0 then return

  for i=0,n_elements(*self.signup)-1 do begin
    call_method, (*self.signup)[i].Method, $
                 (*self.signup)[i].Object, ev
  endfor
end

pro Broker::Cleanup
  ptr_free,self.signup
end

pro Broker::Init, A_ID
  ;; We will pre-process A's events
  widget_control,A_ID, event_pro= 'c_handler'
end

pro Broker__Define
  struct={Broker,A_ID: A_ID, signup:ptr_new(), ...}

  ;; A convenience struct for the signup list
  list_struct={BROKER_SIGNUP, object:obj_new(), method:''}
end
```

This is just an outline, and I haven't tried it. But it gives you an idea of what I had in mind. You can see how I caught A's events before it does, and then send them on to A (via the standard IDL event flow), and also to the B's (via the method/object they signed up for). You could obviously add more intelligence to this dispatch process (e.g. only button events to B1, etc.)

This uses IDL's widget hierarchy some, and some of it's own design too (e.g. the B methods). This is no problem. An event is simply a structure, no different from any other type of data, and you can use it however you want.

> So, ideally, here's the functionality im looking for. On "foreign"
> mode, all events go to foreign_event_handler. If foreign event handler
> wants to do something with it, wonderful. If not, the event goes back
> to where it would go on non-foreign mode.

So, the one thing I didn't specify is when the B's signup for events. I.e. how do they know they are on? Two possibilities:

1. They are always on, i.e. you start them from the command line, and they immediately sign up:

```
a=A_widget(lots_of_args)
c=obj_new('Broker',a)
b=obj_new('Cool_foreign_helper',BROKER=c)
```

and in b's Init:

```
function Cool_foreign_helper::Init, BROKER=brk
  brk->Signup, self, 'MyHandler'
  return, 1
end
```

2. They get turned on by A (which means you'd have to modify A to at least have this ability).
>
> The quick and dirty way is to put in a simple statement in each of the
> four event handlers:
>
> if (foreign) send_event, foreign_event_handler, event (or whatever).
>
> hmm...waitaminit. what if i register foreign_event_handler as the event
> handler for the top level base? what would that do? Would all events
> then go to foreign_event_handler and then bubble up/down?

These kind of uncertainties about just how IDL will handle dynamically re-routed events are just the thing that motivates moving beyond the standard event flow paradigm. What I gave is only a sketch, but once you take events "into your own hands", you can accomplish all sorts of things with them.

Remember, events are data too, just like 4 or "a string". They can be used to control widgets, and all you have to have are the widget_id's to make this work. Just because RSI publishes a manual describing standard event processing doesn't mean you can't innovate beyond that (especially in unusual cases like yours).

Good luck,

JD

Subject: Re: widget_control and group_leader
Posted by davidf on Sun, 24 Dec 2000 21:22:26 GMT
View Forum Message <> Reply to Message

John-David Smith (jdsmith@astro.cornell.edu) writes:

> Just because RSI publishes a manual describing standard
> event processing doesn't mean you can't innovate beyond
> that (especially in unusual cases like yours).

As someone who has tried a great many oddball ways
of processing events, I can attest to the wisdom
of JD's words here.

But I also note that as I've gained more experience
with widgets and events, that my event processing has
become more conservative (as opposed to, say, my
politics). And now I often find myself a proponent of
the Occum's Razor school of event handling, in which the
simpler is the better. :-)

Cheers,

David

P.S. Let's just say I have enough to worry about
keeping track of those darn object methods to
worry overly much about events flying around everywhere.

--
David Fanning, Ph.D.
Fanning Software Consulting
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155

Subject: Re: widget_control and group_leader
Posted by nrk5 on Mon, 25 Dec 2000 05:18:23 GMT
View Forum Message <> Reply to Message

In article <3A4652F8.D47410F8@astro.cornell.edu>,
John-David Smith <jdsmith@astro.cornell.edu> wrote:

Thanks JD. Lots of good info here. Having looked at the code, I realize
that I'm unlikely to have more than one object widget behaving in the B
position. So, rather than having the list revolve around objects, I
thought it might be cute to have it use the event.id as the key.

What I mean is that the first thing when you create the Broker is that
it does a

```
    widget_control, id, event_pro = broker_event
```

to all the widgets. Then, B registers with the broker items in the
signup_list of the following type:

```
  item: Event_ID     ;The ID to match
        Object B     ;The object owning the method
        Method       ;The method to be called
        Call_Before  ;A boolean value indicating when to call the method
```

So, the event handler gets changed to:

1. Find all items in the list such that item.Event_ID = event.id
2. Of these, find those where Call_Before = 1
3. Call each of these methods
4. Send the object back to where it came from (widget control, send..)
5. Find the remaining items where Call_Before = 0
6. Call each of these methods

The principle is the same, the details are a bit different. I also have
some technical questions about your code. Things I couldn't find in the
help.

```
>
>  pro Broker::signup, obj, method, REMOVE=rm
>    if obj_valid(obj) eq 0 then return
>    if ptr_valid(self.signup) then begin
```

--> I couldn't find the keyword COMPLEMENT documented in the call to
'where.' It appears to return those items that are not in 'wh'.

```
>      wh=where((*self.signup).object eq obj,cnt,COMPLEMENT=valid)
>
>      ;; Rid list of obj, if it's already on there
>      if cnt ne 0 then begin
```

-->Assuming I'm right about what 'valid' is, does valid[0] = -1 if there
are no items in the list that arent in 'wh'?

```
>         if valid[0] eq -1 then ptr_free, self.signup $
```

-->I am not sure what (*self.signup)[valid] does. Reissue self.signup to
be valid? [valid] ?

```
>         else *self.signup=(*self.signup)[valid]
>       endif
>    endif
>
>    ;; Add it to the list, if necessary
>    if keyword_set(rm) then return
>
```

-->Why does list_item have 'BROKER_SIGNUP'? What does that do/why is it
there?

```
>    list_item={BROKER_SIGNUP,Object:obj,Method:Method}
>    if ptr_valid(self.signup) then begin ; append item
>       *self.signup=[*self.signup, list_item]
>    endif else $                  ;create list with item
>       self.signup=ptr_new(list_item,/NO_COPY)
>  end
>
>  pro broker_handler, ev
>    widget_control,ev.top, get_uvalue=self
>    self->Handler
>  end
>
>  pro Broker::Handler, ev
>    ;; Send it to A
>    widget_control, self.A_ID, SEND_EVENT=ev
>
>    ;; Send it to all the B's
>    if ptr_valid(self.signup) eq 0 then return
>
>    for i=0,n_elements(*self.signup)-1 do begin
>       call_method, (*self.signup)[i].Method, $
>                    (*self.signup)[i].Object, ev
>    endfor
>  end
>
>  pro Broker::Cleanup
>    ptr_free,self.signup
>  end
>
>  pro Broker::Init, A_ID
>    ;; We will pre-process A's events
>    widget_control,A_ID, event_pro= 'c_handler'
>  end
>
>  pro Broker__Define
>    struct={Broker,A_ID: A_ID, signup:ptr_new(), ...}
>
>    ;; A convenience struct for the signup list
```

-->Again, you use the term 'Broker_Signup'. What is that?

```
>    list_struct={BROKER_SIGNUP, object:obj_new(), method:''}
>  end
>
```

> This is just an outline, and I haven't tried it. But it gives you an
idea of
> what I had in mind. You can see how I caught A's events before it
does, and
> then send them on to A (via the standard IDL event flow), and also to
the B's
> (via the method/object they signed up for). You could obviously add
more
> intelligence to this dispatch process (e.g. only button events to B1,
etc.)
>

>
> So, the one thing I didn't specify is when the B's signup for events.
 I.e. how
> do they know they are on? Two possibilities:
>
> 1. They are always on, i.e. you start them from the command line, and
they
> immediately sign up:
>

```
>  a=A_widget(lots_of_args)
>  c=obj_new('Broker',a)
>  b=obj_new('Cool_foreign_helper',BROKER=c)
```

This seems like a good choice. I really dont want to mess with A.

```
>
>  and in b's Init:
>
>  function Cool_foreign_helper::Init, BROKER=brk
>     brk->Signup, self, 'MyHandler'
>     return, 1
>  end
>
```

> 2. They get turned on by A (which means you'd have to modify A to at least have
> this ability).

> Just because RSI publishes a manual describing standard event
> processing doesn't mean you can't innovate beyond that (especially in
> unusual cases like yours).

True. Sometimes you just cant follow the herd. Moooo.

Thanks much :)

Nidhi

--

Nidhi Kalra
nrk5@cornell.edu

Sent via Deja.com
http://www.deja.com/

Subject: Re: widget_control and group_leader
Posted by John-David T. Smith on Fri, 29 Dec 2000 18:33:38 GMT
View Forum Message <> Reply to Message

Nidhi Kalra wrote:
>
> In article <3A4652F8.D47410F8@astro.cornell.edu>,
>   John-David Smith <jdsmith@astro.cornell.edu> wrote:
>
> Thanks JD. Lots of good info here. Having looked at the code, I realize
> that I'm unlikely to have more than one object widget behaving in the B
> position. So, rather than having the list revolve around objects, I
> thought it might be cute to have it use the event.id as the key.
>
> What I mean is that the first thing when you create the Broker is that
> it does a
>
>   widget_control, id, event_pro = broker_event
>
> to all the widgets. Then, B registers with the broker items in the
> signup_list of the following type:
>

```
>    item: Event_ID     ;The ID to match
>          Object B     ;The object owning the method
>          Method       ;The method to be called
>          Call_Before  ;A boolean value indicating when to call the method
>
>  So, the event handler gets changed to:
>    1. Find all items in the list such that item.Event_ID = event.id
>    2. Of these, find those where Call_Before = 1
>    3. Call each of these methods
>    4. Send the object back to where it came from (widget control, send..)
>    5. Find the remaining items where Call_Before = 0
>    6. Call each of these methods
```

I presume in 4 you mean send the *event* back to where it came from? This might be more work than you need... see below.

```
>
>  The principle is the same, the details are a bit different. I also have
>  some technical questions about your code. Things I couldn't find in the
>  help.
>
>>
>>  pro Broker::signup, obj, method, REMOVE=rm
>>    if obj_valid(obj) eq 0 then return
>>    if ptr_valid(self.signup) then begin
>
>  --> I couldn't find the keyword COMPLEMENT documented in the call to
>  'where.' It appears to return those items that are not in 'wh'.
```

Sorry, that's a 5.4ism, and your guess is correct. In older days I would have used:

```
wh=where((*self.signup).object ne obj,cnt)

if cnt lt n_elements(*self.signup) then ...
   ....
   (*self.signup)=(*self.signup)[wh]
```

before adding the new one on.

```
>
>>      wh=where((*self.signup).object eq obj,cnt,COMPLEMENT=valid)
>>
>>      ;; Rid list of obj, if it's already on there
>>      if cnt ne 0 then begin
>
>  -->Assuming I'm right about what 'valid' is, does valid[0] = -1 if there
>  are no items in the list that arent in 'wh'?
```

You are correct. The equivalent test is cnt eq 0 vs. cnt eq
n_elements(*self.signup). I.e. none of them, or all of them. If none remain
valid, we free the list.

>
>>   if valid[0] eq -1 then ptr_free, self.signup $
>
> -->I am not sure what (*self.signup)[valid] does. Reissue self.signup to
> be valid? [valid] ?

It's just an array indexing of the dereferenced list pointed to by self.signup.
I.e. self.signup is a pointer to a 1-d array of {BROKER_SIGNUP} structs.
*self.signup is that list. (*self.signup)[0] is the first element of that
list. Etc.

Precedence rules with pointer dereference in IDL are goofey (or rather, C rules
are goofey but most people are used to them).

> -->Why does list_item have 'BROKER_SIGNUP'? What does that do/why is it
> there?

That is a named structure, which is defined in the class definition, along with
the class itself. It doesn't actually serve to define the class in any way, but
is an auxiliary helper structure. The reason to use named structures is the
ability to concatenate them together (vs. anonymous structures with the same
fields/data sizes). Hence managing the list is easier.

>> Just because RSI publishes a manual describing standard event
>> processing doesn't mean you can't innovate beyond that (especially in
>> unusual cases like yours).
>
> True. Sometimes you just cant follow the herd. Moooo.

One more piece of advice. If you're sure there's only one B, you don't need a
list at all! In fact, you don't even need a Broker! You just make B itself
intercept the events, like (extra bits ommitted):

```
pro B_Event, ev
  widget_control, ev, get_uvalue=self
  self->Event, ev
end

pro B::Event, ev
  ;; Process the event as it relates to B

  ;; Also Send back to A
  widget_control, self.A_ID, EVENT_PRO=self.A_EVENT,SEND_EVENT=ev
  widget_control, self.A_ID, EVENT_PRO='B_Event'
  ;; or use:
  ; call_procedure, self.A_EVENT, ev
end

function B::Init, A_ID
  self.A_ID=A_ID
  self.A_EVENT=widget_info(A_ID,/EVENT_PRO)
  widget_control,self.A_ID, SET_UVALUE=self, EVENT_PRO='B_Event'
  return, 1
end

pro B__Define
  struct={B, A_ID:0L, A_EVENT:''}
end
```

You get the idea. Much simpler. Less flexible, but easier to code. Simply intercept the event. Process it locally for B's own devious purposes, and forward it on to A. By the way... there was an error in my prior logic. Sending an event to A via SEND_EVENT will bring it right back to the Broker, making an endless event loop, unless you temporarily reset the event handler. That's shown here. Also you really needn't use SEND_EVENT, which normally would use the swallow vs. non-swallow event tree paradigm. In this case, since all event_pro's swallow events (no return value), it's exactly equivalent (and possibly faster?) just to say:

call_procedure, self.A_EVENT, ev

to send them back to A. How's that for simple, David?

JD

Subject: Re: widget_control and group_leader
Posted by John-David T. Smith on Fri, 29 Dec 2000 18:34:10 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>> Just because RSI publishes a manual describing standard
>> event processing doesn't mean you can't innovate beyond
>> that (especially in unusual cases like yours).
>
> As someone who has tried a great many oddball ways
> of processing events, I can attest to the wisdom
> of JD's words here.
>
> But I also note that as I've gained more experience

> with widgets and events, that my event processing has
> become more conservative (as opposed to, say, my
> politics). And now I often find myself a proponent of
> the Occum's Razor school of event handling, in which the
> simpler is the better. :-)

Indeed. All who feel that RSI consistently implements the simpler solution, say "aye"....

  cough.

jd
