Cursor Tracking In SAS/AF FRAME Applications

Don Stanley
Don Stanley Consulting Limited
Wellington
NEW ZEALAND

Overview

Cursor tracking is simply the ability to be able to detect the position of the cursor and automatically, i.e. without any user input, carry out processing specific to the current cursor position.

Perhaps the simplest, and best known usage of cursor tracking in the PC environments is for displaying tool tips. These are small snippets of information that display when the cursor moves over some part of the screen. They tell us briefly and succinctly what will happen when the user presses a mouse button (usually the left mouse button) whilst the pointer is positioned at its current point.

Cursor tracking is used extensively in the Windows 95 Operating Environment. All the automatic popping up of menu options is done by detecting the cursor position and carrying out processing based on that position.

Tool tips are arguably the most 'obvious' cursor tracking application. Tool tips are built into the SAS environment toolbars that display at the top of the screen. They can also be used in the applications we build with SAS/AF FRAME entries.

In this paper I will be demonstrating the use of cursor tracking in FRAME applications. I will show how tool tips work in both the development and build environments. Two types of tool tips will be presented, those that display over the application display in a small rectangular box, and those that display in the status area. The first is built into SAS/AF FRAME region attributes and works in both the development and runtime environments, the second requires minimal code to set up and runs in the runtime environment only.

After discussing tool tips I will move onto to one area where FRAME has in built cursor tracking. This is where you can modify the cursor shape. This has many applications, for example, suppose an application allows some object to be zoomed larger. Changing the cursor to a magnifying glass is very useful as an intuitive on-screen aid in this situation. Also demonstrated is a technique for placing SAS in a loop while it awaits some event. I will show how to change the cursor shape so the user does not realize that SAS is actually still processing. This example demonstrates how to make SAS wait a given time period, but also allow the user to select other widgets.

Error handling can benefit by use of cursor tracking. I will demonstrate a system that pops up an error message box whenever a user moves the cursor over a widget that is in error. This provides a simple error handling system, and gets around the problem of multiple widgets being in error but only one message being available, since the message changes as you move the cursor onto a different widget.

Often screen real estate is a limiting factor in our applications. This can be a particular problem if we need to reserve space for images, or reserve space for buttons that display images upon selection. I will use cursor tracking to show how to display an image when you move over some critical field, for example, moving the pointer over a person's name might display their photograph. The image is removed immediately upon leaving the field.

The intention of this paper is to give you ideas how to use cursor tracking. Some undoubtedly famous (or forgotten in history) individual whose name has never been known to me stated something along the lines of **'.. you should pass on knowledge, no matter how trivial. Your knowledge, which you may believe of little practical use, may be the missing piece of knowledge that some other person needs to create something of immense value ..'**. If you have glossed over cursor tracking, don't understand cursor tracking, never heard of cursor tracking, I hope this paper adds to your FRAME knowledge in such a way that you can increase the usefulness of your FRAME applications by use of the ideas presented.

Tool Tips

With the advent of images that we click on to initiate some actioncame a problem. In text based systems we could usually tell the user, via some form of prompt, just what a particular option would do. However the small images used in software development meant that prompts are usually not an on-screen attribute of the image.

To get around this problem, and permit both a small image (thus maximizing screen real estate) and some text (thus describing what the image will cause to happen upon clicking it), the concept of the tooltip was developed.

Simply put, a tooltip is nothing more than a short piece of text describing briefly what will happen when a particular object is selected.

To be successful, a tooltip has to be able to display when the mouse pointer is situated on the visual object (widget) that the text applies to. Thus it is a prompter for the user, not a conscious selection of a help option. The general presumption behind tooltips is that when a user sits a mouse on a widget and does not click the mouse button, they need some assistance.

A simple example of tooltips in the SAS environment occurs when the toolbar is displayed at the top of the screen. Move the mouse pointer to an image on the tool bar and a small text description appears. In other applications, Microsoft have embedded the tooltip concept throughout their software, whilst WordPerfect (I haven't yet seen the Corel version though) displayed tooltips in the message area rather than near the widget which the cursor is currently over.

Tooltips are built into the SAS application workspace (AWS) toolbox environment. They are also built into the SAS/AF FRAME entry environment, although somewhat more restrictively. Not all objects display tooltips, and not all platforms display them. However, on a platform that does support tooltips on a widget use the following technique to develop the tip:

1. use right mouse button to bring up the objects popup and select REGION ATTRIBUTES
2. select the OUTLINE button
3. select the BUTTON option as the outline type
4. select PUSH BUTTON as the button behavior
5. enter a description, preceded by the text '\n', for example

   \nThis is my tooltip

When you go back to the development environment, the tooltip displays as you move the cursor over the widget, and this also occurs in the runtime environment.

You can also code tooltips. I decided to demonstrate coding tooltips by placing the tooltip in the message area. This is very simple, and is shown in appendix one. Note that the FRAME documentation contains an example showing how you could implement tooltips in code also.

Having now introduced tooltips, discussion of the physical mechanism for creating code to run when the mouse pointer moves over a widget is useful.

Cursor Tracking

Cursor tracking is easiest described as the ability to write code that detects the mouse pointer and take action dependent on where the mouse pointer is placed.

The actual detection of the mouse pointer position is provided internal to FRAME. We do not need to concern ourselves with it. What we need to concern ourselves with is what to do when the mouse pointer is placed over a widget, or over the FRAME itself.

The _CURSOR_TRACKING_ON_ method switches on cursor tracking for a widget or the FRAME. When you switch on cursor tracking for a widget, FRAME automatically tracks the mouse movement.

We define code using the _CURSOR_TRACKER_ method. This method is written for a widget, or for the FRAME. When we execute the _CURSOR_TRACKING_ON_ method (either at the widget or FRAME level), and we have a _CURSOR_TRACKER_ method override in existence, the mere movement of the mouse pointer over the widget or in the FRAME causes the appropriate code (widget or FRAME) to execute.

What we have here is a mechanism for running code without the user needing to do anything more than move the mouse. They do not even need to click a mouse button.

The ramifications of this are enormous. I introduced the cursor tracking topic above using tooltips. However, some of the other things I will demonstrate here include

- simple error handling systems
- automatic image display
- time driven event handling

Per-Instance Methods

I quickly found that widget class overrides become unwieldy when creating comprehensive cursor tracking. This is because you have to create many sub-classes just for cursor handling.

Per-instance methods allow a method override without subclassing. You use the object class _SET_INSTANCE_METHOD_ method to define a per-instance method at runtime. The override implied by the per-instance exists only at runtime.

The concept is very useful, and becomes even more so when you use the BEFORE and AFTER parameters of the _SET_INSTANCE_METHOD_ method. These allow multiple method overrides for the same method, with an execution hierarchy being established.

The BEFORE parameter causes the current override to execute BEFORE any other overrides for the same method that have already been defined. The AFTER parameter forces the current override to execute AFTER any existing overrides.

In general, if more than one override is in existence, only one should call the super method. This is the override method, and usually has no BEFORE or AFTER parameter. The BEFORE and AFTER instance methods do not execute the super method.

Changing the cursor shape & Time Driven Event Handling

You can change the shape of the cursor as it crosses any widget, or when it is placed in the FRAME but not over a widget.

You use the _SET_CURSOR_SHAPE_ method. This is an example of a simple cursor tracking, built into FRAME.

Why would you want to do this? There are many answers, for example, you may have an icon that causes a report to be displayed on the screen. The icon may look like a report in miniature, so a magnifying glass cursor whilst over the icon clearly indicates that clicking on the icon will explode the report. Such a cursor is built into FRAME.

A full list of cursor shapes is documented in the SAS FRAME manual in the description of the _SET_CURSOR_SHAPE_ method.

You may desire to change a cursor shape only while some condition is true. For instance, continuing on the above example, you may want a normal cursor before a report is generated. Then whilst a report is available for viewing, change to the magnifying glass cursor. If the user can delete the report, the magnifying glass is no longer appropriate, so you would want the cursor changed back to the default.

The _RESET_CURSOR_SHAPE_ method is used to set the cursor back to its default shape.

Recently on SAS-L (the Internet SAS forum) the question was posed as to whether a FRAME program could update its display at a given time interval (E.g. every 5 minutes) without user intervention. The user also required that FRAME allow other widgets to be selected whilst awaiting the display update.

Specifically, the update that was desired every 5 minutes is to cause an external database to be re-read and a listbox of selectable values to be updated from that database. The database is constantly being updated by another process.

The technique described here is to place the FRAME SCL in a loop, and that loop ends when one of two conditions is true:

1. the time period has elapsed (in which case read the database, update the display, and re-start the loop)

2. the user selects another widget on the screen (in which case the widget processing should be completed, then the loop re-started)

It is quite a simple matter to code the loop. The difficulty is exiting the loop when a widget is selected. In fact, the difficulty

is even greater than that. Because SCL is processing, the display will have the hour-glass cursor displayed, so the user (who knows that the hour-glass cursor means they don't do anything) cannot select another widget anyway.

If you use the EVENT() SCL function within the loop, FRAME can detect the event generated whenever the mouse button is clicked. If that click is over a widget, the usual widget processing occurs when you exit the loop (i.e. you detect the event and exit the loop, allowing the widget code to execute).

Now the user still sees only the hour glass cursor, so has no idea that the event function will trap a mouse click. You need to understand that the fact that an hour glass cursor is present doesn't have to mean that no processing can take place.

So, to help the user along, just before the loop change the cursor to be a standard pointer. Now the loop is executing, the user doesn't know that something else is happening, the database update occurs, and the user can select a widget for some processing!

Neat stuff this cursor tracking.

See Appendix two for details of the code to support this technique. In that appendix I have an example that automatically shuts down SAS after a period of inactivity in a FRAME screen. The database update example above is easily incorporated by changing the code that shuts down SAS to code to read the database and update the screen.

Error Handling

I presented a paper at SUGI 21 that demonstrated the basic ideas here, but this one is somewhat of a refinement.

The basic idea is to allow error messages for more than one widget to be displayed. The _MSG_ area displays only the current text in _MSG_, and can only easily display text for one error. If several fields are in error, it is often advantageous to be able to see all the errors. This is particularly so when errors arise from cross variable checks.

There wouldn't be a lot of point in attempting to display every error message simultaneously. There would be too much information on screen, and it would not be a simple matter to work out which message went with a given widget.

The technique described here allows a message to be displayed as you move the cursor across a widget. Hence, since the technique is object oriented, it can easily display different messages as the cursor moves over different widgets.

Essentially, an extended text entry exists to display the error messages. That is hidden at the start of the FRAME. When a field is to in error, you define the error to the error handler widget via a method I have written called _SHOW_ERROR_. That method switches on cursor tracking for the widget, and stores the error message in the object. When the cursor moves over the object (widget), the cursor tracker runs, fetches the message from the object, and displays the message (and unhides the object).

The FRAME, as well as the widget has a cursor tracker. That is used simply to hide the error handler extended text entry again. Thus, if the cursor is in the FRAME region and not on a widget, the error handler is hidden (because it is widget specific and no widget is being tracked when the cursor is in the FRAME region).

The code for this is in Appendix three.

Auto display of images

This is really quite a simple use of cursor tracking. It is useful if you need to display something that doesn't fit on the screen, i.e. something can be displayed and removed, occupying no screen real estate when not displayed.

Images tend to take quite a lot of screen real estate. The technique shown here causes an image to display full size when the cursor moves onto a thumbnail version of it. That widget occupies very little screen estate.

Of course, you could just click on the widget that causes the image to display. But we all know how to click a mouse button, right? And we're learning about cursor tracking here.

In this example I need to display a staff members picture. There is already a lot of info about the staff member on screen, and not enough space for the image. So, when the user moves the cursor onto the staff member picture thumbnail, the frame swaps out existing fields, enlarges the image, and then shrinks the image immediately upon moving the cursor off the enlarged image.

See Appendix four for code for this.

Appendices: Code to implement above examples

In the following appendices the code is assumed placed in a catalog called DGS.TOOLBOX. Unless stated otherwise, code resides in METHODS.SCL.

One: Tooltips in the _MSG_ area

Place the following code in METHODS.SCL.

```
/*===================================

Methods used for tooltip control

initial :   INITIALISE_TOOLTIPS
          initialises the tooltip environment at runtime
          on the widget  executing the method

exit    :   EXIT_TOOLTIPS
          switches off the tooltips environment at runtime

tracker :   _CURSOR_TRACKER_ (for widget)
          displays the tip for the current widget.
          Overrides any existing  message.

f_track :   _CURSOR_TRACKER_ (for frame)
          removes an existing tip message

===================================*/

initial:
method optional=rc 8 ;
   if nameditem(_self_,'tooltip') eq 0 then do ;
       rc = 4000 ;
       put 'Error: Unable To Initialise Tips Subsystem';
       call putlist(_self_) ;
       return ;
    end ;

    call send(_self_,'_set_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','tracker',
         'after') ;
    call send(_frame_,'_set_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','ftrack',
         'after') ;
    call send(_self_,'_cursor_tracking_on_') ;
endmethod ;

exit:
```

```
method ;
  call send(_self_,'_cursor_tracking_off_') ;
endmethod ;

tracker:
method x y 8 ;
  call send(_frame_,'_set_msg_',
            getnitemc(_self_,'tooltip')) ;
  call send(_frame_,'_cursor_tracking_on_') ;
endmethod ;

ftrack:
method x y 8 ;
  call send(_frame_,'_set_msg_',' ') ;
endmethod ;
```

In the above the widget cursor tracker methods are first overriden in the INITIAL labeled section (using _self_). This is caused by calling the initialise_tooltips method in the FRAME init section (see below). Then the cursor tracker methods are overriden. The basic idea with cursor tracking is that moving over a widget causes the tracker code to run, moving off it undoes the effect of that code. Moving off a widget is equivalent to moving onto the FRAME, region unless regions overlap.

Now override the FRAME class and add the following _INIT_ override

```
f_init_:
 method ;
   wclass= loadclass('sashelp.fsp.widget');
   call send(wclass,
      '_set_method_','initialise_tooltips',
      'dgs.toolbox.methods.scl',
      'initial') ;
   call send(wclass,
      '_set_method_','exit_tooltips',
      'dgs.toolbox.methods.scl','exit') ;
   call super(_self_,_method_) ;
 endmethod ;
```

This causes every widget that is created to have two new methods defined. Effectively, this override causes a runtime alteration to the base widget class.

Now create a FRAME entry with (e.g. two text entries named birthdte and deathdte), and add the following SCL in the INIT section:

```
 call notify('.','_get_widget_','birthdte',
             wid_id) ;
 rc = setnitemc(wid_id,'tooltip',
    'Enter The Birthdate In Format DD/MM/YY Here') ;
 call notify('birthdte',
             'initialise_tooltips') ;
 call notify('.','_get_widget_','deathdte',
             wid_id) ;
 rc = setnitemc(wid_id,'tooltip',
    'Enter The Deathdate In Format DD/MM/YY Here') ;
 call notify('deathdte',
             'initialise_tooltips') ;
```

Execute the FRAME. When moving the cursor across the text entries you should see the help messages appear in the message area.

Note that you can use the exit_tooltips method to switch off the tooltip handling for widgets if you desire.

Two: Time Driven Event Handling

Override the _OBJECT_LABEL_ method. This is also the _MAIN_LABEL_ override, and a method in its own right.

The following code (the POSTINIT and the OBJLABEL methods) are stored in DGS.TOOLBOX.KILLSAS.SCL.

```
length killtime 8 _method_ $ 40 ;

objlabel:
 method ;
  /* this is the override used for all the methods that
     do the timing countdown to kill the SAS session */

  if _method_ ne 'START_KILL_SAS_SESSION'
       then call super(_self_,_method_)  ;

  /* make sure they give an interval in the local env
     list after which the user is to be zapped */

  if nameditem(envlist('l'),'get_user_off_sas_time') eq 0
   then return ;

  /* find out how long the user has left */

  killtime =
  getnitemn(envlist('l'),'get_user_off_sas_time')
   +time();

  /* change the cursor to a normal arrow so it doesn't
      look like anything is happening */

  call send(_frame_,'_set_cursor_shape_',2,'y');

  /* throw the system into a loop, looking for either a
     mouse click,function  key or enter. If one occurs,
      just exit this override and permit normal
      execution. If the countdown goes beyond their
      limit then destroy them */

  do until(event())  ;
    if event() then leave ;

/* note that the call execcmd in the next line could
    be any processing you desire, E.G. read an external
    database and update a listbox based on that database
    for example */

    if time() > killtime then call execcmdi('endsas') ;
  end ;
endmethod ;
```

Override the _POSTINIT_ method for the FRAME.

```
postinit:
 method ;
   call super(_self_,'_postinit_') ;

  /* define a 15 minute period after which the user will
     be exited (or some other process occur) */

   killtime = '00:15:00't ;
   rc = setnitemn(envlist('l'),killtime,
        'get_user_off_sas_time') ;

  /* start by finding the widgets in the frame and
     creating a per-instance override for each one
     overriding the _OBJECT_LABEL_ method so the
     override gets applied to every widget in the frame
   */

   widget_list = makelist() ;
   call send(_frame_,'_get_widgets_',widget_list) ;
   do i=1 to listlen(widget_list) ;
      widget = popl(widget_list) ;
      call send(widget,'_set_instance_method_',
        '_object_label_',
        'dgs.toolbox.killsas.scl','objlabel') ;

     /* for the first widget keep its object identifier as
         we are going to define a new method to
         execute (don't use _object_label_ as it trigger
         the objects labeled section which is not
         desirable right  here */

      if i=1 then do ;
         call send(widget,'_set_instance_method_',
               'start_kill_sas_session',
               'dgs.toolbox.killsas.scl','objlabel') ;
         first = widget ;
       end ;
   end ;

 /* now override the frames _MAIN_LABEL_ method
    so that if the user clicks off of a widget the code in
     the object label override can still run.
     _MAIN_LABEL is used as the FRAME region has
     no concept of the widgets _OBJECT_LABEL
     method, but they will be the same code.

     Note you MUST also fill the master region for this
     to work, use a container box is simplest */

   frame = loadclass('sashelp.fsp.frame') ;
   call send(frame,'_set_instance_method_','_main_label_',
    'dgs.toolbox.killsas.scl','objlabel') ;

   /* start the timer immediately */

   call send(first,'start_kill_sas_session') ;
endmethod ;
```

To trigger the time driven event handler, place the following code in each FRAME that you want it to execute in:

```
init:
  call notify('.','_set_instance_method_','_postinit_',
     'dgs.toolbox.killsas.scl','postinit') ;
return ;
```

Three: Error handler

Create a subclass of the extended text entry class. Call the new class ERRORS.CLASS. The parent is SASHELP.FSP.GLABEL. Change the description to 'Error Handling Class'.

Select the 'Edit Attributes' option. Make the name _ERRORS. Under Rows select 'Unlimited Rows'. Under Scrollbars switch on Vertical Scroll Bars, and set the Protect and Autoflow options under 'Options'. Change the Text Color to Red and click OK to exit the screen.

In my version I use Set Custom Attributes to replace the attribute window. The replacement just issues a message that no attributes are changeable. The whole functionality of this class is already defined, and I do not want it changed in FRAME entries/

Exit the new class definition and add the class to an appropriate resource entry.

In a FRAME class _INIT_ override add the following code:

```
f_init_:
 method ;
   wclass = loadclass('sashelp.fsp.widget') ;

   call send(wclass,'_set_method_','show_error',
             'dgs.toolbox.methods.scl','s_error') ;
   call send(wclass,'_set_method_','exit_error',
             'dgs.toolbox.methods.scl','e_error') ;
   call send(wclass,'_set_method_','init_error',
             'dgs.toolbox.methods.scl','i_error') ;
   call send(wclass,'_set_method_','get_error_status',
             'dgs.toolbox.methods.scl','g_error') ;

   call super(_self_,_method_) ;
 endmethod ;
```

At runtime this adds four new methods to every widget used in the FRAME. Now add the code corresponding to these new methods. This code, and the F_INIT method above can reside in the same SCL entry. That entry physically resides in the same catalog that you created the class in. In the code above, I placed the class, resource entry, and method code in a catalog called DGS.TOOLBOX. The method code is all in METHODS.SCL in that catalog.

Here is the method code:

```
g_error:
  method status 8 ;
  if nameditem(_self_,'currmsg') eq 0 then status = 0 ;
  else if getnitemc(_self_,'currmsg') eq ' ' then
     status=0 ;
  else status = 1 ;
endmethod ;

i_error:
  method ;

/* at initialisation (called from any widget in the init
  section),find the id of the error handler and hide it */

   call send(_frame_,'_get_widget_','_errors',errorid) ;
   call send(errorid,'_hide_') ;
endmethod ;

s_error:
  method message_id $ 8 ;

/* assign the passed message id as the id of the current
   error for this widget */

  rc = setnitemc(_self_,message_id,'currmsg') ;

/* define the instance method that will be used to display
   the error message when the user moves over a widget
   that has had error tracking switched on */

  call send(_self_,'_set_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','w_track','before') ;
  call send(_self_,'_cursor_tracking_on_') ;

/* define the instance method that will run the frame
   cursor tracking */

  call send(_frame_,'_set_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','f_track','before') ;
  call send(_frame_,'_cursor_tracking_on_') ;
endmethod ;

e_error:
  method ;

/* end the error system for a widget. Delete the instance
   method */

  if nameditem(_self_,'currmsg') eq 0 then return ;
  if getnitemc(_self_,'currmsg') eq ' ' then return ;

   call send(_self_,'_delete_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','w_track','before') ;

/* end the error system for the frame. Delete the instance
   method */

  call send(_frame_,'_delete_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','f_track','before') ;

/* set the current message for this object to blank */

  rc = setnitemc(_self_,' ','currmsg') ;
endmethod ;

f_track:
  method x y 8 ;

/* the frame cursor tracker just hides the error widget.
   Note there is no super call here as it is a BEFORE
   method */

   call send(_frame_,'_get_widget_','_errors',errorid) ;
   call send(errorid,'_hide_') ;
endmethod ;

w_track:
  method x y 8 ;

/* check that we haven't somehow got here without things
   being initialised properly. */

   if nameditem(_self_,'currmsg') eq 0 then return ;
   if getnitemc(_self_,'currmsg') eq ' ' then return ;

/* get the list id for the messages for this widget, and
   extract the message text that is to be displayed */

   messages = getniteml(_self_,'messages') ;
   currmsg = getnitemc(_self_,'currmsg') ;
   text = getnitemc(messages,currmsg) ;

/* define a BEFORE instance method to operate as the
    FRAME cursor tracker */

   call send(_frame_,'_set_instance_method_',
         '_cursor_tracker_',
         'dgs.toolbox.methods.scl','f_track','before') ;
   call send(_frame_,'_cursor_tracking_on_') ;

/* find the error handler id, fill it with the appropriate
   text, and unhide the error handler */

   call send(_frame_,'_get_widget_','_errors',errorid) ;
   call send(errorid,'_get_viscol_',ncols) ;
   l_text = length(text) ;
   if ncols lt l_text then do ;
      nrows = ceil(l_text/ncols) ;
      position = 1 ;
      do i=1 to nrows-1 ;
         display_line = substr(text,position,ncols) ;
         call send(errorid,'_set_line_',display_line,i) ;
         position = position + ncols ;
      end ;
      display_line = substr(text,position) ;
      call send(errorid,'_set_line_',display_line,nrows) ;
   end ;
   else call send(errorid,'_set_text_',text) ;
   call send(errorid,'_unhide_') ;
endmethod ;
```

The error handler is now almost completely defined. You still need to define the error messages though. In my applications I have created a FRAME class override that allows me to enter as many error messages as I require via an attribute window at development time. I don't have room to show that here so for a demonstration, I am adding the error messages in the FRAMES INIT section: Note that an error message consists of two parts, an 8 character name and the actual message. In order to allow multiple messages, they are stored in the objects list, as a list called MESSAGES.

```
init:

/* define the messages for a widget called testfld */

 call notify('.','_get_widget_','testfld',_testfld) ;
 message_testfld = makelist() ;
 rc = setnitemc(message_testfld,
                'Numbers Less Then 1000 Are Illegal',
                'm0000001') ;
 rc = setnitemc(message_testfld,
                'Numbers Above 10000 Are Illegal',
                'm0000002') ;
 rc = setniteml(_testfld,message_testfld,'messages') ;

/* call init_error on any widget to initialise the errors
    subsystem */

 call notify('testfld','init_error') ;
return ;
```

Here is a sample labelled section for a field called TESTFLD to display error conditions in the error handler.

```
testfld:
  call notify('testfld','exit_error') ;
  if testfld < 1000 then
    call notify('testfld','show_error','m0000001') ;
  else if testfld > 10000 then
    call notify('testfld','show_error','m0000002') ;
return ;
```

To demonstrate how it all works, follow the steps above, then create a FRAME with a number of widgets. Create some error conditions and set up code like the above INITand TESTFLD section. Error data into the widgets to cause the error to occur, then move the cursor over the widget. You should see the error box appear, disappear when you move the mouse off the widget, then re-appear with a different message if you move the mouse over a different widget that also has error handling on.

Four: Auto Image Display

Place the following code into an entry called DGS.TOOLBOX.IMAGE.SCL:

```
length tmp $ 3 ;

imagecur:
  method x y 8 ;

  /* override for image cursor tracker */

  /* check if the is_large item is in the image list. This
     item is set to YES  if the image is already large. If
     the item is not yet in the list, add itand initialise
     to'NO' */

   if nameditem(_self_,'is_large') eq 0 then
      rc = setnitemc(_self_,'NO','is_large') ;

  /* check if it is already large and bypass the large
      processing if it is */

   if getnitemc(_self_,'is_large') eq 'YES' then return ;

  /* get a list of widgets on the frame and swap them
     all out except the image. This prevents the image
     making them a group. If you do not do this then
     all the 'child' objects of the enlarged image shrink
     as well when the frame cursor tracker shrinks the
     image back */

   reglist = makelist() ;
   call send(_frame_,'_get_widgets_',reglist) ;
   do i=1 to listlen(reglist);
     name=nameitem(reglist);
     widget=popl(reglist);
     if name ne 'SIMAGE' then
       call send(widget,'_swap_out_') ;
   end;
   rc = dellist(reglist) ;

  /* determine the screen rows and columns and
     enlarge the image to within one character of the
      frame boundaries */

   call send(_frame_, "_winfo_", "numrows", nr );
   call send(_frame_, "_winfo_", "numcols", nc );
   call send(_self_,'_move_region_', ., .,nc-1,nr-1,'c') ;

  /* tell the image object that it is large */

   rc = setnitemc(_self_,'YES','is_large') ;
  endmethod ;

framecur:
  method x y 8 ;

  /* override for FRAME cursor tracker */

  /* start by obtaining the image id */

   call send(_self_,'_get_widget_','simage',simage) ;

  /* check if the is_large item is in the image list. This
      item is set to YES if the image is already large. If
      the item is not yet in the list, add it and
      initialise to 'NO' */

   if nameditem(simage,'is_large') eq 0 then
      rc = setnitemc(simage,'NO','is_large') ;

  /* if the image is not large return */

   tmp = getnitemc(simage,'is_large') ;
   if tmp ne 'YES' then return ;

  /* it was large so since we are about to shrink tell
     the object it is now small */

   rc = setnitemc(simage,'NO','is_large') ;

  /* find the top left hand co-ordinates of the image
      and shrink to just a 15x15 pixel size */

   reglist = makelist() ;
   call send(simage,'_get_region_',reglist,'p') ;
   ulx = getnitemn(reglist,'ulx') ;
   uly = getnitemn(reglist,'uly') ;
   call send(simage,'_move_region_',.,.,ulx+15,uly+15) ;

  /* swap all the other objects back in */

   rc = clearlist(reglist) ;
   call send(_frame_,'_get_widgets_',reglist) ;
   do i=1 to listlen(reglist);
     name=nameitem(reglist);
     widget=popl(reglist);
     if name ne 'SIMAGE' then
       call send(widget,'_swap_in_') ;
   end;

   rc = dellist(reglist) ;
  endmethod ;
```

Create a frame and place some objects on it. Make one a very small image named SIMAGE and set SIMAGE To display an image of your choice. Add the following SCL in the init section, and run the FRAME. Move the cursor across the image to see the cursor tracking outcome.

```
init:
 call notify('simage','_set_instance_method_',
        '_cursor_tracker_',
        'dgs.toolbox.image.scl','imagecur','before') ;
 call notify('.','_set_instance_method_',
        '_cursor_tracker_',
        'dgs.toolbox.image.scl','framecur','before') ;
 call notify('simage','_cursor_tracking_on_') ;
 call notify('.','_cursor_tracking_on_') ;
return ;
```