

Window Driver Contents

Overview

wd

Windows Forms

Event Handlers

wdhandler

Entering Information

Form Locales

Other Message Handlers

Wait

System Events

Overview

The J/Windows interface is referred to as the *Window Driver* and provides access under program control to many features of the Windows graphical and operating system environment. It provides essentially two mechanisms for communication:

- you can send instructions and queries to Windows.
- you can react to events signaled by Windows, such as the press of a button on a form.

The verbs `wd` and `wdhandler` defined in script file `winlib.ijs` provide these mechanisms:

- `wd` is used to send instructions and queries to the Window Driver - and is the only way of doing so.
- `wdhandler` is used to react to events signaled by Windows. It is the standard way of doing so, but may be changed if required.

These verbs are defined in the standard profile when you load J.

wd

This is defined as:

```
wd=. 11!:0
```

wd is used monadically and takes an argument of a character string of one or more statements, separated by semicolons. These statements are converted by the Window Driver into the appropriate Windows function calls. The result is the result of executing the last statement. Each statement is a command followed by any required parameters.

Commands supported include:

- creation and use of Windows controls to allow user input
- instructions to the J session manager
- DDE and OLE links to other Windows software
- access to custom controls
- information queries

For details and a list of commands see the wd Commands help file.

The result of wd is the information requested, typically as a character string, or empty if none. For example, the following queries the system metrics used by Windows (screen width, screen height, x logical units and y logical units):

```
wd 'qm'  
640 480 8 16
```

This shows a 640 by 480 screen, with the base dialog font (used when creating forms) being 8 pixels wide and 16 pixels high.

The next example creates a dialog box that allows the user to select a font. The result is returned after the user closes the dialog box, and contains a string of items that describe the font selected:

```
wd 'mbfont'  
"Lucida Console" 10 bold
```

In some cases, the result is a character string delimited by LF, which can be formatted by list. The following displays all control styles:

```
list wd 'qs'  
bs_defpushbutton    bs_lefttext        bs_ownerdraw  
cbs_autohscroll     cbs_sort           es_autohscroll  
es_autovscroll      es_center          es_lowercase  
...
```

An invalid command is signalled as a domain error:

```
wd 'qz'  
|domain error  
| wd'qz'
```

After an error, you can run command `qer` to query the error - the result is an error message, following by the index in the command where the error occurred:

```
wd 'qer'  
invalid command : 0
```

The following example executes Notepad, assuming it is available on your system - if not, try executing another program.

```
wd 'winexec "\windows\notepad.exe" '  
0
```

The next example executes Notepad with the given filename argument (Notepad loads the file).

```
wd 'winexec "\windows\notepad.exe system\main\dates.ijs" '  
0
```

In the case of an instruction, the result is empty. For example, the following sets the session manager to tile open windows (assuming there are two or more windows open), and returns an empty result:

```
$ wd 'smtile'  
0 0
```

Windows Forms

`wd` is used to create the forms (windows and controls) in a user interface. The easiest way to do this is to use the Form Editor, which builds the appropriate `wd` calls for you. For a tutorial introduction to the Form Editor, see lab *Form Editor*.

The following example creates a window `mywin`, adds a push button named `pressme`, then shows it as a topmost window (i.e. it stays on top of any other window):

```
wd 'pc mywin;cc pressme button;pshow;ptop'
```

Once it is displayed, you can move the window away from the J session, so that you can see both the session and the window. You can set focus on the J session or in the window, by clicking on them.

If you now click on the button, it depresses, but otherwise nothing seems to happen. In fact, the click on the button causes an *event*. If a corresponding *event handler* is defined, then it is invoked, otherwise the event is ignored.

The following example is an event handler for the button. To try this out, enter the following definition:

```
mywin_pressme_button=: wdinfo bind 'button pressed'
```

Now when you click on the button, this event handler is run, and an information box pops up.

The event handler `mywin_pressme_button` is an ordinary J verb. It was defined using the standard utility `wdinfo` that displays an information message box, together with a specific argument `'button pressed'`. For example, you can run this program from the J session, by entering:

```
mywin_pressme_button''
```

Event Handlers

An event handler is a verb that is invoked when a Windows event occurs, for example, when a button is pressed. The standard event handling mechanism in J supports three levels of event handler for a given form. If the form name is *abc*, these are:

- `abc_handler` (the *parent* handler)
- `abc_id_class` (e.g. `abc_pressme_button`)
`abc_formevent` (e.g. `abc_close`)
- `abc_default` (the *default* handler)

When an event occurs, the system searches for an event handler in the order given, and executes the first one it finds. Therefore, for a form `abc`:

- if a verb `abc_handler` is defined, it is executed for every event associated with that form
- if no such verb is defined, and the event is for a specific control, such as button `pressme`, then if a verb for that control such as `abc_pressme_button` is defined, it is executed; or if the event is a form event and a verb for that event is defined, such as `abc_close`, then it is executed
- if no such verb is defined, then if `abc_default` is defined, it is executed
- if no event handler verb is found, the event is ignored

Typically, most forms will be written using only the second level of event handler. The other two levels allow the programmer to deal easily with special cases.

We can try this out on the form defined above. First define a default handler:

```
mywin_default=: wdinfo bind 'this is the default'
```

Click on the form to give it focus, then try pressing a function key. Each time, this new default event handler will be executed. However, when you press on the button, its own event handler is executed.

Now try defining a parent handler:

```
mywin_handler=: wdinfo bind 'this handles all events'
```

Once this is defined, it handles all events from the form.

Note that these event handlers are ordinary verbs and can be defined and modified as required. The search for an appropriate event handler takes place at the time the event occurs. For example, delete the parent handler:

```
erase 'mywin_handler'  
1
```

Now the other event handlers will again be used to respond to the form's events.

wdhandler

This is the verb that provides the mechanism described above. When a Windows event occurs, the system typically invokes the following sentence (but does not show it in the session):

```
wdhandler ''
```

To demonstrate this, try defining a new wdhandler as follows:

```
wdhandler=: wdinfo bind 'my new handler'
```

Now any action you take on the form will invoke this new definition. Typically, you would not want to redefine wdhandler, but the fact that you can do so gives you complete control over the way events are handled. To erase your definition and recover the old definition (which is in locale z), enter:

```
erase 'wdhandler'
```

How does the standard wdhandler work? It first queries the event that has been signaled, using wd'q', and assigns the result to a global variable wdq :

```
wdq
+-----+-----+
|syshandler |mywin_handler |
+-----+-----+
|sysevent   |mywin_pressme_button |
+-----+-----+
|sysdefault |mywin_default |
+-----+-----+
|sysparent  |mywin |
+-----+-----+
|syschild   |pressme |
+-----+-----+
|systype    |button |
+-----+-----+
|syslocale  | |
+-----+-----+
|syshwndp   |1388 |
+-----+-----+
|sysfocus  |pressme |
+-----+-----+
|syslastfocus|pressme |
+-----+-----+
|sysinfo    |1 552 146 200 200 192 173 800 600|
+-----+-----+
```

Note that wd'q' only returns information about the last event that occurred. Re-running it will provide new information only if another event has occurred, otherwise it will give a domain error.

The result wdq is a boxed array describing the event and the current state of the form. The first column contain various identifiers, and the second column corresponding values. Note that the first three rows correspond to the three levels of event handler discussed above. wdhandler checks whether any of these event handlers exist, then

- defines each name in the first column with the corresponding value in the second column, for example a global variable `sysfocus` will be defined with the value `pressme`
- executes the first event handler it has found.

As another example, click on the form to give it focus, then press the Esc key. Click on the J session window, and look at the variable `wdq`:

```

wdq
+-----+-----+
|syshandler |mywin_handler |
+-----+-----+
|sysevent   |mywin_cancel  |
+-----+-----+
|sysdefault |mywin_default |
+-----+-----+
...

```

This shows that the second-level event handler for the Esc key is named `mywin_cancel`. Define a verb of this name to close the form:

```
mywin_cancel=: wd bind 'pclose'
```

Now click on the form to give it focus, press the Esc key, and the form will close.

Entering Information

We next look at a simple example of creating a form to enter information. This example is in a script file included with the J system.

First, clear out any existing definitions with:

```
clear''
```

Then open the script file:

```
open 'system\examples\demo\name.ijs'
```

You may find it helpful to print out the script file; to do so, with the script window in focus, select the menu item File/Print. The relevant definitions are as follows:

```
NAME=: 'Jemima Puddle Duck'
```

```
EDITNAME=: 0 : 0
pc editname;
xywh 5 5 70 10;cc name edit;
xywh 85 5 32 12;cc OK button;
xywh 85 20 32 12;cc Cancel button;
pas 6 6;pcenter;pshow;ptop;
)
```

NB. this creates and initializes the form:

```
editname=: 3 : 0
wd EDITNAME
wd 'set name *',NAME
wd ''''''pshow'
)
```

NB. this handles the Cancel button:

```
editname_Cancel_button=: wd bind 'pclose'
```

NB. this handles the OK button:

```
editname_OK_button=: 3 : 0
NAME=: name
wd 'pclose'
)
```

NB. run the form:

```
editname''
```

The script defines a global variable, `NAME`, and a form, `EDITNAME`, with an edit field and OK and Cancel buttons. The following verbs are also defined:

- `editname` is used to create the form. It first resets the Window Driver, then sends the form definition `EDITNAME` to the Window Driver to create the form, then sets the value of the global `NAME` into the name field.
- `editname_Cancel_button` is used to handle a click on the Cancel button. It simply closes the form.

- `editname_OK_button` is used to handle a click on the OK button. It redefines the global `NAME` with the current value of the name field, then closes the form.

Try this out by running the script (select menu option Run/Window):

Change the value of the name field, press OK, and check the value of the global, `NAME`.

Form Locales

A key point about forms is that they may be created and run in any locale, in fact this would typically be the case. Forms can be created as a class, then instantiated as an object when they are to be run. For a description, see the labs on *Locales* and *Object Oriented Programming*.

When a form is created, the current locale is recorded as the form locale. This locale is part of the event information, and allows an event to be handled by the form handler in the locale.

For example, this means a form can be run in its own locale, without conflicting in any way with definitions in other locales. You can design a form in the base locale, and run it without change in another locale.

To experiment with this, switch to the J session, and clear out existing definitions in the base locale:

```
clear''
```

Check there are no definitions in the base locale:

```
names''
```

Load the form into a locale myname:

```
'myname' load 'system\examples\demo\name.ijs'
```

The form is shown. Change the name and click OK to close the form and update the global, NAME. Note that there are still no definitions in the base locale:

```
names''
```

However, there *are* definitions in the myname locale:

```
names_myname_''
EDITNAME      NAME
editname      editname_Cancel_button
editname_OK_button  wdq
```

Read the value of the name defined:

```
NAME_myname_
Squirrel Nutkin
```

Other Message Handlers

`wdhandler` is the main message handler, but other handlers may be used if appropriate. There are essentially three alternatives:

- You can replace `wdhandler` with your own definition. This gives you complete control over the way you respond to events.
- You can define a specific handler function for a given form, using the `wd` command `phandler`. For example, when the following form is run, any event specific to it will invoke the sentence: `abc_event 0`:

```
wd 'pc abc;pshow;ptop'  
wd 'phandler "abc_event 0"  
abc_event=: wdinfo bind 'abc form handler'
```

- You can bypass the handler mechanism entirely by issuing the `wd` command `wait`, described below.

Wait

The `wait` command allows you to create a form which bypasses the standard event handler mechanism. When it is used, other forms are disabled - this is referred to as a *modal* form, and is typically used where you want the user to respond to a query, or acknowledge an information message. The mechanism is similar to that used in Windows Common Dialog Boxes.

The `wait` command shows your form, disables events except for that form, and waits for an event. After an event occurs, you can use `wd 'q'` to query the event and respond to it. Events for other forms are only enabled when the form is closed.

For example, the following creates a form with a list box. When you select an entry, and click OK, the form is closed and the index of your entry is returned. While the form is running, other J windows are disabled.

```
PICKNUM=: 0 : 0
pc picknum;
xywh 70 9 34 12;cc ok button;cn "OK";
xywh 8 8 50 57;cc nos listbox;
pas 6 6;pcenter;
)
```

```
picknum=: 3 : 0
wd PICKNUM
wd 'set nos zero one two three four'
wd 'setselect nos 0'
wd 'wait'
res=. wd 'q'
wd 'pclose'
ndx=.({"1 res)i.<'nos_select'
".>{:ndx{res
)
```

When you use a wait command, take care that the form has some means of closing itself, for example, by defining the form with style `closeok`, or explicitly closing the form after the wait is ended.

System Events

Some events are not attached to a specific form, and are considered as system events - these are the timer and DDE events. These events have a `sys_` prefix for their handler names, instead of the `formname_` prefix used for form events. For example, the wd 'q' result for a timer event is:

```
+-----+-----+
|syshandler|sys_handler|
+-----+-----+
|sysevent  |sys_timer  |
+-----+-----+
|sysdefault|sys_default|
+-----+-----+
```

The wd 'q' result for a ddepoke event is:

```
+-----+-----+
|syshandler|sys_handler|
+-----+-----+
|sysevent  |sys_ddepoke|
+-----+-----+
|sysdefault|sys_default|
+-----+-----+
```

To respond to such events, you define an appropriate event handler just as for a form event handler.

For example, define a handler for a timer event that writes the current time to the session, then set the timer to be 1000 milliseconds. The timestamps when Windows signals the timer event are written to the current session

```
sys_timer=: (6!:0) (1!:2) 2:
wd 'timer 1000'
1996 1 3 10 15 37.72
1996 1 3 10 15 38.76
1996 1 3 10 15 39.81
1996 1 3 10 15 40.9
```

To switch off the timer, create a new execution session (i.e. a jx window) and enter:

```
wd 'timer 0'
```