Using the Clipboard and Drag and Drop

This chapter covers the following topics:

- Introduction
- Clipboard Specifics
- Drag and Drop Specifics
- Drag and Drop Insertion Marks
- Drag-Drop Checklist

Introduction

Both clipboard and drag/drop data transfer make use of a logical clipboard at the Natural language level, allowing a single set of methods to handle both requirements. The `PROCESS GUI` actions for handling the logical clipboard are as follows: `OPEN-CLIPBOARD`, `SET-CLIPBOARD-DATA`, `CLOSE-CLIPBOARD`, `GET-CLIPBOARD-DATA` and `INQ-FORMAT-AVAILABLE`. Each Natural process has exactly one logical clipboard, which is why it is referred to in the product documentation as the "local" clipboard.

`OPEN-CLIPBOARD` is the first step in building up the logical clipboard data. It takes an optional parameter (owner window), which is typically the handle of the control sourcing the data. If anything was previously on the logical clipboard, this action empties it. Note that you don’t need to call this for drag and drop, because Natural does this implicitly before raising the `BEGIN-DRAG` event (see below).

`SET-CLIPBOARD-DATA` puts the actual data on the logical clipboard. The first parameter is the clipboard format, specified as a string. There are two pre-defined formats (defined in NGULKEY1 as CF-TEXT and CF-FILELIST), which are used for standard text transfer, and lists of files (suitable for data exchange with the Windows Explorer and many other applications) respectively. In addition, an arbitrary string (which should not begin with a dgit) should be used to indicate a private clipboard format that only Natural applications can understand (they just need to know the format string so they can pass it to `GET-CLIPBOARD-DATA` to retrieve the data). The second and subsequent arguments are an arbitrary number of data operands. These can be any combination of arrays (incl. index ranges) or scalars (incl. dynamic and large alpha variables). Arrays are internally expanded into their individual elements, which are then handled individually as for scalars.

For example, the following code:

```
DEFINE DATA LOCAL
1 #ARR(A1/2,3) INIT (1,1)<'A'> (1,2)<'B'> (1,3)<'C'>
                            (2,1)<'X'> (2,2)<'Y'> (2,3)<'Z'>
END-DEFINE
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT #ARR(*,*)
```

is equivalent to:

```
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT
  'A' 'B' 'C' 'X' 'Y' 'Z'
```

For the pre-defined formats, the operands must be alphanumeric (format A). For private formats, the data arguments can be of almost any type. Exception: handle variables (incl. `HANDLE OF OBJECT`) are not supported, because they are process-specific. The data for private formats is stored "as-is" (i.e., no conversion).

Note that multiple data formats can be placed on the clipboard by performing a `SET-CLIPBOARD-DATA` action for each required format. However, any call to `SET-CLIPBOARD-DATA` for a particular format replaces any data that may already exist in that format.

Note also that the data is not placed on the Windows clipboard. This is done when the logical clipboard is closed (see below).

`CLOSE-CLIPBOARD` closes the logical clipboard, and places the data on the Windows clipboard, so that it becomes available for pasting into other applications. The data cannot be modified by `SET-CLIPBOARD-DATA` after this call. Note that this call is not necessary for drag and drop because you usually don't need to also make the dragged data available for pasting. Drag and drop can work directly with the logical clipboard.

`GET-CLIPBOARD-DATA` is used by the application performing the paste or acting as the drop target to retrieves the data from the drag-drop clipboard (if a drag and drop operation is in progress) or from the Windows clipboard otherwise. The drag-drop clipboard is a synonym for the logical clipboard belonging to the source Natural process. `SET-CLIPBOARD-DATA`, a clipboard format is specified, followed by an arbitrary list of data operands (with the same format type restrictions). For private formats, the operands need not have the same format type as used in the `GET-CLIPBOARD-DATA` action. For example, you can place an integer on the clipboard and read it back into a packed numeric (P) variable. Internally, a `MOVE` conversion is done. Therefore, if different format types are used for setting and getting the data, they must be `MOVE`-compatible.

For pre-defined formats, where the individual data items are either delimited by CR/LF (for CF-TEXT format) or by null-terminators (for CF-FILELIST format), only one item is usually read into each receiving field. Exception: If the last receiving field is a dynamic alpha variable, it receives all remaining data items, including the delimiters. This exception allows the application to use (for example) a single dynamic alpha variable to set and get multiple lines of data or multiple file/directory names. Regardless of the format used, if too many receiving operands are specified, the excess fields are reset (see the `RESET` statement). Note that individual data fields may be skipped by using the *n*`X` notation. For example, `5X` skips 5 data items (where a "data item" is a single line for CF-TEXT format).

`INQ-FORMAT-AVAILABLE` is used for querying whether data is available in a given format (see specification for syntax). It is typically used to determine whether to enable or disable the **Paste** command, or whether to display the "no drop" cursor for drag/drop operations.

Clipboard Specifics

The actual clipboard data transfer has been covered above. However, Natural allows you to define signals, menu items and toolbar items of the special types **Cut**, **Copy**, **Paste**, **Delete** and **Undo**, which (unlike normal commands) do not raise `CLICK` events. For input fields, edit areas, selection boxes and table controls, it's obvious what Natural should do, and Natural does this implicitly. For Natural, these commands now support list boxes and ActiveX controls. However, the mechanism is different in this case, because it is ambiguous as to how Natural should respond to these commands. Therefore, Natural needs

some assistance from the application. This assistance comes in the form of six new events: CUT, COPY, PASTE, DELETE, UNDO and CLIPBOARD-STATUS, all of which are suppressible (via the new SUPPRESS-CUT-EVENT, SUPPRESS-COPY-EVENT, SUPPRESS-PASTE-EVENT, SUPPRESS-DELETE-EVENT, SUPPRESS-UNDO-EVENT and SUPPRESS-CLIPBOARD-STATUS-EVENT attributes). All six events are suppressed by default. The CUT, COPY, PASTE, DELETE and UNDO events are raised whenever the respective command is triggered. The corresponding event suppression flags are used by Natural to decide whether to enable or disable the corresponding command(s) in the user interface.

The CLIPBOARD-STATUS event is sent to the focus control during idle processing to give the application a chance to set these event suppression flags dynamically according to the context (e.g., whether or not there is an active selection). Natural raises this event before it queries the event suppression flags for the purpose of clipboard command status updating). Note that these new events are (currently) only sent to list boxes and ActiveX controls (and, of course, only if they currently have the focus). Input fields, selection boxes, etc., are still handled implicitly.

The CLIPBOARD-STATUS event is only raised if there is at least one clipboard command in the user interface that needs to be updated.

The following example shows a typical CLIPBOARD-STATUS event for a list box control:

```
DEFINE DATA LOCAL
1 #CONTROL HANDLE OF GUI
1 #FMT (A10) CONST<'MYDATAFMT'>
1 #AVAIL (L)
END-DEFINE
...
#CONTROL := *CONTROL
/*
/*
Cut, Copy & Delete are enabled if an item is selected,
/*or disabled otherwise
/*
IF #CONTROL.SELECTED-SUCCESSOR <> NULL-HANDLE
  #CONTROL.SUPPRESS-CUT-EVENT := NOT-SUPPRESSED
  #CONTROL.SUPPRESS-COPY-EVENT := NOT-SUPPRESSED
  #CONTROL.SUPPRESS-DELETE-EVENT := NOT-SUPPRESSEDELSE
  #CONTROL.SUPPRESS-CUT-EVENT := SUPPRESSED
  #CONTROL.SUPPRESS-COPY-EVENT := SUPPRESSED
  #CONTROL.SUPPRESS-DELETE-EVENT := SUPPRESSED
END-IF
/*
/* Paste command is enabled if data is available in a
/* recognized format, or disabled otherwise
/*
PROCESS GUI ACTION INQ-FORMAT-AVAILABLE
 WITH #FMT #AVAIL GIVING *ERROR
/*
IF #BOOL
  #CONTROL.SUPPRESS-PASTE-EVENT := NOT-SUPPRESSED
ELSE
  #CONTROL.SUPPRESS-PASTE-EVENT := SUPPRESSED
END-IF
```

Drag and Drop Specifics

Drag and drop operations can be triggered automatically (for list boxes and bitmap controls) or manually, via the new PERFORM-DRAG-DROP action (typically for ActiveX controls in response to control-specific mouse click or drag start events). For automatic drag/drop, the mouse cursor must be over the active selection (if any). For manual drag/drop, the parameters for PERFORM-DRAG-DROP include the handle of the control that should receive the drag/drop events (the drag source), and an optional flag indicating whether drag and drop should begin immediately, or only after the user moves the mouse a system-defined minimum number of pixels. Both automatic and manual drag/drop use the same code internally, so the same events are received in both cases.

Drag/drop is controlled by by two new I4 attributes, DRAG-MODE (for drag sources) and DROP-MODE (for drop targets). These attributes can be set to one of 8 values (defined in NGULKEY1): DM-NONE (no drag/drop allowed), DM-COPY (copy allowed), DM-MOVE (move allowed), DM-COPYMOVE (copy and move allowed), DM-LINK (link allowed), DM-COPYLINK (copy and link allowed), DM-MOVELINK (move and link allowed), DM-COPYMOVELINK (copy, move and link allowed). Link operations imply that the drop target should create a link to the source data, rather than creating a copy of it. For file operations, desktop shortcuts are typically used (not currently explicitly supported by Natural). Drag operations are only initiated if the source's DRAG-MODE attribute is set to something other that the default DM-NONE value. In addition, the application must respond to the BEGIN-DRAG event (see below).

Control types capable of acting as drop targets are: ActiveX controls, bitmap controls, list boxes, control boxes, edit areas, and dialogs (tab controls and table controls are planned for the future but are not currently supported). These windows are, however, only registered with OLE as drop targets if their DROP-MODE attribute is set to something other that the default DM-NONE value. During a drag/drop operation, OLE automatically searches up through the window hierarchy, starting with the window immediately under the cursor, until it finds a window that has been registered as a drop target. This is the window that gets the OLE drop notifications and therefore is the window that receives the Natural drag/drop events (see below).

The new drag/drop related events are: BEGIN-DRAG, END-DRAG, DRAG-ENTER, DRAG-OVER and DRAG-LEAVE. In addition, the existing DRAG-DROP event (for the Mickey Mouse non-OLE drag/drop support for bitmap controls) is also used. All events are suppressible via the appropriate event suppression attributes (SUPPRESS-BEGIN-DRAG-EVENT etc.), all of which are SUPPRESSED by default.

The BEGIN-DRAG event (if not suppressed) is sent to the drag source on initiation of a drag operation. The application *must* use the SET-CLIPBOARD-DATA action to place some data on the drag/drop clipboard before returning from this event, otherwise the drag/drop operation is implicitly cancelled without the mouse cursor having changed. Note that there is no need to call either of the OPEN-CLIPBOARD or CLOSE-CLIPBOARD actions.

The END-DRAG event (if not suppressed) is sent to the drag source after a drag/drop operation has completed (even if the drag operation was cancelled). The main use of this event is to delete the source data if a Move operation occurred. The application can find out whether a Move operation has occurred by calling the existing INQ-DRAG-DROP action, which has been extended with two new optional integer output parameters. The first of these new parameters indicates which mouse buttons are currently pressed (1 = Left button, 2 = Right button, 4 = Middle button, or a combination thereof). The second new parameter is the one we need here, and contains the drop effect resulting from a drag/drop operation (DM-NONE if no drop or if the operation was cancelled, DM-COPY if a Copy operation was performed, DM-MOVE if a Move operation was performed, and DM-LINK if a link operation was performed.

The DRAG-ENTER event (if not suppressed) is sent to the drop target when the drag cursor (re-)enters the region occupied by the drop target. The application typically responds to this event by calling the INQ-FORMAT-AVAILABLE action to find out if a compatible data format is available on the clipboard, and then setting the SUPPRESS-DRAG-DROP-EVENT attribute accordingly. The SUPPRESS-DRAG-DROP-EVENT is important because it not only determines whether the DRAG-DROP event should be raised, but also informs Natural as to whether a drop should be allowed. After raising the DRAG-ENTER and DRAG-OVER events, Natural inspects the SUPPRESS-DRAG-DROP-EVENT attribute and displays a "no drop" symbol. Otherwise, the drop effect is determined by the combination of the drag source's DRAG-MODE value, the drop target's DROP-MODE value, and the augmentation keys (SHIFT and CTRL) that are currently being pressed.

The DRAG-OVER event (if not suppressed) is frequently sent to the drop target as the drag cursor moves over the drop target. It can be used, for example, to update the drop emphasis (if any) as the user traverses the items within the control and/or to update the SUPPRESS-DRAG-DROP-EVENT attribute if the feasibility of a drop operation depends on the position within the drop target.

The DRAG-LEAVE event (if not suppressed) is sent to the drop target when the drag cursor leaves the region occupied by the drop target without a drop having occurred. This is mainly used (if at all) to remove the drop emphasis (if any) applied in the DRAG-OVER event.

The DRAG-DROP event (if not suppressed) is sent to the drop target when the user performs a drop. drag cursor leaves the region occupied by the drop target without a drop having occurred. The application should respond to this by effectively performing a Paste operation, using the current relative position within the control, if necessary. Both the relative position and the type of operation can be retrieved via the INQ-DRAG-DROP action. The latter is returned in the new (optional) "drop effect" parameter (see the description of the END-DRAG event above for more information).

Drag and Drop Insertion Marks

For list boxes, a new "insertion mark (i)" style can be used to indicate that a dashed horizontal line be used to indicate the current insert position when the drag cursor is moved over the control (assuming it is a drop target). The application cannot query the insertion mark position directly, but can find out where to insert the data by querying the relative position within the control via the INQ-DRAG-DROP action, then passing these coordinates to the INQ-ITEM-BY-POSITION action, as in the following example:

```
DEFINE DATA LOCAL
1 #Y (I4)
1 #CONTROL HANDLE OF GUI
1 #ITEM    HANDLE OF GUIEND-DEFINE
...
/* DRAG-DROP event:
PROCESS GUI ACTION INQ-DRAG-DROP WITH 4X #Y GIVING *ERROR
*
IF #Y < 0
   #Y := 0
END-IF
#CONTROL := *CONTROL
PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
 #CONTROL 0 #Y #ITEM GIVING *ERROR
```

After the above code has executed, the variable #ITEM contains the handle of the item immediately following the insertion point. You can then dynamically insert one or more list box items at this position by calling the ADD action with the WITH PARAMETERS clause, setting the SUCCESSOR attribute to #ITEM.

Note that the correction for negative y-coordinate in the above example is necessary to cover the situation where the drop position is on the list boxes top border. If no correction would be made here, #ITEM would be set to NULL-HANDLE and the new list box item(s) would be added undesirably at the end of the list instead of at the beginning if we were to directly use #ITEM as the SUCCESSOR attribute, as described above.

Drag-Drop Checklist

For convenience, here is a brief overview of the steps involved in implementing drag-drop in Natural applications:

1. Set the DRAG-MODE for each drag source. If the drag source is a bitmap control, its DRAGGABLE attribute must also be set to TRUE.
2. Set SET-CLIPBOARD-DATA in the BEGIN-DRAG event for each drag source to provide the transfer data.
3. Set the DROP-MODE for each drop target.
4. In the DRAG-ENTER event, use the INQ-FORMAT-AVAILABLE action to set the SUPPRESS-DRAG-DROP-EVENT attribute to NOT-SUPPRESSED (0) if a supported clipboard format is available, or SUPPRESSED (1) otherwise. If the control can also act as a drag source and you need to prohibit drag-drop operations within the control, call INQ-DRAG-DROP to get the source control handle and compare it to the current control (*CONTROL), suppressing the drag-drop event if both are identical.
5. If the effect of the drop is position-sensitive within the target control, use the INQ-DRAG-DROP action within the DRAG-OVER event to get the current position, determine the item under the drag cursor (e.g. via the INQ-ITEM-BY-POSITION action) and set the SUPPRESS-DRAG-DROP-EVENT attribute appropriately. Highlight the current item if desired.
6. If the current item was highlighted in step 5 above, unhighlight it (if necessary) in the DRAG-LEAVE and (potentially) DRAG-DROP events.
7. Use GET-CLIPBOARD-DATA in the DRAG-DROP event to retrieve the transfer data and process it accordingly.
8. In the END-DRAG event for the drag source, delete the source data if the drop effect returned by INQ-DRAG-DROP is set to DM-MOVE.
9. If the drag source is an ActiveX control, call the PERFORM-DRAG-DROP action to initiate the drag-drop operation in response to a "MouseDown" event (for example) if a location within the current selection is clicked.

Example - Use of X-Arrays for Transferring Data

One of the problems in setting or retrieving data that may need to be placed or already have been placed on the Windows or drag-drop clipboard in response to a user interaction is being able to cope with an arbitrary amount of data at run-time. For example, the user may select a single, a few, or possibly even hundreds or thousands of list box items before performing a clipboard or drag-drop operation on them. With fixed-size arrays, one would have to define huge arrays to cope with the worst-case scenario, even though typically only a small percentage would be used most of the time.

There are two possible solutions to this problem available in Natural. The first way is to use a single dynamic alpha variable to contain all items to be set or retrieved. The application is then responsible for building up the items (including delimiters) in the dynamic variable before calling SET-CLIPBOARD-DATA, and for extracting the items from the dynamic variable after calling GET-CLIPBOARD-DATA. This approach is not possible for private formats, because these are not delimited.

The second approach is to make use of X-Arrays. For setting clipboard data, these behave similarly to fixed-size arrays, except that their size can be modified to contain exactly the number of elements needed in a specific situation. For example, if there are 17 items that need to be written to the clipboard, then you can use:

```
DEFINE DATA LOCAL
1 #X-ARR(A80/1:*)
1 #UPB (I4) INIT <17>
END-DEFINE
RESIZE ARRAY #X-ARR TO (1:#UPB)
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT #X-ARR(*)
```

instead of having to use a wastefully large fixed array, of which only a small range is used:

```
DEFINE DATA LOCAL
1 #ARR(A80/10000)
1 #UPB (I4) INIT <17>
END-DEFINE
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT ARR(1:#UPB)
```

When retrieving clipboard data, X-Arrays are even more useful, because the application does not know in advance how many items are on the clipboard. Passing all array elements (10,000 in the above example) would be relatively slow, because all unused elements need to be reset.

However, if an X-Array is used instead, Natural automatically resizes the array to (1:N), where N is the minimum of the number of items (remaining) on the clipboard and the array's maximum upper bound (as defined in DEFINE-DATA, where * indicates the maximum possible value). Note that there are three restrictions on the use of X-Arrays in conjunction with GET-CLIPBOARD-DATA:

- The X-Array must be the last (or only) parameter.
- Only 1-dimensional X-Arrays are supported.
- The X-Arrays defined range must include the element 1.

Here is an example program illustrating the use of a dynamic X-Array for retrieving clipboard data, including the use of a second X-Array to store and display the data lengths:

```
DEFINE DATA LOCAL
1 #FMT (A10) CONST<'MYDATAFMT'>
1 #X-ARR (A/1:*) DYNAMIC
1 #X-LEN (I4/1:*)
1 #UPB (I4)
1 #I (I4)
END-DEFINE
PROCESS GUI ACTION OPEN-CLIPBOARD GIVING *ERROR
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH #FMT
  'MIKE' 'FRED' 'JIM' 'LULU' 'FRANK' 'JANA' 'ELIZABETH'
'TONY'
  GIVING *ERROR
```

```
PROCESS GUI ACTION CLOSE-CLIPBOARD GIVING *ERROR
PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH #FMT #X-ARR(*)
  GIVING *ERROR
#UPB := *UBOUND(#X-ARR)
RESIZE ARRAY #X-LEN TO (1:#UPB)
FOR #I 1 #UPB
  #X-LEN(#I) := *LENGTH(#X-ARR(#I))
END-FOR
DISPLAY #X-ARR(*) (AL=10) #X-LEN(*) / '*** END OF DATA ***'
END
```