Storing and Retrieving Client Data for a Dialog Element

Storing and Retrieving Client Data for a Dialog Element

This chapter covers the following topics:

- Introduction
- Integer Data
- Handle Data
- Keyed Alphanumeric Client Data
- Keyed Client Data in Native Format
- Key Enumeration

Introduction

This section refers to the association of arbitrary user-defined information ("client data") with a (dialog or) dialog element. There are various complementary ways of achieving this, which will be discussed in detail in the following sections. The attributes and actions relating to the manipulation of client data in Natural are (in the order they are discussed in this document):

- `CLIENT-DATA` attribute
- `CLIENT-HANDLE` attribute
- `CLIENT-KEY` attribute
- `CLIENT-VALUE` attribute
- `SET-CLIENT-VALUE` action
- `GET-CLIENT-VALUE` action
- `ENUM-CLIENT-KEYS` action

Integer Data

For a number of dialog element types, the `CLIENT-DATA` attribute may be used to associate a single arbitrary 4-byte integer value with the dialog element. This may be useful for linking data to a specific dialog element. A list box item, for example, can receive and pass on the ISN of a database record. The `CLIENT-DATA` attribute value may be changed at any time.

In Natural code, this might look like this:

```
DEFINE DATA
LOCAL
  1 #LBITEM-1 HANDLE OF LISTBOXITEM

  1 #ISN (I4)
   ...
END-DEFINE
...
READ...
   #LBITEM-1.CLIENT-DATA:= #ISN
END-READ
...
```

Note:
The CLIENT-DATA attribute of a dialog is reserved for its dialog ID, and should not be used for the storage of user-defined client data.

Handle Data

Similarly, for all dialog element types, the CLIENT-HANDLE attribute may be used to associate a single arbitrary GUI object handle with the dialog element. For example, in the section *Working with Dialog Bar Controls*, sample generic code is provided for building up a context menu containg entries for each tool bar control and dialog bar control in use by the dialog, allowing the user to individually show and hide them. In this example, the CLIENT-HANDLE attribute of each such menu item is set to the handle of the respective tool or dialog bar, allowing it to be both simply and directly retrieved when the menu item is clicked.

Keyed Alphanumeric Client Data

Note:
The term "keyed" refers to the ability to store multiple items of information for a given dialog element, each item being stored under a unique retrieval key.

Client data may also be set and retrieved as an alphanumeric string of up to 253 characters by using the CLIENT-KEY and CLIENT-VALUE attributes in combination.

To update a dialog element with a particular string

1. You first assign a value to the dialog element's CLIENT-KEY attribute, if this attribute does not already contained the desired value. This determines the key under which the string is to be stored for a dialog element.

2. You then assign an alphanumeric string to the CLIENT-VALUE attribute of the dialog element.

This enables you to store a number of key/value pairs for one dialog element.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
#LB-1.CLIENT-VALUE:= 'ANYSTRING'        /* The string to be stored
```

Note:
In this and all following examples, the handle variable #LB-1 is used, which (by convention) normally refers to a list box. However, with the exception of the CLIENT-DATA attribute, client data can be associated with GUI objects of any type, even those without a user interface, such as timers or signals.

To query a dialog element for a particular string

1. You first assign a CLIENT-KEY value to the dialog element, if this attribute does not already contained the desired value.
2. Then you query the CLIENT-VALUE attribute for the dialog element to retrieve the corresponding value.

If you query the CLIENT-VALUE of a CLIENT-KEY and there is no such key among the key/value pairs of the dialog element, an empty string is returned.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
IF #LB-1.CLIENT-VALUE EQ 'ANYSTRING' THEN
...
END-IF
```

If non-alphanumeric data is to be stored and retrieved, getting the data back into the original format may be a little more complicated, as shown below.

Example:

```
DEFINE DATA LOCAL
01 #DATE (D)
...
END-DEFINE

#LB-1.CLIENT-KEY := 'ANYKEY'
/* Store the current date
#LB-1.CLIENT-VALUE := *DATX

/* Retrieve it as a date (D) field
STACK TOP DATA #LB-1.CLIENT-VALUE
INPUT #DATE
```

The STACK statement retrieves the client value in alphanumeric form and places it one the Natural stack, from which the INPUT statement unstacks it into the specified variable, #DATE, implicitly converting the data from alphanumeric to date form. Alternatively, it would be possible to retrieve the client value into an alphanumeric variable, followed by explicitly converting it to the date field via a MOVE EDITED statement. However, the above approach has the advantage that it is not dependent on the date format (DTFORM), as well as not requiring the above-mentioned alphanumeric variable.

For some data types, such as dates and times, the default alphanumeric representation of the type (as used by the CLIENT-VALUE attribute) does not contain all the information contained in the original data type. For example, the default alphanumeric represention for time (T) values only contains the hours, minutes and seconds, and does not contain either the date component or tenths of a second. Similarly, the default alphanumeric represention for date (D) values does not contain century information. Thus, in order for the correct century to be assumed in the above example, it may be necessary to set the "Sliding Window" (YSLW) parameter correctly before running the program.

If a dynamic alpha variable is used to directly receive the CLIENT-VALUE attribute value, the resulting value will have a length of 253 characters, being padded with blanks if necessary. This is due to the use of an attribute buffer of format A253 internally, and will be discussed later. The same effect is obtained when assigning an explicitly-defined A253 field to a dynamic variable. In either case, to prevent these trailing blanks from being stored in the dynamic variable, a COMPRESS statement should be used instead of a simple MOVE or assignment, as shown below.

```
DEFINE DATA LOCAL 01 #DYN (A) DYNAMIC ... END-DEFINE
#DYN := 'ANYSTRING' /* Set the client data #LB-1.CLIENT-KEY := 'ANYKEY' #LB-1.CLIENT-VALUE
:= #DYN /* Retrieve value as 253-character string: #DYN := #LB-1.CLIENT-VALUE
/* Retrieve value without trailing blanks: COMPRESS #LB-1.CLIENT-VALUE INTO #DYN
```

Regardless of which of these approaches are used, any trailing blanks in dynamic alphanumeric variables are effectively lost if stored and retrieved via the CLIENT-VALUE attribute.

To delete a particular string for a dialog element

1. You first assign a CLIENT-KEY value to the dialog element, if this attribute does not already contained the desired value.
2. Then you RESET (or explicitly assign an all-blank value to) the CLIENT-VALUE attribute for the dialog element to delete the corresponding value.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY' RESET #LB-1.CLIENT-VALUE
```

Keyed Client Data in Native Format

As an alternative to setting client data in alphanumeric string form using the CLIENT-KEY and CLIENT-VALUE attributes in combination, the SET-CLIENT-VALUE and GET-CLIENT-VALUE actions may be used to store and retrieve client data directly in the supplied format, with no conversion. The value may, however, be stored in compressed form. In particular, trailing blanks in non-dynamic alphanumeric data are not stored, in order to save space. For example, if you supply an A253 field containing the value FRED followed by 249 filler blanks, only the A4 value FRED will be stored as client data internally. This latter optimization also applies to client data stored via the CLIENT-VALUE attribute.

The two techniques may be intermixed (i.e., one technique used to set the data and the other technique used to retrieve it). However, the use of the actions provides a number of advantages over the use of the attributes, as will become clear in the following sections.

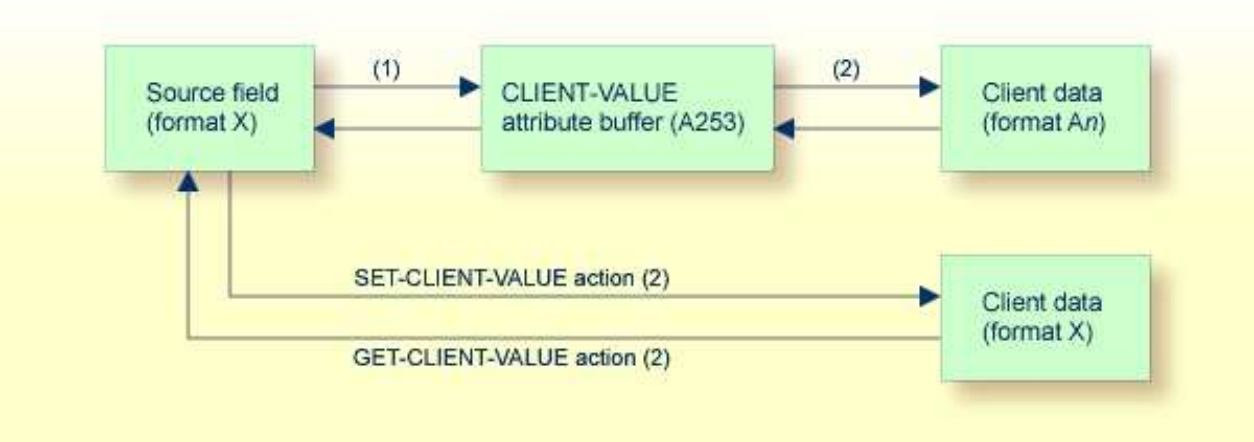
To update client data for a dialog element using the action-based technique

- Call the SET-CLIENT-VALUE action, passing the handle of the dialog element, the (client) key under which the value is to be stored, and the value itself. Alternatively, the key parameter can be omitted, in which case the current value of the dialog element's CLIENT-KEY attribute is implicitly used as the key.

Example:

```
#LB-1.CLIENT-KEY := 'ANYKEY' /* The following three statements are equivalent
ways of setting the same /* information: /* (1) attribute-based approach: #LB-1.CLIENT-VALUE
:= 'ANYVALUE' /* (2) action-based approach, with explicitly-specified key PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 'ANYVALUE' 'ANYKEY' GIVING *ERROR /* (3)
action-based approach without key; CLIENT-KEY attribute implicitly used PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 'ANYVALUE' GIVING *ERROR
```

A significant advantage of storing client data via the `SET-CLIENT-VALUE` action is that there is no intermediate conversion to alphanumeric (A253) format involved, as is the case if the `CLIENT-VALUE` attribute is used. This is shown in the following diagram, where format X is used to imply any particular data type, and format A*n* represents an alphanumeric value stripped of any trailing blanks:



Here we see that the storage and retrieval of client data via the `CLIENT-VALUE` attribute is a two-step process (as is indeed the case for all attributes in Natural) depicted by the arrows (1) and (2) above, involving an attribute buffer corresponding to the defined format for the attribute - in this case A253. In contrast, the use of the `SET-CLIENT-VALUE` and `GET-CLIENT-VALUE` actions is a single step process that is effectively equivalent to performing step (2) alone, by-passing the conversion between the attribute buffer and the source or target field. This offers the following advantages (aside from being somewhat faster):

- Alphanumeric data longer than 253 characters may be stored without truncation, due to not having to pass through the attribute buffer.
- Handle values may be stored. These are incompatible with the use of an alphanumeric attribute buffer, because conversions between handles and alphanumeric fields are not allowed.
- If the data is being sourced from a dynamic alphanumeric variable, any trailing blanks are preserved. If the attribute buffer is used, trailing blanks become indistinguishable from (and are assumed to be) buffer filler characters and are thus stripped from the value when it is stored.
- Because the data is stored without conversion to and from alphanumeric format, non-alphanumeric data may be stored without any loss of information. For example, date information and tenths of a second are not lost when time values are stored, and century information is not lost when dates are stored.

In addition, there are other advantages to using the action-based approach for client data storage:

- Alphanumeric values consisting entirely of blanks may be stored. This is not possible via the CLIENT-VALUE attribute, as this would imply a delete operation.
- Error codes (e.g., in the case where an invalid control handle is passed) are returned in the GIVING field (if specified), without standard error handling necessarily being invoked (although this can be achieved, if desired, by the use of GIVING *ERROR).

To query client data for a dialog element using the action-based technique

- Call the GET-CLIENT-VALUE action, passing the handle of the dialog element, the (client) key for which the value is to be retrieved, and a field to receive the value itself. Alternatively, the key parameter can be omitted, in which case the current value of the dialog element's CLIENT-KEY attribute is implicitly used as the key.

Example:

```
DEFINE DATA LOCAL 01 #VALUE (A253) ... END-DEFINE PROCESS GUI ACTION GET-CLIENT-VALUE
WITH #LB-1 #VALUE 'ANYKEY' GIVING *ERROR IF #VALUE <> ' ' /* Value found ... ELSE
/* Value not found ... END-IF
```

Note that the format of the field specified to receive the value must be MOVE-compatible with the format of the stored value.

If the specified key is not found for the specified dialog element, the value field is RESET. For example, an alphanumeric receiving field is filled with blanks, and a numeric receiving field is set to zero.

However, if such values can be explicitly stored for this key by the program, the value alone cannot be used to determine whether the requested client data was found.

To query client data if resetted values are being explicitly stored

- Call the GET-CLIENT-VALUE action, also passing (in addition to the standard parameters mentioned above) a field of type L to receive the found/not found status.

Example:

```
DEFINE DATA LOCAL 01 #VALUE (A253) 01 #FOUND (L) ... END-DEFINE * PROCESS GUI
ACTION GET-CLIENT-VALUE WITH #LB-1 #VALUE 'ANYKEY' #FOUND GIVING *ERROR * IF #FOUND
... END-IF
```

The main advantage of reading client data via the GET-CLIENT-VALUE action is again the avoidance of going via an attribute buffer (see earlier diagram), implying that no intermediate conversion to or from alphanumeric (A253) format involved, as is the case if the CLIENT-VALUE attribute is used. Instead, the stored data is converted directly to the format of the receiving field for the value. This offers the following advantages:

- Alphanumeric data longer than 253 characters may be retrieved, without being truncated to the length of the (not used) CLIENT-VALUE attribute buffer.
- Handle values may be retrieved, which are not MOVE-compatible with the alphanumeric format of the CLIENT-VALUE attribute buffer.

- If the data is being read into a dynamic alphanumeric variable, any trailing blanks in stored alphanumeric data are preserved. If the `CLIENT-VALUE` attribute is used, the dynamic variable would receive the buffer's filler characters and be unable to distinguish them from any trailing blanks in the original data.

In addition, Stored alphanumeric values consisting entirely of blanks may be recognized. This is not possible via the `CLIENT-VALUE` attribute, as there is no way to distinguish them from the implicit "not found" value.

To delete client data for a dialog element using the action-based technique

- Call the `SET-CLIENT-VALUE` action, passing the handle of the dialog element and the (client) key for which the value is to be deleted, but omitting the value itself. Alternatively, the key parameter can be omitted, in which case the current value of the dialog element's `CLIENT-KEY` attribute is implicitly used as the key.

Example:

```
/* No value supplied => delete any existing value for specified key PROCESS GUI
ACTION SET-CLIENT-VALUE WITH #LB-1 1X 'ANYKEY' GIVING *ERROR /* Alternatively,
a mixed attribute/action approach can be used: #LB-1.CLIENT-KEY := 'ANYKEY' PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 GIVING *ERROR
```

Key Enumeration

The above sections have dealt with the creation, updating, querying and deletion of client key and client value data. In most cases this is enough. However, in some situations, the keys that are being used by a dialog element are either not known to the code that reads them, or it is necessary to be able to verify that the expected keys are present for debugging or testing purposes. The iterative process of retrieving the keys currently being used by a particular dialog element is known as *key enumeration*.

To enumerate the client keys for a dialog element

1. Call the `ENUM-CLIENT-KEYS` action, passing the handle of the dialog element for which the client keys should be enumerated, but omitting the key parameter. This has the effect of resetting the dialog element's enumeration cursor (i.e., position) back to the beginning of its internal key list. Since the enumeration cursor is initially reset when a dialog element is created, this step is strictly not required for the first key enumeration for a particular dialog element. However, it is good practice to explicitly reset the cursor in this manner, in order to make the enumeration context-insensitive.

2. Call the `ENUM-CLIENT-KEYS` action again, passing the handle of the dialog element and the key parameter, into which the first key (if any) will be returned.

3. If the key field was internally `RESET` to blanks by the above call, this indicates that no (more) keys remain, and the program should terminate the enumeration process.

4. Otherwise, go back to step 2 in order to retrieve the next key (if any).

Example:

```
 /* Enumerate and output all client keys in use by
control #LB-1: /* (1) Reset enumeration cursor: PROCESS GUI ACTION ENUM-CLIENT-KEYS
WITH #LB-1 GIVING *ERROR /* (2) Enumerate and delete the keys one-by-one: REPEAT
PROCESS GUI ACTION ENUM-CLIENT-KEYS WITH #LB-1 #LB-1.CLIENT-KEY GIVING *ERROR
IF #LB-1.CLIENT-KEY <> ' ' RESET #LB-1.CLIENT-VALUE /* delete the key END-IF WHILE
#LB-1.CLIENT-KEY <> ' ' END-REPEAT
```

This example illustrates that the `ENUM-CLIENT-KEYS` action is tolerant of keys being deleted during the enumeration process. If (as shown here) the last enumerated (i.e., "current") key is deleted, Natural automatically moves the internal enumeration cursor to its predecessor in th enumeration sequence, or resets it if there no predecessor. In either case, the next key returned by `ENUM-CLIENT-KEYS` is the one that would have been returned had the previous key not been deleted.

Note:
The sequence in which the keys are enumerated is implementation-dependent and is not guaranteed to remain the same in future Natural versions. Therefore, do not code your programs such that they are dependent on any particular enumeration sequence.