NS-DK

Version 5.00 Edition 1

Extended Templates and User Controls

Réf. n° DSN50002TPL00001

Contents

Chapter 3 Example of Developing an Extended Template

About this manual

This manual initiates you to the templates using and developing.

Organization of this manual

This manual contains the following chapters:

Chapter 1	***Template***
Chapter 2	***Extended templates and Custom Controls***
Chapter 3	***Example of developing an extended template***
	This chapter provides a simple example that illustrates the various phases involved in developing and using an extended template.

Conventions

Typographic conventions

Important term	Important terms are printed in **bold**.
Interface component	The names of windows, dialog boxes, controls, buttons, menus and options are printed in *italics*.
[F9]	Function key names appear in square brackets.
FILENAME	Filenames are printed in UPPERCASE.
`syntax example`	Syntax examples are printed in a `fixed-width font.`

Notational conventions

•	A round bullet is used for lists
♦	A diamond is used for alternatives
1.	Numbers are used to mark the steps in a procedure to be carried out in sequence

definition	A **definition** has a special presentation. It explains the term in a single paragraph. The term appears in the first column, then once in bold in the definition.

Operating conventions

Choose *XXX \ YYY*	This means you need to open the *XXX* menu, then choose the *YYY* command (option) from this menu. You can perform this action using the mouse or mnemonic characters on the keyboard.
Click the *XXX \ YYY* button	This means you need to display the tool bar named *XXX*, then click the *YYY* button in this tool bar (the name of each button is shown by its help bubble). You can only perform this action with the mouse.
Choose the *XXX* button	This means you need to choose the *XXX* button in a dialog box. You can perform this action using the mouse or mnemonic characters on the keyboard.

Icon codes

☞	**Comment,** note, etc.
👓	**Reference** to another part of the documentation
⚠	**Danger**: precaution to be taken, irreversible action, etc.
💡	**Suggestion**: helpful hints, etc.
⇗	**To go a step further**: level of detail or expertise greater than the average level of the document

	Indicates specific information on using the software under DOS-Windows (all versions)
32	Indicates specific information on using the software under DOS-Windows 32 bits
32 bits	Indicates specific information on using the software under DOS-Windows 32 bits
UNIX	Indicates specific information on using the software under Unix systems

Chapter 1 Templates

This chapter expains

- Definitions
- Designing a Template
- Using a Template
- Programming

Contents

Definitions

Template

A **template** is a user-defined NS-DK resource comprising a group of standard controls (Entry fields, Static text fields, etc.). Once it's been created, you can manipulate this resource like any other standard control.

Templates are used in different applications to standardize the interface and allow code to be reused. For example, whenever you use a set of standard controls whose appearance and behavior must comply with specific rules, it's preferable to group them together in a template. As a result, they'll no longer be manipulated individually but as a whole, and the associated code will be developed and tested only once.

Templates are generated in files with the .TPL extension.

Template control

Once inserted into a window, a template is treated and used like a standard control. It is then known as a **template control** and represents an instance of the template in the window.

When a template control is selected, all of its components are also selected, and they can no longer be accessed individually.

The behavior defined for a template can be "overridden" in a template control. This means that specific processing can be coded in the template control and, if required, combined with the standard processing defined in the template itself.

Link between a template and a template control

A link is established between a template and the corresponding template control inserted into a window.

In edit mode, this link is dynamic – any changes made to the template are immediately applied to the template controls based on this template.

In a generated application, this link is dynamic if the template was generated as a DLL. Any changes to the template in the DLL are automatically applied to the template control in the application without you having to regenerate the application.

However, this link is no longer dynamic if the template is included in the application's code. The application must be fully regenerated before the changes made to the template are applied to the template controls.

A template generated in DLL form is called an **extended template**. This type of template is covered in Chapter 3 of this manual.

Designing a template

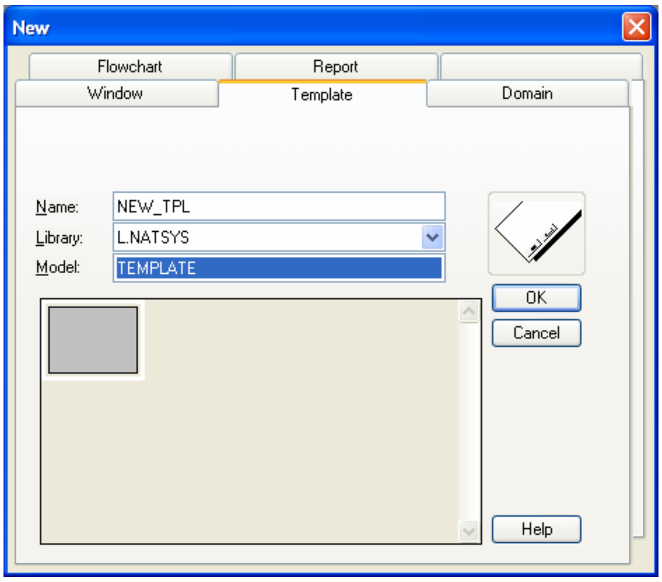
Creation

Templates are edited separately from windows.

To create a template

1. Choose *File \ New*. The *Open* box opens.

2. Select the *Templates* tab.

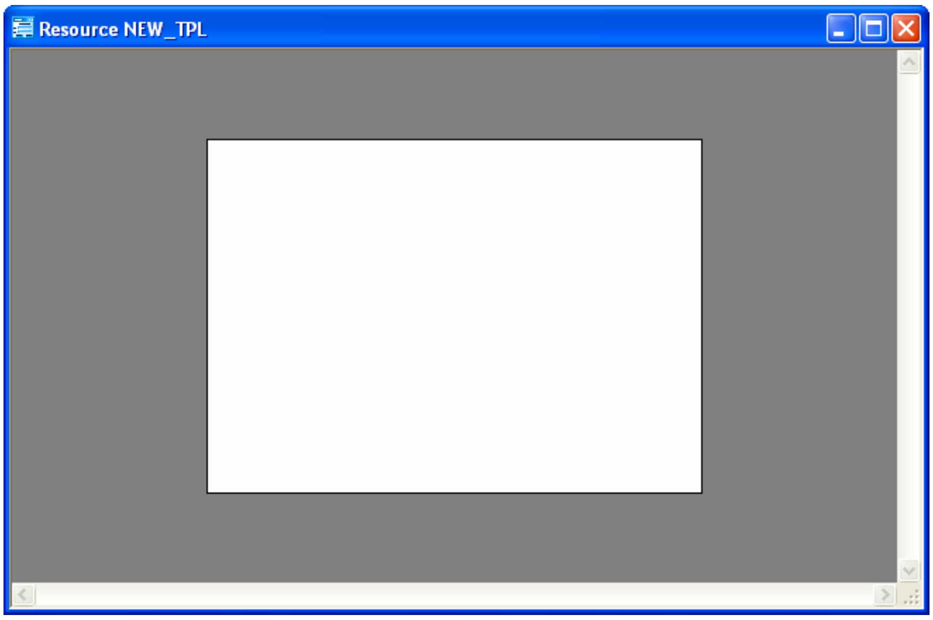


Name: NEW_TPL
Library: L.NATSYS
Model: TEMPLATE

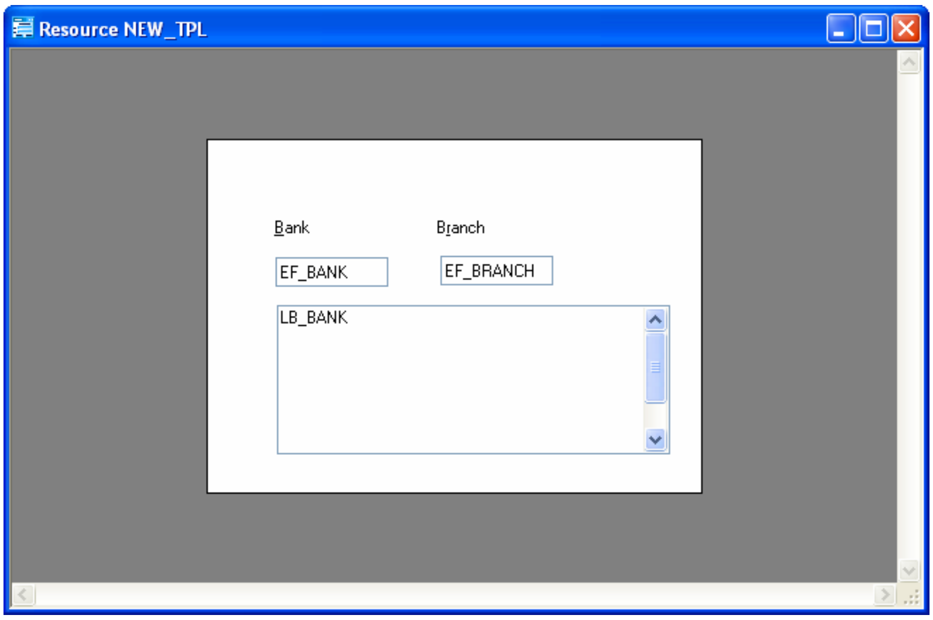
3. Enter the name of the template you want to create.

4. Choose *OK*.

5. The new template is now displayed. Resize it so that the desired controls will fit inside it.



6. Insert the standard controls used in the template. Controls are inserted in the same way as in a window based on the *Dialog* model.



7. Save the template by choosing *File \ Save*, or by pressing [F2].
Like a window, you can close the template by choosing *File \ Save*. After creating the template, you can add it to the *Custom* tool bar.

For details on how to add a template to the Custom tool bar, see the "User Manual", "Graphical Builder" part.

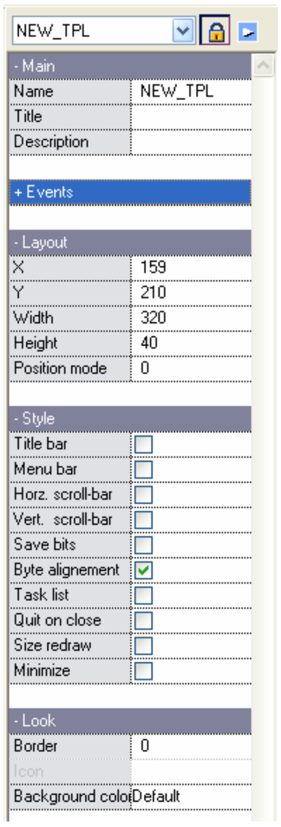
Configuring a template

Template property pane

To configure a template, use the property pane, which allows you to set the following parameters:

- Horizontal and vertical position of the template in the workspace.
- Width and height of the template.
- Description of the template.
- Focus sequence between a window and a template control based on this template.

Double-click with the right button of the mouse at the template, the property pane displays.



- Main	
Name	NEW_TPL
Title	
Description	
+ Events	
- Layout	
X	159
Y	210
Width	320
Height	40
Position mode	0
- Style	
Title bar	☐
Menu bar	☐
Horz. scroll-bar	☐
Vert. scroll-bar	☐
Save bits	☐
Byte alignement	✓
Task list	☐
Quit on close	☐
Size redraw	☐
Minimize	☐
- Look	
Border	0
Icon	
Background colo	Default

Parameters

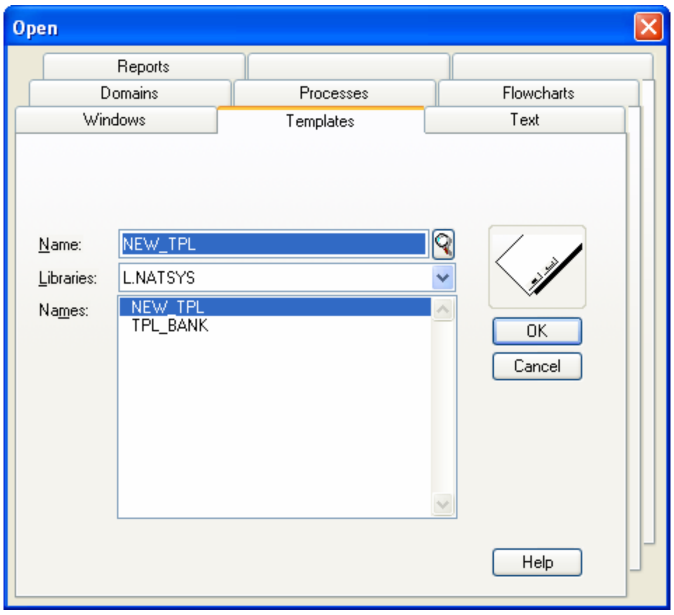
X, Y	Horizontal and vertical position of the template's bottom left-hand corner. These positions are expressed in pixels relative to the bottom left-hand corner of the workspace.Position
Width, Height	Width and height of the template in pixels.
Name	Name of the template. This field can't be modified in this dialog box.
Description	Comment associated with the template. This is intended for the developer's use only.
First Control	Name of the first control in the template that will get the focus after it moves to the template when the end-user presses [Tab]

Last Control	Name of the last control in the template that will have the focus before it leaves the template control when the end-user presses [Tab]. The focus will then be set on one of the fields in the dialog box that contains the template control.

Modifying a template

To modify a template

1. Choose *File \ Open*. The *Open window or template* box opens.



Name: NEW_TPL

Libraries: L.NATSYS

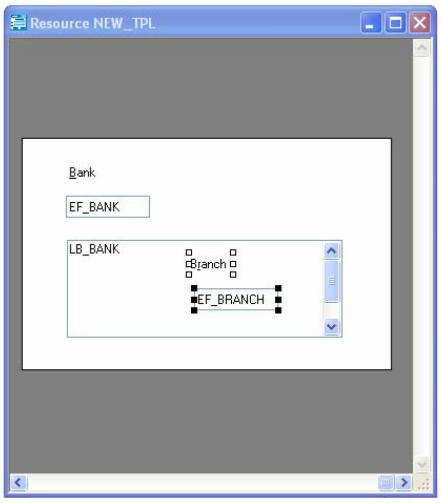
Names:
NEW_TPL
TPL_BANK

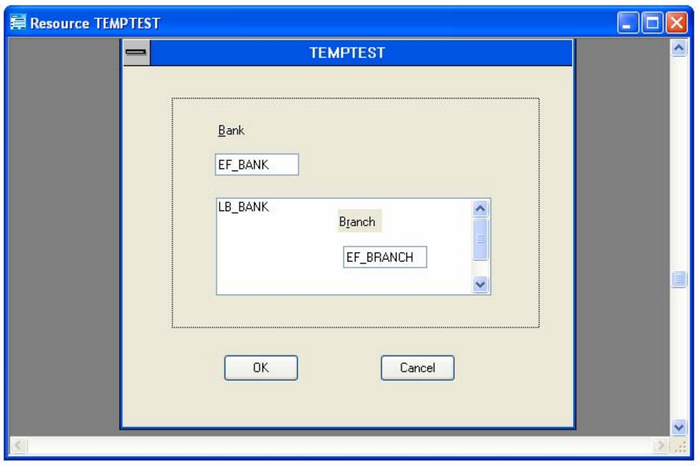
2. Select the template you want to modify.

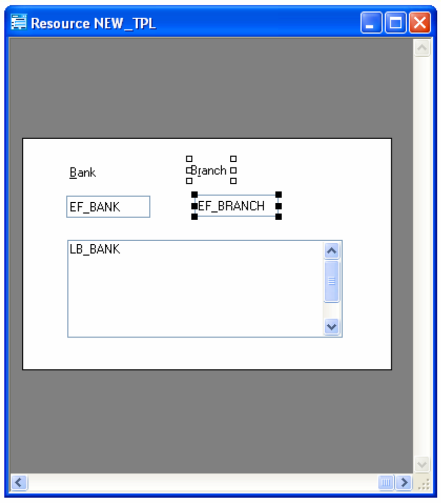
3. Choose *OK*. The template appears and can be modified.

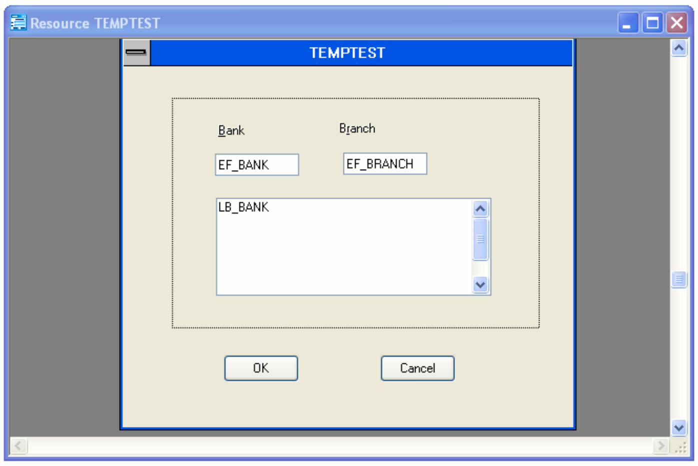
Dynamic changes to templates

A dynamic link is established between the template and any template controls based on this template. As a result, any changes made to the edited template are automatically applied to the template controls as these changes occur.
In the example below, repositioning the EF_BRANCH Entry field in the template automatically repositions this field in the template control included in a window.


Resource NEW_TPL
Bank
EF_BANK
LB_BANK
Branch
EF_BRANCH


Resource TEMPTEST
TEMPTEST
Bank
EF_BANK
LB_BANK
Branch
EF_BRANCH
OK
Cancel


Resource NEW_TPL
Bank
Branch
EF_BANK
EF_BRANCH
LB_BANK


Resource TEMPTEST
TEMPTEST
Bank
Branch
EF_BANK
EF_BRANCH
LB_BANK
OK
Cancel

Using a template

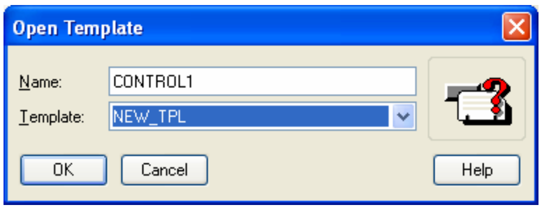
Inserting a template into a window

You can insert a template any window based on the *Dialog* model. You do this as you'd insert a standard control.

To insert a template into a window

1. You can either:
- Choose *Controls \ Standard \ Template*.
- Click the button that represents the template in the Controls tabbed window (if it includes this button), then move it into the window.

2. In both cases, the *Open template* box opens.



Name: CONTROL1
Template: NEW_TPL

3. In the *Name* field, enter the name of the template control that will be inserted into the window.

4. Select the name of the corresponding template from the *Template* field, then choose *OK* to confirm the insertion.

Template control

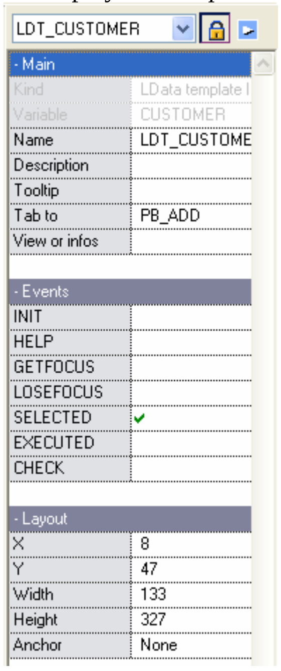
Once the template's inserted into a window, it is considered as a special control known as a template control.
In edit mode, when a template control is selected, all the controls in it are also selected. This global selection is indicated visually by a dotted box with the same size as the template used to generate the template control.
You can't access the controls in the template control individually. It behaves like a standard control with several items, such as a Check box control (a Check box and its associated text are inseparable when the control is selected or moved but are considered as separate objects when the control is managed internally).

Property pane

Like any other control, a template control has a property pane that allows you to set the following parameters:

- Name and position of the template control in the window.
- Description of the template control.
- Name of the control in the window that will get the focus after the template control loses the focus when the end-user presses [Tab].

To display the *Template_Name* property pane, double-click the template control.



LDT_CUSTOMER

- Main	
Kind	LData template l
Variable	CUSTOMER
Name	LDT_CUSTOME
Description	
Tooltip	
Tab to	PB_ADD
View or infos	
- Events	
INIT	
HELP	
GETFOCUS	
LOSEFOCUS	
SELECTED	✓
EXECUTED	
CHECK	
- Layout	
X	8
Y	47
Width	133
Height	327
Anchor	None

Parameters

X, Y	Horizontal and vertical position of the template control's bottom left-hand corner. These positions are expressed in pixels relative to the bottom left-hand corner of the window containing it.
Width, Height	Width and height of the template control in pixels. These are the width and height set for the original template and therefore can't be modified.
Name	Name of the template control.

Description	Comment associated with the template control. This is intended for the developer's use only.
Tab to	Name of the control in the window that will get the focus after the template control loses the focus when the end-user presses [Tab].

Conditions for inserting a template

To be able to insert a template into the window you're editing, the template must have been saved at least once. The template control will be saved in the window in the same state as the latest save of the template.

Setting the focus

As for dialog boxes, it's important to set the sequence of the focus inside the template when the end-user presses [Tab] successively. It's also important to define the first control and the last control in the template that will have the focus.
The focus sequence is defined on three different levels:

- Between the template and the "outside world".
- Between the controls in the template.
- Between the template control and the window that includes it.

Template

Use the *First Control* and *Last Control* fields in the template's property pane to indicate the first and last control in the template that will have the focus.
These two fields determine how the template generally interacts with the "outside world" in terms of the focus sequence.

Controls in the template

Set the focus sequence inside the template starting from the first control that will get the focus (defined in *First Control*) up to the last control (defined in *Last Control*).
Use the *Tab to* field in each control's property pane to specify the next control in the template that will get the focus when the end-user presses [Tab].

Template control

As for any other control in the window that contains the template control, use the *Tab* field in the template control's property pane to specify which control in the window will get the focus when the end-user presses [Tab].
Similarly, the template control's name should appear in the *Tab to* field of the property pane for one of the other controls in the window.

Focus sequence

After entering the above information, the focus sequence in each window that contains a template control will be as follows when the end-user presses [Tab] successively:

- Each of the controls in the window will receive the focus as indicated in their *Tab to* fields.
- The focus will move from one of the controls in the window to the template control. The focus will then be given to the control specified in the template's *First Control* field.
- Each of the controls in the template will receive the focus, as indicated in their *Tab to* fields, until the control defined as *Last Control* gets the focus.
- The focus will leave the template control. It will then be given to the control in the window specified by the *Tab to* field of the template control's property pane.

Logic for using a template

Black box

It is preferable for application developers who use templates to consider each template as a black box. This means that a template is identified by its interface only, not by its internal components.
This approach ensures that the template is totally independent from your applications: the template's internal components can be modified without affecting the application's code.
However, this also means that anyone who wants to use a template must be supplied with full documentation on the template's access interface and its behavior.
Details on the access interface should include at least the following:

- How to assign one or more values to the template (assignment operations, indexing, dynamic qualifiers, specific events to send to the template).
- How to retrieve values from the various fields in the template.

If events are used to interface with the template, these events must be specific to the template, i.e. their names should be different from the ones used for the standard behavior events generated automatically by various NCL functions or instructions.

Non-portability

Templates don't have to be supplied as black boxes: this is simply a method of ensuring that the templates you develop are portable.
In fact, NCL allows you to code a MOVE statement applied directly to a template's component if you know its name.
For example :

```
MOVE <value> TO TEMPLATE_CONTROL.INTERNAL_CONTROL
```

moves a value directly to the control named `INTERNAL_CONTROL` in the template control named `TEMPLATE_CONTROL`.
Although this method of coding is more straightforward, it can result in errors if the control is removed from the template or if its name is changed. By using an interface, you can extend or update the contents of a template without changing the way it's accessed.

Keyboard shortcuts

If your template contains Static text fields associated with other controls, we recommend that you assign keyboard shortcuts to them so that you comply with the standard design conventions for graphical interfaces.
Your choice of keyboard shortcuts should be made independently of any windows. Consequently, when you add a template to a window, some keyboard shortcuts may be duplicated, preventing direct access to certain fields using these shortcuts.
This type of problem leads to two constraints when developing templates:

- The documentation for each template's access interface should include the keyboard shortcuts used in the template (hence not recommended in a window).
- Avoid repeating the same keyboard shortcuts in different templates that may be used together in the same window.

Inserting a template into another template

You can insert a template into any template except itself.
All the operations described above for inserting a template into a window, setting the focus and complying with certain rules still apply when inserting a template into another template. The inserted template will be considered as a control that is part of the other template.

Programming

Behavior events

Definition

Behavior events	They allow you to define the behavior of a template. They are automatically proposed for each template and are activated directly by the NCL functions and instructions used to manipulate objects.

NCL instructions for manipulating objects

Template controls have no default behavior.
To enable a template control to respond to NCL instructions for manipulating controls (e.g. SETTEXT, SETPOS, HIDE, etc.), you need to process the corresponding behavior events. These events usually have the same name as the instruction (e.g. SETTEXT, SETPOS, HIDE, etc.). The resulting action depends entirely on the behavior you have defined for the template.

For example:

```
HIDE TEMPLATE_CONTROL
```

If this instruction is coded in an event for the window that contains the template, it won't hide all the fields in the template control. You will need to define the template's behavior when the HIDE event occurs. If the required result is to hide all the controls in the template, you should code a HIDE instruction for each control.

NCL functions for manipulating objects

As with instructions, to enable a template control to respond to the NCL functions for manipulating controls (e.g. ISDISABLED%, LINECOUNT%, etc.), you need to process the corresponding events (e.g. ISDISABLE, LINECOUNT, etc.) and terminate them with a RETURN instruction so that a value is passed back to the NCL function that triggered the event.
For example:
Suppose we have a template containing a List box control. The following NCL script has been coded in an event for the window that contains the template:

```
Local Nbr_Lines
Nbr_Lines = LINECOUNT% (TEMPLATE_CONTROL)
```

To retrieve the number of lines in the template control's List box, we need to code the following in the template's LINECOUNT event:

```
RETURN LINECOUNT% (LIST_BOX_CONTROL_IN_TEMPLATE)
```

If the template's LINECOUNT event is not coded, an unpredictable value will be returned by:

```
LINECOUNT% (TEMPLATE_CONTROL)
```

Correspondences between events and ncl functions/instructions

The correspondences between behavior events and NCL functions/instructions used to manipulate objects are described in the chapter “Events Reference” in the NCL Programming Manual.
These correspondences are shown by a table in which the PARAM1%, PARAM2% and PARAM2$ columns indicate the values returned in the event's parameters and the Return column indicates how to return a value to the function that triggered the event.
Note also that complementary actions such as HIDE/SHOW and LOCK/UNLOCK only generate one type of behavior event (HIDE and LOCK respectively) and that the actual action is indicated by the value in the PARAM1% parameter.

GETVALUE and SETVALUE events

GETVALUE and SETVALUE are the two most important events for a template.

GETVALUE event

This event is received when the contents of the template control are assigned to a variable or to another object. These contents depend on the template and are defined by its designer.
The GETVALUE event is generated by the MOVE instruction:

```
MOVE <TEMPLATE_CONTROL> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL[index]> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL.DynParam> TO <VARIABLE>
```

or any other evaluation instruction (IF, EVALUATE, etc.).

SETVALUE event

This event is received when a value is set for the template control. The way that this value is used depends on the template and the code in the SETVALUE event.
The SETVALUE event is generated by the MOVE instruction:

```
MOVE <value> TO <CONTROL_TEMPLATE>
MOVE <value> TO <CONTROL_TEMPLATE[index]>
MOVE <value> TO <CONTROL_TEMPLATE.Paramdyn>
```

Parameters for GETVALUE and SETVALUE

- PARAM1% contains:
  - DEFRET% if the event was generated by:
    ```
    MOVE <value> TO <TEMPLATE_CONTROL>
    MOVE <TEMPLATE_CONTROL> TO <VARIABLE>
    ```
  - The index specified in the instruction that generated the event:
    ```
    MOVE <value> TO <TEMPLATE_CONTROL[index]>
    MOVE <TEMPLATE_CONTROL[index]> TO <VARIABLE>
    ```
- PARAM2% is set to zero if the event was generated by:
  ```
  MOVE <value> TO <TEMPLATE_CONTROL>
  MOVE <value> TO <TEMPLATE_CONTROL[index]>
  MOVE <TEMPLATE_CONTROL> TO <VARIABLE>
  MOVE <TEMPLATE_CONTROL[index]> TO <VARIABLE>
  ```
  It contains a non-zero value if the event was generated by:
  ```
  MOVE <value> TO <TEMPLATE_CONTROL.DynParam>
  MOVE <TEMPLATE_CONTROL.DynParam> TO <VARIABLE>
  ```
  In this case, PARAM1$ will contain the dynamic parameter "DynParam."
- PARAM2$ contains a character string value to return or assign to the template control.

Overriding events

Principles

Any event intended for a template is first sent to the template control then to the template itself.
By default, a template control (which is an instance of a template in a window) behaves in the same way as the original template. This means that any events sent to the template simply pass through the template control and are not processed by it.
However, this default behavior can be modified by overriding events: in the template control, you simply code the event intended for the template. This processing will supplement or replace the processing coded in the template.

☞ An event that contains only a comment is still considered coded.
By using the PASS instruction in the script for the template control's event, you can trigger the template's event. This event can be triggered at any point in the template control's event according to the position of the PASS instruction in its script.
If the PASS instruction is not used in the template control's event, the processing coded for this event will replace the standard processing coded in the template.

Events for a template control

A template control has only 5 default events:

- INIT
- TERMINATE
- CHECK
- INITCONTEXT
- TERMINATECONTEXT.

Note that all the behavior events for a template can also be added to the standard events for a template control, even if they are not processed by the template. This means that their processing will only apply to the template control itself.

For example:
Suppose that the SETTEXT instruction has been applied to a template control named CTRLT from any script for the window (SCRN) that includes the template named TEMPL:

```
; Any event for an object in the window named SCRN
SETTEXT "Hello" TO CTRLT
```

The SETTEXT event is initially sent to the template control named CTRLT. After this, there are two possible scenarios:

- ♦ *If this event isn't coded, it's automatically sent to the TEMPL template and the script in its SETTEXT event is executed.*
- ♦ *If this event is coded, its script is executed, then:*
  - *The script in the SETTEXT event for TEMPL is only executed if PASS has been used in the script for CTRLT.*
  - *Otherwise, the script in the SETTEXT event for TEMPL isn't executed.*

Recap

- The events in the original template define its standard behavior.
- Events can be coded in a template control to modify the original template's behavior, i.e. override it.
- By overriding an event, you can modify the generic code for a template depending on the window that includes it.

Notifying the template control

Manipulating the template control in a window

Like any other control, a template control can be manipulated using its name.

For example:

```
MOVE TEMPLATECONTROL to variable$
```

or

```
HIDE TEMPLATECONTROL
```

Manipulating the template control from the original template

When you develop a template, you can't name the template control, since this is determined by developers when they insert it into a window. The template control must therefore be referenced by using the keyword SELF.
SELF identifies the current instance of the template in the window.
For example:
Suppose we have a template containing two Entry fields. How do we notify the template control that one of its fields has changed?
We need to code the following in the CHANGED event for each of its Entry fields:

```
SEND CHANGED to SELF
```

The event sent by this instruction follows the same path as an event sent to a template control by an assignment operation,the CHANGED event is sent to the template control.:

- ♦ *If this isn't coded, the CHANGED event is sent to the template.*
- ♦ *If it is coded, the associated processing is executed.*
- ♦ *If it is coded and the PASS instruction is used in the code, the event is also sent to the template.*

☞ To avoid infinite recursion, the SEND instruction should never send the event that's currently being processed in the template.

Example template programming

The example's purpose

In this example, we will create a template that will allow us to use the three different types of assignment operations. We will then include this template in a window and use it.

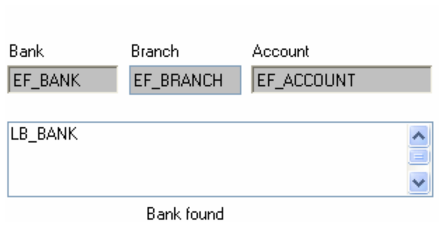
Description of the template

The template we're going to create will contain:

- A full bank account ID consisting of:
  - Bank code.
  - Branch code.

  - Account number.
- A List box that displays a list of banks (code and name).
- A Static text field that informs the user whether or not the bank was found in the List box.

The template will look like this.



Write functions

Our example should enable us to perform the following actions on the template:

- Split a bank account ID number into three fields:
  - Bank code
  - Branch code
  - Account number.
- Select the required bank from a List box.
- Insert a bank at a given position in the template's List box.
- Display a Horizontal Scroll bar in the bank list.

Read functions

Our example should enable us to obtain the following information from the template:

- Compose a full bank account ID number from the 3 fields: bank code, branch code and account number.
- Retrieve the name of a bank based on its position in the list.
- Find out whether the bank List box has a Horizontal Scroll bar.

Access interface

This template is designed as a black box. Its access interface is defined as follows:

- **Writing** to the template:

- To set the values for the bank code (5 digits), branch code (5 digits) and account number (11 digits), a full 21-digit account ID number must be assigned to the template control.

For example:
```
  MOVE "000011234510987654321" TO TEMPLATE_CONTROL
```
assigns: "00001" to the bank code
"12345" to the branch code
"10987654321" to the account number.
If the bank code exists in the bank list, the corresponding line will be selected, otherwise the message "bank not found" will be displayed.

- To add a bank to the list, the template must be treated like an array and a character string must be assigned to the template control.
  This string must contain both the bank code and bank name (separated by a comma). The template control must be indexed.

For example:
```
   MOVE "00001,XYZ" TO TEMPLATE_CONTROL[1]
```
inserts the bank "XYZ" with the code "00001" into line 1 of the bank list. If the specified index is set to 0, the bank will be inserted at the top of the list (after the title line).

- The dynamic parameter HORZSCROLLBAR is used to add or remove the Horizontal Scroll bar displayed in the bank list. To control this Scroll bar, the dynamic parameter HORZSCROLLBAR must be set to TRUE% or FALSE%.

For example:
```
   MOVE TRUE% TO TEMPLATE_CONTROL.HORZSCROLLBAR
```
adds a Horizontal Scroll bar to the bank list.

- **Reading** the template
  - A full 21-digit account ID number is obtained by moving the contents of the template control to a character string variable.

For example:
```
  MOVE TEMPLATE_CONTROL TO AccountNb$
```
moves: "000011234510987654321" to the AccountNb$ variable if the fields in the template contain the following values:

bank code:	*"00001"*
branch code:	*"12345"*
account number:	*"10987654321"*

  - To retrieve the bank name from a line in the bank list, simply assign the template control, indexed by the required line number, to a character string variable.

For example:
```
  MOVE TEMPLATE_CONTROL[2] TO BankName$
```
retrieves the bank on the second line in the bank list and places it in the variable BankName$. If the specified index is set to 0 (list's title line), this instruction will return the bank displayed on the first line.

- To check whether a Horizontal Scroll bar is displayed, we must read the template control's HORZSCOLLBAR dynamic parameter.

  For example:

  ```
    MOVE TEMPLATE_CONTROL.Horzscrollbar TO Exists%
  ```

  returns TRUE% or FALSE% in Exists% depending on whether or not the Scroll bar is displayed.

- List of **keyboard shortcuts** used:
  - [Alt + B] gives the focus to the bank code field (EF_BANK).
  - [Alt + R] gives the focus to the branch code field (EF_BRANCH).
  - [Alt + A] gives the focus to the account number field (EF_ACCOUNT).

Creating the example template

Create a template with the following characteristics:

```
Name TPL_BANK
Width 310
Height 155
```

Add the following controls to the template:

```
Type Entry Field
Name EF_BANK
X, Y 2, 92
Width 80
Parameters Force Fill, Auto-Tab,
           Length=5, Characters='0'..'9'
           Shadow=Dark, Background=Light gray
```

```
Type Static Text
Name ST_BANK
Text "Bank"
Link EF_BANK
X, Y 2, 117
Parameters Auto-Size
```

```
Type Entry Field
Name EF_BRANCH
X, Y 89, 92
Width 80
Parameters Force Fill, Auto-Tab,
           Length=5, Characters='0'..'9'
           Shadow=Dark, Background=Light gray
```

```
Type Static Text
Name ST_BRANCH
Text "Branch"
Link EF_BRANCH
X, Y 89, 117
Parameters Auto-Size
```

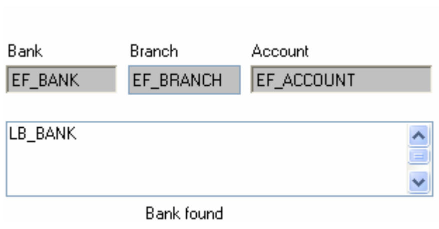
```
Type Entry Field
Name EF_ACCOUNT
X, Y 176, 92
Width 130
Parameters Force Fill, Auto-Tab,
           Length=5, Characters='0'..'9'
           Shadow=Dark, Background=Light gray
```

```
Type Static Text
Name ST_NUM
Texte "Account"
Link EF_ACCOUNT
X, Y 176, 117
Parameters Auto-Size
```

```
Type Static Text
Name ST_STATUS
Text "Bank found"
X, Y 100, 1
Parameters Auto-Size
```

```
Type List Box
Name LB_BANK
X, Y 2, 19
Width 304
File BANK_LST.TXT Preload
Tabulations 0, 100
Séparator ',' (virgule)
```

This will produce the following template on the screen.



Coding the template

➔ *GETVALUE event*

We will return the following information in PARAM2$, depending on how the event was called:

- A concatenation of the three fields that form the account ID number.
- The name of the bank whose index in the LB_BANK list was passed when the event was called.

- TRUE% or FALSE% to indicate whether or not a Horizontal Scroll bar is displayed in LB_BANK.

Therefore, we need to handle this event using 3 types of instructions:

```
MOVE <TEMPLATE_CONTROL> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL[index]> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL.DynParam> TO <VARIABLE>
```

The type of information returned will be determined by the contents of the PARAM1% and PARAM2% parameters passed to GETVALUE.

The following code must be entered:

```
;_________________________________________________________
;BankName$: String variable used to extract the bank name
;           from the selected line in the list box
;           LB_BANK
;I%:        Index used to select the lines from the List
;           Box LB_BANK
;_________________________________________________________
Local BankName$
Local I%


;Test how the GETVALUE event was generated:
; 1. MOVE <ctl_template> TO <variable>
;    then PARAM1% = DEFRET%
; 2. MOVE <ctl_template[index] TO <variable>
;    then PARAM2% = 0
;    and PARAM1% = index
; 3. MOVE <ctl_template.Paramdyn> TO <variable>
;    then PARAM2% <> 0
;    and PARAM1$ = "Paramdyn"
;
;    Return the corresponding value in PARAM2$
IF PARAM1% = DEFRET%

    ; Concatenate the 3 components of a bank account
    ; number
    PARAM2$ = EF_BANK & EF_BRANCH  & EF_ACCOUNT

ELSE
   IF PARAM2% <> 0
      IF PARAM1$ = "HORZSCROLLBAR"

      ; Return TRUE% or FALSE%
      PARAM2$ = LB_BANK.HORZSCROLLBAR

      ENDIF
   ELSE

      ; Select the indexed line from the List box
      ; LB_BANK then extract the bank name
      IF PARAM1% = 0
          I% = 1
      ELSE
          I% = PARAM1%
      ENDIF

      IF I% > LINECOUNT% (LB_BANK) - 1
         PARAM2$ = "Unknown Index"
      ELSE
         SELECT I% FROM LB_BANK
         BankName$ = LB_BANK
         BankName$ = COPY$(BankName$,              \
                           POS%(",",BankName$) + 1,\
                           Length (BankName$) -  \
                           POS%(",",BankName$))
         PARAM2$ = BankName$
      ENDIF

   ENDIF
ENDIF
```

SETVALUE event

We will perform the following operations, depending on how the event was called:

- Initialize the three fields that form the account number.
- Insert a bank name at a specified position in the LB_BANK list.
- Control the display of a Horizontal Scroll bar in the LB_BANK list.

Therefore, we need to handle this event using 3 types of instructions:

```
MOVE <value> TO <TEMPLATE_CONTROL>
MOVE <value> TO <TEMPLATE_CONTROL[index]>
MOVE <value> TO <TEMPLATE_CONTROL.DynParam>
```

The following code must be entered:

```
;________________________________________________________
;Param2Value$:Value of PARAM2$
;             PARAM2$ is initialized when the following
;             instructions are executed:
;             MOVE <value> TO <CONTROL_TEMPLATE>
;             MOVE <value> TO <CONTROL_TEMPLATE[index]>
;             MOVE <value> TO <CONTROL_TEMPLATE.ParamDyn>
;
;I%:          Index used to select the lines from the
;             List Box LB_BANK
;
;ListLine$:   Selected line in LB_BANK
;________________________________________________________
Local Param2Value$
Local I%
Local ListLine$


;Test how the SETVALUE event was generated:
; 1. MOVE <variable> TO <ctl_template>
;    then PARAM1% = DEFRET%
; 2. MOVE <variable> TO <ctl_template[index]
;    then PARAM2% = 0
;    and PARAM1% = index
; 3. MOVE <variable> TO <ctl_template.Paramdyn>
;    then PARAM2% <> 0
;    and PARAM1$ = "Paramdyn"
;
;    The corresponding value is in PARAM2$

; Save the value passed by the MOVE intruction
Param2Value$ = PARAM2$

IF PARAM1% = DEFRET%

       ; Initialize the components of the bank account
       ; number with the full account number passed by
       ; the MOVE instruction
       EF_BANK = Copy$ (Param2Value$,1,5)
       EF_BRANCH = Copy$ (Param2Value$,6,5)
       EF_ACCOUNT = Copy$ (Param2Value$,11,11)

       ; Select the corresponding bank in the list box,
       ; if exists.
       ; If the bank does not exist in the list, display
       ; "bank not found"
       I% = 0
       NOUPDATE LB_BANK
       While I% < Linecount%(LB_BANK)
        SELECT I% FROM LB_BANK
        ListLine$ = LB_BANK
        ListLine$ = Copy$(ListLine$,1, \
                          POS%(",",ListLine$) - 1)
        IF EF_BANK = ListLine$
          I% = Linecount%(LB_BANK) + 2
          ST_STATUS = "Bank found"
        ELSE
          I% = I% + 1
        ENDIF

       EndWhile
       UPDATE LB_BANK
       IF I% = Linecount%(LB_BANK)
          ST_STATUS = "Bank not found"
          UNSELECT I% - 1 FROM LB_BANK
       ENDIF

ELSE
   IF PARAM2% <> 0
      IF PARAM1$ = "HORZSCROLLBAR"
        ; Show or hide the horizontal scroll bar
        LB_BANK.HORZSCROLLBAR = Param2Value$
      ENDIF
   ELSE

      ; Insert a line into the list box
      IF PARAM1% = 0
          I% = 1
      ELSE
          I% = PARAM1%
      ENDIF
      INSERT AT I% Param2Value$ TO LB_BANK

   ENDIF
ENDIF
```

Save the TPL_BANK template by choosing *File \ Save* so that it can be used in a window.

Using the template

Now let's create a window based on the Dialog model. We will use it to manage the TPL_BANK template.
This window will carry out the read and write operations described above for the TPL_BANK template.

1. Create a window based on the Dialog model with the following characteristics:

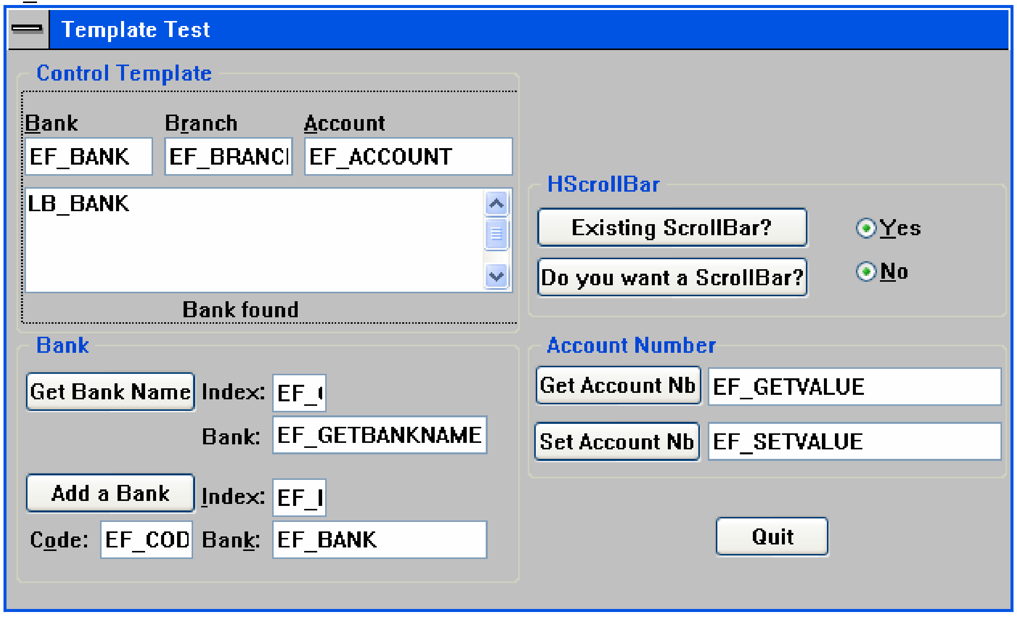
```
Name: D_MAIN
X, Y: 3, 90
Width , Height.: 634, 310
Title: Template Test
Parameters: Title Bar
```

2. Choose *Controls \ Load template* to insert a template control with the following characteristics:

```
Name: TPL1
Template: TPL_BANK
X, Y: 7, 125
```

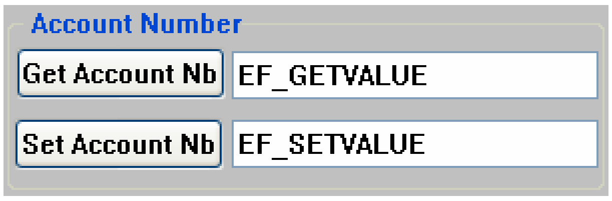
Configuring the window

We're now going to add and program the controls in the D_MAIN window used to read and write to the TPL1 template control.
D_MAIN now looks like this.



Reading and writing an account number

1. Create the part of the window used to read and write an account number (*Account Number* Group box).



2. Enter the following code for the Push button used to retrieve the account number from the template control (*Get Account Nb*):

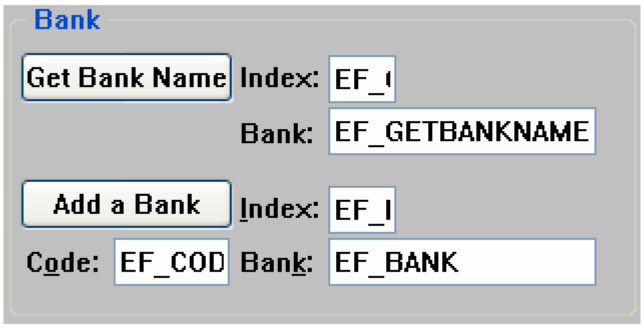
```
MOVE TPL1 TO EF_GETVALUE
```

3. Enter the following code for the Push button used to write the account number to the template control (*Set Account Nb*):

```
MOVE EF_SETVALUE TO TPL1
```

Reading and writing a bank name

1. Create the part of the window used to read a bank name and add a bank to the bank list (*Bank* Group box):



2. Enter the following code for the Push button used to extract a bank name (*Get Bank Name*):

```
MOVE TPL1[EF_GETINDEX] TO EF_GETBANKNAME
```

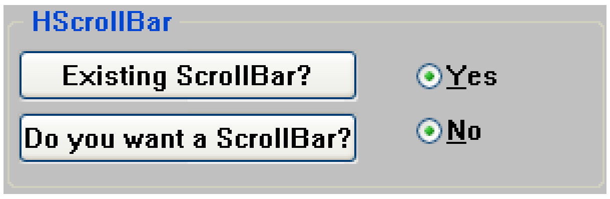
3. Enter the following code for the Push button used to add a bank to the bank list (*Add a Bank*):

```
MOVE EF_CODE &","& EF_BANK TO TPL1[EF_INDEX]
```

Controlling the horizontal scroll bar in the bank list

To control the Horizontal Scroll bar dynamically, we need to use the constants TRUE% (1) or FALSE% (0) in the dynamic parameter .HORZSCROLLBAR. Since we can't assign the value 0 to a Radio button, we'll increment these values by 1 before assigning them.
Then, when we need to assign the value of a Radio button to TPL1, we'll simply decrease it by 1 before assigning it.
Conversely, when we check whether a Horizontal Scroll bar is displayed in LB_BANK, we'll simply increment the value returned by TPL1 to determine which Radio button we need to select.
Do the following:

1. Create the part of the window used to control the Horizontal Scroll bar (*HScrollBar* Group box):



2. The two Radio buttons are used to add or remove the Scroll bar. Enter the following code for the Push button used to assign the value selected in the Radio buttons (*Do you want a ScrollBar?*):

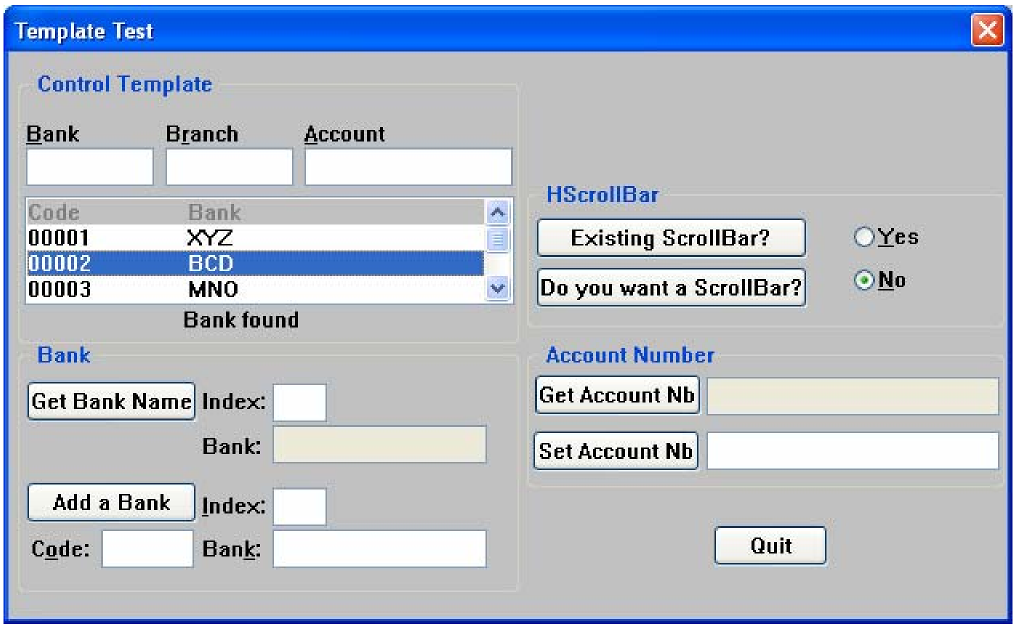
```
MOVE RB_YES - 1 TO TPL1.HORZSCROLLBAR
```

3. Enter the following code for the Push button used to find out whether the bank list contains a Scroll bar (*Existing ScrollBar?*):

```
MOVE TPL1.HORZSCROLLBAR + 1 TO RB_YES
```

Testing the application

To start the test, press the [F5] key or click the test start-up icon. The *Template Test* window opens.



Code	Bank
00001	XYZ
00002	BCD
00003	MNO

- Processing the account number:
  - Enter a bank code, branch code and account number in the *Control Template* group. Choose the *Get Account Nb* button in the *Account Number* group: this initializes the field next to this button with the full account ID number (21 digits).

    - Enter the full 21-digit account number in the field next to the *Set Account Nb* button, then choose this button to set the various fields in the template: bank code, branch code and account number (the bank code will be retrieved from the list).

- Processing the bank list:
    - Use the fields in the *Bank* group to enter the code and name of the bank you want to add, together with the index of the line where you want to insert it. Then, choose the *Add a Bank* button. A blank line is inserted into the bank list.
    - Use the *Index* field to enter the index of the line whose bank name you want to extract, then choose the *Get Bank Name* button. The bank name appears in the *Bank* field.

- Controlling the Horizontal Scroll bar in the bank list:
    - Choose the *Existing ScrollBar?* button in the *HScrollBar* group to check whether a Horizontal Scroll bar is displayed in the bank list. The corresponding Radio button (*Yes* or *No*) is selected.
    - Select the *Yes* (or *No*) Radio button to add (or remove) the Horizontal Scroll bar in the bank list, then choose the *Do you want a ScrollBar?* button.

Chapter 2 Extended templates and Custom controls

This chapter explains

- Introduction
- Using Extended Templates and Custom Controls
- Developing Extended Templates and Custom Controls

Contents

Introduction

Definitions

NS-DK provides a range of functions that allow you to create your own resources. Once you've developed these "custom resources," you manipulate them like any other standard NS-DK control.

There are two types of custom resources:

- Extended templates.
- Custom controls.

Extended templates

An **extended template** is a template that's stored in a DLL. Like any other template, it contains standard NS-DK controls, and can also include other templates.

Custom controls

A custom control is also stored in a DLL but, unlike an extended template, it doesn't use NS-DK controls. You must fully define its graphical appearance using the functions in the NS- Graph library, then program its behavior.

Once you've developed these two types of resources and compiled them in DLL form, you can use them in any NS-DK library.

These resources are inserted into a window as template controls. Therefore, by default, they'll behave like template controls.

However, in most cases, this type of resource is intended to behave in a specific way based on precise specifications. So, to design an extended template or custom control, you need to define:

- Its appearance and behavior in NS-DK (optional), which we'll refer to as Design mode.
- Its appearance and behavior during the execution of the application that uses it, which we'll refer to as Execution mode.

Dlls: recap and advantages

Recap

Dynamic Linking enables programs to use and share functions and memory resources outside their own .EXE file. These resources and functions are stored in DLLs (Dynamic Link Libraries).

Like an executable file, a DLL contains code and data. However, a DLL is not executed directly but called by executable files or other DLLs.

Advantages

There are two main reasons why we will generate a program in DLL form from our libraries:

- To enable the program's components to be reused by several programs.
- To restrict the amount of memory used by a program at a given time.

In addition, by isolating processes and resources in libraries, we can modify them without having to regenerate the programs that use these DLLs, unless of course we modify the parameters used to call the functions in these libraries.

Using libraries for extended templates and custom controls

Extended templates and custom controls are both generated in DLLs and thus benefit from the above advantages.

When an extended template or custom control is used in NS-DK, the way that the corresponding DLL is loaded and unloaded is transparent to the user. The DLL is loaded by system when it's used, and unloaded when it's no longer required.

☞ A library can contain several extended templates and/or custom controls.

Development principles

Each extended template or custom control is programmed in a NS-DK library, which must then be generated as a DLL.

Extended templates and custom controls are developed in two phases: the first phase defines its graphical appearance and behavior when it is edited (Design mode) and the second phase defines its graphical appearance and behavior while the application runs (Execution mode).

Design mode

The aim of this phase is to define the appearance of the control in the tool bar and in a window edited in NS-DK. You must also define the Info box that NS-DK will open to configure the control.

- You must define the control's appearance in the tool bar so that it's displayed correctly here, whether it's selected or not.

- The Info box is defined in a window based on the Dialog model. This window contains fields that allow the user to modify the coordinates and size of the control as well as its background color, for example. This information can then be used to define the control's appearance in a window edited in NS-DK.

This second phase is optional since, by default, NS-DK automatically assigns the characteristics of a template control.

Execution mode

- For an extended template, this phase involves the same steps as those used to create a standard template in Chapter 1 of this manual.

- For a custom control, this phases involves creating a window based on the USERCTRL model (*User Control*) and coding the events required for the control to function: drawing the window, handling the keyboard and mouse, etc.

To initialize the control using the parameters set in its Info box under NS-DK, you need to code the INITCONTEXT event for the template or *User Control* window during this phase.

Using extended templates and custom controls

Extended templates and custom controls are used in the same way.

After generating the library that contains the template, you need to:

1. Select the library in which you want to use this template.
2. Choose *Edit \ Modify*, then choose the *Uses*... button.
3. Select the library that contains the template, then associate it with the library that uses it.

☞ See Chapter 1 of this manual for details on how to use a template control.

Abbreviated template

Definition

Abbreviated Template

An abbreviated template is a special template whose sole purpose is to provide an entry point to an extended template or custom control.

This type of template isn't created with NS-DK. It's simply a text file containing a number of keywords and the name of the DLL that contains the extended template or custom control.

This file should have the same name as the extended template (.TPL resource generated as a DLL) or the custom control (.SCR resource based on the User Control model - USERCTRL - and located in the library generated as a DLL).

You must therefore create one file for each resource located in the same DLL.

This file must end with the .TPL extension.

Format of the .TPL file

FILE#	<number of file backups>
DEVICE	<screen width > <screen height > <system font width > <system font height > <window border width > <window border height > <border width for a Dialog window> <border height for a Dialog window> <border width> <border height> <title bar height> <menu height> <icon height> <icon width>
CONTROLTPL	< filename without the .TPL extension> X Y W H '<Comment>' NULL NULL EXTERNAL < DLL name without the .DLL extension>

where:

X, Y	stands for the horizontal and vertical position of the template's bottom left-hand corner. These positions are expressed in pixels relative to the bottom left-hand corner of the workspace.
W, H	stands for the width and height of the template, expressed in pixels.

☞ The name of the template resource specified after CONTROLTPL must be identical to the name of the .TPL file.

Sample contents of an abbreviated template :

The following file, CONTROL.TPL, provides access to the control named CONTROL situated in the DLL named TEST:

```
FILE# 1
DEVICE 640 480 6 16 4 4 4 4 1 1 19 19 32 32;
CONTROLTPL CONTROL 10 10 350 73 '' NULL NULL EXTERNAL 'TEST'
```

Developing extended templates and custom controls

This section describes the various aspects required to build an extended template or custom control:

1. Data being managed.
2. Resources used in this library.
3. Functions and instructions required for development.
4. Programming methods.

Data being managed

The data you will manage differs according to the mode you're developing.

There are 3 main types of data:

- System data for the manipulated control (name, position, size, etc.).
- Contextual data defined in NS-DK and used when the application runs (initial value, color, initial state, etc.).
- Run-time data.

System data

System data is data that's stored by the system and provides a basic description of the extended template or custom control.

Since applications will use the extended template or custom control via a template control, this data includes the name, description, tab sequence, position, width and height of the template control.

NS-DK maintains 6 system data items:

name	internal control name used by all NCL functions and instructions that reference the control.
description	comment used to describe the control. This is intended for the developer's use only.
tabulation	name of the control in the window that will get the focus when the template control loses the focus. This fields defaults to blank, which indicates that the next control in the window will get the focus.
position	horizontal and vertical position of the template control's bottom left-hand corner. These positions are expressed in pixels relative to the bottom left-hand corner of the window that contains the template control.
width and height	width and height of the template control (for a custom control, this is the control's client area, i.e. the area in which it can be drawn).

This system data is automatically managed and stored internally by NS-DK. It can be modified by the user via the template control's Info box.

If you use the default Info box displayed (i.e. the Info box for a template control), the contextual data will be updated automatically by NS-DK. Otherwise, if you define your own Info box, the function <Name>__CHANGEPARM described below will enable you to display this data to the user in this box and save it.

Contextual data

This data describes the control's initial state: initial value, colors, status (hidden or displayed, locked or unlocked, disabled or enabled, etc.).

These options are defined in Design mode via the template control's Info box; they are then used at run-time.

♦ If you define your own Info box for your extended template or custom control, you need to:

  a) Store this data in Design mode.

  b) Retrieve this data in Execution mode.

♦ If you use the default Info box displayed (i.e. the one for a template), the user of your extended template or custom control can only modify its system data. In this case, the data is fully managed by NS-DK.

Storing contextual data in design mode

When the control is displayed in an open window, NS-DK stores its contextual data in a segment which is loaded into memory. We will call this the context segment.

The contents of this segment include the control's background color, initial value, etc.

When this window is saved, NS-DK stores this data in a character string, which we'll call the context string.

Using the data in execution mode

Contextual data is defined in Design mode before being used in Execution mode.

In Execution mode, this data can be retrieved when the control is initialized by reading the PARAM1$ parameter in the control's INITCONTEXT event.

PARAM1$ will contain the context string saved by NS-DK in the .SCR file.

☞ The INITCONTEXT event is described later on in this chapter.

Run-time data

Each extended template or custom control can have its own data used to manage it internally. We will call this its run-time data.

It may be advantageous to store this data in the context string together with the contextual data.

- In Design mode: the context segment is used automatically by NS-DK to maintain the contextual data.
- In Execution mode: you use the same segment to manage both contextual data and run-time data.

Thus, the context segment makes it easier to manage contextual data and run-time data when programming the Execution mode for the extended template or custom control.

This segment can contain:

- ♦ Contextual data only.
- ♦ Run-time data only.
- ♦ Both of the above.

Developers are fully responsible for managing the run-time data. Above all, they should ensure that this data is specific to each occurrence of the control in the application.

You can manage the context segment by:

- Defining the fields in the context segment (contextual data and run-time data).
- Allocating this segment in the INITCONTEXT event for the template (extended template) or User Control window (custom control).
- Storing the segment's address.
- Initializing the segment's contextual data using the context string in PARAM1$.
- Initializing the segment's run-time data.
- Deallocating the context segment once it's no longer required (TERMINATECONTEXT event described later on in this chapter).

Functions for managing the new property pane defined by an XML file

The NSDK/NatStar 5.00 design tool lets you manage the properties of all the controls from the pane located on the right of the tool.

In the case of user controls, some adaptation is necessary in order to access them:

- firstly functions need to be created that allow the value of each of the properties of a control to be retrieved and changed.

- then an XML format template file needs to be created, describing all the properties of the control.

The advantage of this new functionality is that it means you no longer have to develop a specific property box for each user control.

A reminder concerning user controls

Three types of data need to be managed for each user control:

- system data (name, description, tabulation, position, width, height, and from version 5, tooltips and anchoring)
- context sensitive data (managing the initial state of the control, such as its value, its colors, etc.).
- execution data (by definition not managed in Design).

The property pane, therefore, lets you change system data and context sensitive data.

Context sensitive data is stored in a segment loaded into memory called the context segment.

Access to the properties of the context segment is enabled via read/write functions (Getter/Setter), the syntax of which is set out below.

<Name>_Get_<PtyName> Function

Lets you read a template property.

Syntaxe	**<Name>_Get_<PtyName>** (*Context*, *retVal$*, *ext*)			
Parameters	*Context*	SegContext	I/O	segment variable describing the execution context of the template control. SegContext is the segment defined by the developer.
	retVal$	CSTRING	I/O	a character string containing the return value.
	ext	POINTER	I/O	pointer to a segment of the SEG_ExtendedProp type.
Valeur retournée	Constante TRUE% ou FALSE% : INT(1)			

Comments

1. The SEG_ExtendedProp type extension segment lets you manage special cases.

```
SEGMENT SEG_ExtendedProp
      INT      action(1) ; determines the behaviour of the function
      CONTROL  ctrl
      INTEGER  indexCount
      POINTER  callback
      POINTER  reserved
      INTEGER@ Indexs[]
ENDSEGMENT ; SEG_ExtendedProp
```

2. Prior to displaying the value in the property pane, a call to the function is made in order to find out if the property can currently be changed. If it can, the function returns TRUE%, otherwise FALSE%. If it is FALSE%, the property is disabled in the property pane.

3. If the properties cannot be changed, the **action** field of the SEG_ExtendedProp segment is equal to PEA_IsDisabled%, and the code follows the following structure:

```
IF @ext <> 0
  IF ext.action = PEA_IsDisabled%
             ; Determination of status
             IF Propriete_Non_Modifiable%
                     RETURN TRUE%
             ELSE
                    RETURN FALSE%
             ENDIF
  ENDIF
       RETURN FALSE% ; other types of action
ENDIF
```

4. In a situation where the value of a property can be chosen from the different values in a combo-box, and where these values are not fixed but determined during the Design (cf the isDynamicallyFillable option in the XML file), the function is then responsible for filling the combo-box, which is given by the CONTROL type **ctrl** field of the extension segment. In this case the **action** field corresponds to PEA_GetValues%, and you have a style code:

```
IF @ext
IF ext.action = PEA_GetValues%
             ; Filling the combo
             FILL_COMBO ext.ctrl
             RETURN TRUE%
       ENDIF
       RETURN FALSE% ; autres types d'actions
ENDIF
```

Exemple

```
FUNCTION BANK_GET_HORZSCROLLBAR(seg_bank_context@ context, \
                                @retVal$, \
                                SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
             RETURN FALSE%
      ENDIF

      retVal$ = context.horzscrollbar%

      RETURN TRUE%
ENDFUNCTION ; BANK_GET_HORZSCROLLBAR
```

Voir aussi <Name>_Set_<PtyName>

<Name>_Set_<PtyName> Function

Lets you change a template property.

Syntaxe **<Name>_Set_<PtyName>** (*Context*, *val$*, *ext*)

Paramètres

Context	SegContext	I/O	variable segment décrivant le contexte d'exécution du contrôle template. SegContext est le segment défini par le développeur.
val$	CSTRING	I/O	chaîne de caractères qui contient la valeur en entrée
ext	POINTER	I/O	pointeur sur un segment de type SEG_ExtendedProp

Valeur retournée INT(1)

Comments

1. The values returned by the function can be:

- PSR_BadAction%, if the extension segment action cannot be processed.
- PSR_BadValue%, if the input value is invalid.
- PSR_SameValue%, if the input value is equal to the current value.
- PSR_OK%, if the value is actually reassigned.

2. Changing the value of a property may lead to a change in the value or modification permissions of one or more other properties (we speak of dependent properties). It is also the responsibility of the <Name>_Set_<PtyName> function of the current property to manage these situations. In addition, you must specify the dependencies in the XML file (see the section concerning the Property tag) so that all the properties affected are redisplayed.

3. The SEG_ExtendedProp type extension segment lets you manage special cases.

```
SEGMENT SEG_ExtendedProp
      INT      action(1) ; détermine le comportement de la fonction
      CONTROL  ctrl
      INTEGER  indexCount
      POINTER  callback
      POINTER  reserved
      INTEGER@ Indexs[]
ENDSEGMENT ; SEG_ExtendedProp
```

4. In the case of a complex property, the value of which is difficult to manage in a simple control, it is possible to start specific processing in order to update the property, such as opening a specific window (see theButton and ExtDlgSetter type editors in the XML file).

For example, this is used to open the "Font selection" window for a Font property. In this case the action is PEA_SetValues%, and you have a style code:

```
IF @ext <> 0
       IF ext.action <> PEA_SetValues%
             RETURN PSR_BadAction%
       ENDIF
       ; traitement spécifique
       CHOOSEFONT%(FONT.NAME, FONT.SIZE, FONT.SELS)
       RETURN PSR_OK%
ENDIF
```

Exemple

```
FUNCTION BANK_SET_HORZSCROLLBAR(seg_bank_context@ context, \
                                @val$, \
                                SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
             RETURN PSR_BADACTION% ; invalid action
      ENDIF

      IF NOT IsInt%(val$)
             RETURN PSR_BADVALUE% ; invalid value
      ENDIF

      IF context.horzscrollbar% = INT val$
             RETURN PSR_SameValue%  ; no update necessary
      ENDIF

      context.horzscrollbar% = INT val$

      RETURN PSR_OK% ; update successful
```

```
ENDFUNCTION ; BANK_SET_HORZSCROLLBAR
```

Voir aussi <Name>_Get_<PtyName>

Control modeling file

The control modeling file is an XML format file that contains all the information required to display the properties of the control in the property pane.

It must have the same name as the .TPL file and be located in the same directory.

☞ This new functionality means that you no longer need to develop a specific properties box for each template and user control.

File structure

The file starts with the following prolog.

```
<?xml version="1.0" encoding="utf-8" ?>
```

The user control is then described in an **Entity** tag.

The **Entity** tag contains a list of categories (**Category** tag), each category containing a list of properties (**Property** tag).

Syntax of the Entity tag

```
<Entity xmlns:NS="http://tempuri.org/Control.xsd" id="NomTemplate Info"
visibleName="NomTemplate" stdProperties="true" dllName="NomDLL.dll">
      …
</Entity>
```

The attributes of the **Entity** tag are defined as follows:

- the value of the attribute **id** is a string chosen by the developer.
- the value of the attribute **visibleName** must necessarily be the name of the template.
- the value of **dllName** must be the name of the DLL that contains the functions <Name>_Get_<PtyName> and <Name>_Set_<PtyName> of the control.
- the attribute **stdProperties** must be TRUE in order for you to be able to manage system data.

Syntax of the Category tag

```
      <Category stdName="TitreCategorie">
             …
      </Category>
```

The only attribute that needs specifying is **stdName** which corresponds to the name displayed by the category.

Syntax of the Property tag

There are two major types of properties:

- standard properties, therefore the descriptions have already been defined.
- properties specific to the control.

In the case of a standard property, the only attribute is **stdName** which equates to the name of the property already defined.

```
      <Property stdName="NOMPROPRIETE"/>
```

In the case of user controls, the system properties are standard properties (see the Description of the system data).

Specific properties have the following structure:

```
<Property id="numero_propriete" text="Texte_Propriete">
      <Description>Texte_de_description_de_la_propriete</Description>
      <Editor kind="Type_Editeur"/>
      <Callback>
             <Get>NOM_FUNCTION_GETTER</Get>
             <Set>NOM_FUNCTION_SETTER</Set>
      </Callback>
      <DependantProperty id="numero_propriete_dependante1" />
      <DependantProperty id="numero_propriete_dependante2" />
</Property>
```

The **Property** tag has two attributes:

- **id**, which is the number of the property, each property needing to have a different number in the XML file.

- **text**, which is the title of the property (displayed in the left-hand column of the pane).

Syntax of the Description tag

The **Description** tag contains a string describing the property.

Syntax of the Editor tag

The **Editor** tag describes the type of editing control (entry-field, combo-box...) used to specify the value of the property.

Text type

This type of editor lets you manage character strings. It is represented by an Entry-Field.

Its definition can be completed using **Param** tags.

In a **Param** tag, the attribute **name** lets you specify one of the following types:

- CHARACTERS, in order to specify the characters accepted (all by default).
- MAXLEN, a value up to 255, in order to specify the number of characters accepted (no limit by default).
- REQUIRED, value 0 or 1, in order to specify that a value is required.
- UPCASE, value 0 or 1, so that all characters will automatically be in upper case.

```
<Editor kind="Text">
<Params>
<Param name="CHARACTERS">'A'..'Z','a'..'z','0'..'9','_'</Param>
<Param name="MAXLEN">31</Param>
<Param name="REQUIRED">1</Param>
      <Param name="UPCASE">1</Param>
</Params>
</Editor>
```

Spin type

This type of editor lets you manage integers. It is represented by an Entry-Field in Spin-button mode (that is to say two arrows in order to increase or decrease the value).

Its definition can be completed using **Param** tags.

In a **Param** tag, the attribute **name** lets you specify one of the following types:

- CHARACTERS, in order to specify the characters accepted (all by default).
- MAXLEN, a value up to 255, in order to specify the number of characters accepted (no limit by default).
- REQUIRED, value 0 or 1, in order to specify that a value is required.
- TEXT, containing a set of three integers specifying the default value, the minimum value and the maximum value of the spin.

```
<Editor kind="Spin">
      <Params>
             <Param name="CHARACTERS">'0'..'9'</Param
             <Param name="MAXLEN">5</Param>
             <Param name="TEXT">0 0 10000</Param>
      </Params>
```

```
</Editor>
```

Bool type

This type of editor lets you manage Boolean properties. It is represented by a check box.

```
<Editor kind="Bool" />
```

Enum and CustomEnum types

The Enum editor type is represented by a combo-box.

CustomEnum is represented by a combo-box with an Entry-Field in order to specify a value other than the choices offered.

Two ways of filling in the combo-box are possible: either the values are specified in the XML file, or they are retrieved dynamically by the Design.

In the first case, all the values are specified in the **Values** tag, which contains a set of **Value** tags.

A **Value** tag contains two tags:

- the **Memory** tag which contains the real value handled by the functions <Name>_Get_<PtyName> and <Name>_Set_<PtyName>.
- the optional **Label** tag which contains the text to be displayed in the pane, enabling revamping of the value handled.

```
<Editor kind="Enum">
      <Values>
             <Value><Memory>0</Memory><Label>Default</Label></Value>
             <Value><Memory>1</Memory><Label>Integer</Label></Value>
             …
      </Values>
</Editor>
```

In the second case, you must use the **isDynamicallyFillable** attribute and give it the value TRUE. This attribute is optional and set to FALSE by default.

```
<Editor kind="Enum" isDynamicallyFillable="true">
```

Colors type

This type of editor lets you manage color properties. It is represented by a combo-box containing the set of colors available.

```
<Editor kind="Colors" />
```

FontNames type

This type of editor lets you manage the properties of character sets. It is represented by a combo-box containing the set of fonts available.

```
<Editor kind="FontNames">
```

Button type

This type of editor permits particular processing in the function <Name>_Set_<PtyName>, such as opening specific windows.

```
<Editor kind="Button" />
```

ExtDlgSetter type

This editor is composed of an Entry-Field on the left and a button on the right. The behavior of the button is similar to that of the Button type and the behavior of the Entry-Field is that of the Text type.

```
<Editor kind="ExtDlgSetter" />
```

ResourceEdit type

This editor is used for file names. Its representation is similar to ExtDlgSetter, but the button always opens a file explorer system window.

```
<Editor kind="ResourceEdit"/>
```

Syntax of the Editor tag

The **Callback** tag encompasses:

- the **Get** tag which contains the name of the function required to read the property (<Name>_Get_<PtyName>).
- the **Set** tag which contains the name of the function required to write the property (<Name>_Set_<PtyName>).

Syntax of DependantProperty tags

DependantProperty tags let you specify the properties whose display must also be refreshed when the current property is changed (because the <Name>_Set_<PtyName> function also changes the values or change permissions of other properties).

Description of the system data

Specifically, system data is defined in the template file by inserting three standard categories, MAIN, LAYOUT, and EVENTS.

MAIN contains general data relating to the control: its type (MODEL), its name (NAME), its optional description (DESCRIPTION), the next control when the tab key is pressed (TAB), and its TOOLTIP.

```
<Category stdName="MAIN">
       <Property stdName="MODEL"/>
       <Property stdName="NAME"/>
       <Property stdName="DESCRIPTION"/>
       <Property stdName="TOOLTIP"/>
       <Property stdName="TAB"/>
</Category>
```

LAYOUT contains data relating to the size (WIDTH, HEIGHT) and position (X,Y, ANCHORING) of the control.

```
<Category stdName="LAYOUT">
       <Property stdName="X"/>
       <Property stdName="Y"/>
       <Property stdName="WIDTH"/>
       <Property stdName="HEIGHT"/>
       <Property stdName="ANCHOR"/>
</Category>
```

EVENTS is a special category containing the set of events belonging to the control. All are managed automatically by the tool.

```
<Category stdName="EVENTS"/>
```

Appearance and behavior

In Design mode, each extended template and custom control is inserted into a window as a template control. In most cases, the default appearance and behavior for a template control is not adequate for these resources. Therefore, for Design mode, you need to define how the resource will be displayed in the window and tool bar, and how to interface with the resource.

To fully define this appearance and behavior under NS-DK, you need to code 12 functions and instructions. NS-DK automatically calls these functions and instructions and sets the values of their parameters.

These functions and instructions are optional. If one of them is missing, NS-DK will automatically trigger default actions which correspond to the behavior of a template control in Design mode.

These 12 functions and instructions must be prefixed by the name of the extended template or custom control:

- ♦ Name of the template (.TPL).
- ♦ Name of the User Control window (.SCR).

They must be coded in a library.

☞ You can also call these functions and instructions in the code used to program the Execution mode for your extended template or custom control.

You can also call these functions and instructions in the code used to program the Execution mode for your extended template or custom control.

These functions and instructions are divided into six categories covering the following areas:

- Managing the context.
- Managing the template control's size.
- Displaying the extended template or custom control in Design mode,
- Changing the contextual data.
- Managing the template control's events.
- Initializing the extended template or custom control in Design mode.

Code the relevant functions and instructions for your requirements.

Managing the context

<Name>__SAVECONTEXT	Saves the context string as a character string in the window that contains the extended template or template control.
<Name>__LOADCONTEXT	Loads the context segment into memory when the window that contains the extended template or custom control is displayed in the NS-DK workspace.
<Name>__GETSIZECONTEXT	Returns the size of the context segment.

Managing the template control's size

<Name>__CHANGEWIDTH	
<Name>__CHANGEHEIGHT	Indicate whether the control can be resized using the mouse and its property pane.
<Name>__MINWIDTH	

<Name>__MINHEIGHT	Specify the control's minimum size in Design mode.

Displaying the extended template or custom control in design mode

<Name>__PAINT	Describes how the extended template or custom control is displayed in the window or tool bar.

Changing the contextual data

<Name>__CHANGEPARM	Calls the property pane defined by the developer.

Managing the template control's events

<Name>__GETDEFAULTEVENTS	Specifies the events that will be handled by the extended template or custom control (i.e. the events that can be programmed by the user of the extended template or custom control).

Initializing the extended template or custom control in design mode

<Name>__INIT	Initializes or allocates the resources used by the extended template or custom control (files, bitmaps, etc.)
<Name>__TERMINATE	Deallocates any resources allocated by the function <Name>__INIT.

Under Windows NT, the names of these functions must be coded in uppercase.

Functions and instructions used to control the object's appearance and behavior in Design mode

This section describes each function and instruction that allows you to fully control the appearance and behavior of your extended templates and custom controls in Design mode.

Each one is accompanied by the following information:

- Description

  Expected result of the function or instruction.

- Syntax

  This is illustrated by the NCL definition that you must enter in your library.

- Parameters

  Type and meaning of each one.

- Return value

For a function, type of the value to return.

- Notes, which can include the following details:

  - How to code the function or instruction

    Make sure you follow the advice in this section, otherwise the resource you're creating may behave abnormally.

  - Default action

    Describes the default action taken. If this is suitable, you won't need to code the function or instruction.

  - How the function or instruction is called by NS-DK

    Describes the conditions under which NS-DK calls the function or instruction.

- See also

  Names of any related functions or instructions.

See Chapters 3 and 4 in this manual for examples of how to code these functions and instructions.

⚠ When coding, do not code a PASS to the GETFOCUS, LOSEFOCUS, or HELP event of a Custom control: for technical reasons, this will have no effect whatsoever.

<Name>__SAVECONTEXT Instruction

Saves the contextual data defined in the template control's Info box. This data will be stored as a character string in the window that contains the extended template or custom control.

Syntax **<Name>__SAVECONTEXT** *StrContext, Context, X, Y, W, H*

Parameters

StrContext	STRING	I/O	character string used to store the contents of the context segment (context string)
Context	SegContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.
X, Y	INTEGER	I	position of the template control in the window.
W, H	INTEGER	I	size of the template control

Notes

1. The contextual data is loaded into memory and manipulated by NS-DK in a segment defined by the developer (context segment). This segment is stored as a character string (context string).

2. Warning: the length of this string can't exceed 255 characters.

3. This instruction allows you to retrieve the contextual data defined in the template control's Info box each time you open the window.

4. You must use this instruction to save the contents of the context string in a character string.

5. The data is saved by concatenating the various segment fields into the string and inserting separators. The choice of separators should make it easy to extract each data item when the string is retrieved.

6. If this instruction isn't coded, NS-DK will save an empty string ('') in the window (.SCR file).

7. This instruction is called by NS-DK:

- When the template control is moved within the window.
- When the window is saved.

See also <Name>__LOADCONTEXT and <Name>__GETSIZECONTEXT.

<Name>__LOADCONTEXT Instruction

Loads the context segment into memory and initializes it using the context string saved in the window that contains the extended template or custom control.

Syntax **<Name>__LOADCONTEXT** *StrContext, Context*

Parameters

StrContext	CSTRING	I/O	character string containing the contextual data (context string).
Context	SegContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.

Notes

1. This instruction allows you to retrieve the contextual data defined in the template control's Info box each time you open the window.
2. The developer should use this instruction to retrieve the information in the context string before storing it in the segment.
3. This information is retrieved based on the separators inserted into the string when it was saved by <Name>__SAVECONTEXT.
4. If this instruction is not coded, NS-DK doesn't load anything into memory.
5. This instruction is called by NS-DK:
   - When the window that contains the template control is opened.
   - When the template control is added to a window.

See also <Name>__SAVECONTEXT and <Name>__GETSIZECONTEXT.

<Name>__GETSIZECONTEXT Function

Returns the size of the context segment.

Syntax	**<Name>____GETSIZECONTEXT**	
Return value	Type	INTEGER

Notes

1. This size enables NS-DK to allocate the segment in memory. Therefore, if this number is incorrect, memory may be overwritten.
2. The developer must code the following in this function:

   Return SizeOf <context_segment_name>
3. No default action (no memory allocation).
4. This function is called by NS-DK:
   - When the Custom tabbed window is displayed, if the resource has previously been added to the tool bar.
   - When a window that contains the resource is opened.
   - When the resource is added to a window.

See also <Name>__SAVECONTEXT and <Name>__LOADCONTEXT.

<Name>__CHANGE* Functions

Return TRUE% if the template control can be resized with the mouse or *Info* dialog-box under NS-DK; otherwise return FALSE%.

When you use xxx__changewidth or xxx__changeheight for an extended template you should not be able to change template size neither with mouse nor with *Info* box.

Syntax	**<Name>__CHANGEWIDTH** (*Context*)			
	<Name>__CHANGEHEIGHT (*Context*)			
Parameters	*Context*	SegContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.
Return value	TRUE% or FALSE% constant: INT(1).			

Notes

1. The context segment variable passed as a parameter can be used to:
   - Update an indicator when these dimensions are changed.
   - Test certain conditions to determine this function's return value.
2. The developer should return TRUE% or FALSE%.
3. By default, the template control is resizable under NS-DK (TRUE%).
4. These two functions are called by NS-DK when the mouse moves over the template control in the workspace. This enables NS-DK to determine whether the mouse pointer should change shape to allow the template control to be resized (double arrow cursor).
5. Neither of these two functions are called in the template control's Info box. If the Width and Height fields are available in this box, it is up to the developer to disable them if required. For more details on this point, see Chapters 3 and 4 in this manual.

<Name>__MIN* Functions

Return the minimum width and height (in pixels) of a template control if it is resizable.

Syntax	**<Name>__MINWIDTH**		(*Context*)	
	<Name>__MINHEIGHT		(*Context*)	
Parameters	*Context*	SegContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.
Return value	Type	INTEGER.		

Notes

1. The context segment variable passed as a parameter can be used to:
   - Update an indicator when these dimensions are changed.
   - Test certain conditions to determine this function's return value.
2. The developer must return a size in pixels.
3. By default, these functions return 16 pixels.
4. These two functions are called by NS-DK when the developer:
   - Moves the control within the window using the mouse or keyboard.
   - Modifies the size of the template control in the window using the mouse.
5. Neither of these two functions are called in the template control's Info box. If the Width and Height fields are available in this box, it's up to the developer to disable them if required. For more details on this point, see Chapters 3 and 4 in this manual.

<Name>__PAINT Function

Used to define the appearance of the extended template or custom control when it's displayed in the window and/or tool bar.

Syntax **<Name>__PAINT** (*NoPaint, PS, X, Y, W, H, Wnd, BackCol, Context*)

Parameters

NoPaint	INT(1)	I	display mode in NS-DK: 0: in a window in Design mode 1: in the tool bar, released button 2: in the tool bar, pressed button 3: in the tool bar, disabled button.
PS	INT(4)	I	handle of the presentation space (where the template control will be painted).
X, Y	INTEGER	I	position of the template control in the window.
W, H	INTEGER	I	size of the template control.
Wnd	POINTER	I	handle of the window that contains the template control.
BackCol	INTEGER	I	background color of the window that contains the template control.
Context	SegmentContext		I segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.

Return value INT(1)

value	**meaning**
TRUE%	display after drawing the resource using the functions in the NS-GRAPH library
FALSE%	display default drawing

Notes

1. If the resource is an extended template:

   - It will be drawn in the window as it was designed when it was defined (.TPL).

   - It will be represented in the tool bar by its name (.TPL name).

   You are strongly advised to return FALSE% when displaying the resource in the window (NoPaint = 0).

2. If the resource is a custom control:

   - It will be represented in the window by a semi-rectangle.

   - It will be represented in the tool bar by its name.

3. This function is called:

   - When the Custom tabbed window is displayed (if the template has been added to it).

   - When the template control is displayed in a window, i.e. when it is added to a window, or when a window that contains it is opened.

<Name>__CHANGEPARM Function

Opens the Info box used to set the template control's parameters and returns TRUE% if the control has been modified, otherwise FALSE%.

Syntax **<Name>__CHANGEPARM** (*X, Y, W, H, NameId, TabId, Description, AddrCheck, AddrGetTab, AddrOpenDlgEvents(4), Context*)

Parameters

X, Y	INTEGER	I/O	position of the template control in the window.
W, H	INTEGER	I/O	size of the template control.
NameId	CSTRING	I/O	name of the template control in the window.
TabId	CSTRING	I/O	name of the control in the window that will receive the focus when the template control loses the focus.
Description	CSTRING	I/O	comment associated with the template control. This is intended for the developer's use only.
AddrCheck	INT(4)	I/O	address of the function used to check the template control's system data.
AddrGetTab	INT(4)	I/O	address of the function used to extract the controls in the window that can get the focus after the template control.
AddrOpenDlgEvents	INT(4)	I	address of the function that displays the window used to code the template control's events (Events box).
Context	SegmentContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.

Return value TRUE% or FALSE% constant: INT(1).

Notes

1. In this function, the developer must code a modal call to the Info box that he/she has defined (Dialog window).

2. This Info box is used to display and modify:

   - System data.
   - Contextual data.

   The developer must therefore:

   - Pass this data as a parameter when the Info box is opened.
   - Save this data in the context segment together with the system data passed by address to this function.

3. The three addresses passed as parameters to <Name>__CHANGEPARM are used by some of the functions in NSCUSTOM.NCL:

   - CheckXYWHNameTab%
   - Gettab$
   - OpenDlgEvents.

   Since these functions are called by the Info box, these addresses must passed to this box.

4. Development overview for this function:

   - Allocate an information segment containing:
   - The system data.
   - The three procedure addresses passed as parameters to <Name>__CHANGEPARM.
     - Initialize the fields in this segment using the system data received in this function's parameters.
     - Open the Info box modally and pass the following as parameters:
       - Address of the information segment.
       - Address of the context segment.

    - (CALL ... USING).

- Save the fields in the information segment together with the system data passed in parameters.

5. NS-DK will display the Info box for a template control, which can only maintain system data.

6. This function is called by NS-DK when you display the Info box for the template control.

<Name>__GETDEFAULTEVENTS Instruction

Used to define the events which will be displayed in the template control's Events box, and which can be programmed by the end user.

Syntax **<Name>__GETDEFAULTEVENTS** *Context, EventsSet*

Parameters

Context	SegmentContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.
EventsSet	TEVENTSSET	I/O	segment variable used to save the events displayed in the template control's Events box. TEVENTSSET is a segment defined in NSCUSTOM.NCL.

Notes

1. The developer must use this instruction to specify the events displayed in the template control's Events box.
2. The NSCUSTOM.NCL library provides 4 functions and instructions for managing events: CLEAREVENTSSET, ADDEVENT, SUBEVENT and ISINEVENTSET%.
3. A template control has five default events:
   - INIT
   - TERMINATE
   - CHECK
   - INITCONTEXT
   - TERMINATECONTEXT.
4. This instruction is called by NS-DK when you display the window used to code the template control's events.

<Name>__INIT Function

Enables the user to allocate and initialize resources (bitmaps, text files, etc.) when the window that contains the template control is opened and before creating the template control in this window.

Syntax **<Name>__INIT** (*StrContext, Context, Wnd*)

Parameters

StrContext	CSTRING	I	context string.
Context	SegmentContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.
Wnd	POINTER	I	handle of the window that displays the template control.

Return value INT(4)

value	**meaning**
FALSE%	the template control will not be created (if it doesn't already exist in the window); if the template control already exists in the window, it will be ignored as though the DLL didn't exist.
TRUE%	the template control will be created (if it doesn't already exist in the window); the template control in the window will be displayed; NS-DK won't load anything into memory if this instruction isn't coded.

Notes

1. You are advised not to use this function to initialize the context segment - use the instruction <Name>__LOADCONTEXT instead.
2. Developers can allocate and initialize resources used by their extended template or custom control (bitmaps, text files, etc.).

3. This instruction is called by NS-DK:

   - When the window that contains the template control is opened.
   - When the template control is added to a window.

See also <Name>__TERMINATE.

<Name>__TERMINATE Instruction

Frees from memory any resources allocated by the <Name>__INIT function.

Syntax **<Name>__TERMINATE** *Context*

Parameters	*Context*	SegmentContext	I/O	segment variable used to describe the template control's run-time context. SegContext is the segment defined by the developer.

Notes

1. You should use this instruction to deallocate any resources allocated by the <Name>__INIT function.
2. No default action (no memory deallocated).
3. This instruction is called by NS-DK:
   - When the template control is deleted from the window.
   - When the window that contains the control is closed.

See also <Name>__INIT.

Events for managing the context

Two additional events enable you to manage your resource's context in Execution mode: INITCONTEXT and TERMINATECONTEXT.

The INITCONTEXT event allows you to initialize your extended template or custom control using the contextual data entered in its Info box. This event is received before the window's INIT event.

The TERMINATECONTEXT event is received after the window's TERMINATE event. It allows you to deallocate any resources allocated by the INITCONTEXT event.

INITCONTEXT Event

This event is generated in Execution mode when the extended template or custom control is loaded into the window that contains it.

Parameters	*PARAM1%*	Least significant part of the context string's address (received in PARAM1$). This context string will be used to initialize the control.
	PARAM2%	Most significant part of the context string's address.
	PARAM1$	Contains the context string, i.e. the extended template or custom control's data stored in a character string whose address is in PARAM12%.
	PARAM3%	Width of the extended template or custom control if it is resizable.
	PARAM4%	Height of the extended template or custom control if it is resizable.

Notes

1. Events are received in the following order:
   - INITCONTEXT event for the extended template or custom control.
   - INIT event for the window that contains the extended template or custom control.
   - INIT event for the template control in the window (i.e. the template control that enables the extended template or custom control to be used).
   - INIT event for the extended template or custom control if PASS is used in the template control's INIT event or if this event is not coded.
2. PARAM1% and PARAM2% contain a 2-byte integer.
3. PARAM12% is a programming feature that enables you to obtain a 4-byte integer by combining PARAM1% and PARAM2%. Therefore, PARAM12% will contain the context string's address and is equivalent to:

```
MOVE DWORD% ( PARAM2%, PARAM1%) TO variable
```

4. The context string is created in Design mode by the instruction <Name>__SAVECONTEXT, which concatenates the contextual data into the string.

Objects Extended templates and custom controls.

See also PARAMSTR$, EXECPROG

TERMINATECONTEXT Event

This event is generated in Execution mode when an extended template or custom control is loaded into the window that contains it.

Parameters	*PARAM1%*	Not used
	PARAM2%	Not used
	PARAM3%	Not used
	PARAM4%	Not used

Notes

1. Events are received in the following order:

   - TERMINATE event for the template control in the window (i.e. the template control that enables the extended template or custom control to be used).
   - TERMINATE event for the extended template or custom control if PASS is used in the template control's INIT event or if this event is not coded.
   - TERMINATE event for the window that contains the extended template or custom control (.SCR).
   - TERMINATECONTEXT event for the extended template or custom control.

Objects Extended templates and custom controls.

See also PARAMSTR$, EXECPROG

NSCUSTOM.NCL

This library contains a programming interface that extends the NCL language.

It allows you to:

- Manage the events for the template control that contains the extended template or custom control (add, delete, etc.). These are the events that can be programmed by the user of the extended template or custom control

- Check the template control's system data.

Therefore, this library is mainly used by:

- The 12 functions and instructions used to operate the extended template or custom control in Design mode (in particular: <Name>__PAINT, <Name>__CHANGEPARM and <Name>__GETDEFAULTEVENTS).

- The Info box displayed for the extended template or custom control.

ADDEVENT Instruction

Deletes the default events before adding the event required.

Syntax	**ADDEVENT**	*events-seg-var,event-const*		
Parameters	*events-seg-var*	TEVENTSSET	I	segment variable used to save the events added to the template control. TEVENTSSET is described below.
	event-const	INT(1)	I	event to add (one of the EVENT_* constants described below).

Notes

1. This instruction saves the event passed as a parameter in the segment variable.
2. This instruction is generally used by <Name>__GETDEFAULTEVENTS, which tells NS-DK which events will be handled by the extended template or custom control. These are the events which are proposed in the template control's Events box, and can be programmed by the user of the template control.

CHECKXYWHNAMETAB% Function

Checks the validity of the system data items passed as parameters:

- the name must be syntactically correct,
- the template control must not be larger than the window that contains it,
- the template control must not be smaller than its minimum size,
- the field that will get the focus after the template control must exist in the window.

Syntax **CHECKXYWHNAMETAB%** (*addr-check-proc, pos-x, pos-y, width, height, control-name, tab, min-width, min-height, display-error*)

Parameters	*addr-check-proc*	POINTER	I	address of the function used to check the template control's system data.
	pos- x	INTEGER	I	
	pos- y	INTEGER	I	horizontal and vertical position of the template control's bottom left-hand corner. These positions are expressed in pixels relative to the bottom left-hand corner of the workspace.
	control-name	CSTRING	I	name of the template control in the window.
	tab	CSTRING	I	name of the next control in the window that will get the focus when the template control loses the focus.
	width	INTEGER	I	
	height	INTEGER	I	size of the template control, expressed in pixels.
	display-error	INT(1)	I	indicates whether a message should be displayed if an error is detected.

Return value INT(4)

value	**meaning**
0	no error
1	the pos-x parameter is incorrect
2	the pos-y parameter is incorrect
3	the width parameter is incorrect
4	the height parameter is incorrect
5	the control-name parameter is incorrect
6	the tab parameter is incorrect

Notes

1. In the event of an error, a message will only be displayed if display-error is set to TRUE%.

2. This function is generally called by the extended template or custom control's Info box to check its system data.

CLEAREVENTSSET Instruction

Deletes all the events previously added to the events list and reinitializes it with the default events, i.e INIT, TERMINATE, CHECK, INITCONTEXT, TERMINATECONTEXT.

Syntax **CLEAREVENTSSET** *events-seg-var*

Parameters	*events-seg-var*	TEVENTSSET	I	variable segment used to save the events added to the template control. TEVENTSSET is described below.

Notes

1. This variable stores the events that will be proposed in the template control's Events box.
2. This instruction is generally used by <Name>__GETDEFAULTEVENTS, which tells NS-DK which events will be handled by the extended template or custom control. These are the events which are proposed in the template control's Events box and can be programmed by the user of the template control.

Exemple

```
;the l'instruction
SWITCH__GETDEFAULTEVENTS
AddEvent EventsSet,EVENT_CHANGED%
AddEvent EventsSet,EVENT_EXECUTED%

; Reinitialize the eventset segment
; if the eventset segment is empty while insertion of the custom control
; the default events are proposed :
;INIT, TERMINATE,CHECK, INITCONTEXT, ;TERMINATECONTEXT

CLEAREVENTSSET EventsSet
```

EVENT_*% Constants

Events handled by NS-DK.

Syntax

EVENT_ACTIVATE%
EVENT_ENDMENU%

EVENT_INITCONTEXT%

EVENT_TERMINATECONTEXT%

EVENT_LAST%

EVENT_DDEINITIATE%

EVENT_DDEINITIATEACK%

EVENT_DDETERMINATE%

EVENT_DDEADVISE%

EVENT_DDEUNADVISE%

EVENT_DDEREQUEST%

EVENT_DDEDATA%

EVENT_DDEPOKE%

EVENT_DDEEXECUTE%

EVENT_NULL%

EVENT_INIT%

EVENT_TERMINATE%

EVENT_HELP%

EVENT_CHAR%

EVENT_MOUSEMOVE%

EVENT_BUTTONDOWN%

EVENT_BUTTONUP%

EVENT_BUTTONDBLCLK%

EVENT_TIMER%

EVENT_PAINT%

EVENT_CHANGED%

EVENT_VSCROLL%

EVENT_HSCROLL%

EVENT_GETFOCUS%

EVENT_LOSEFOCUS%

EVENT_SELECTED%

EVENT_EXECUTED%

EVENT_CHECK%

EVENT_GETVALUE%

EVENT_SETVALUE%

EVENT_GETTEXT%

EVENT_SETTEXT%

EVENT_GETXPOS%

EVENT_GETYPOS%

EVENT_GETWIDTH%

EVENT_GETHEIGHT%

EVENT_SETPOS%

EVENT_SELECTION%

EVENT_ISDISABLED%

EVENT_ISLOCKED%

EVENT_ISHIDDEN%

EVENT_ISSELECTED%

EVENT_LINECOUNT%

EVENT_SETFOCUS%

EVENT_DISABLE%

EVENT_LOCK%

EVENT_HIDE%

EVENT_SELECT%

EVENT_UPDATE%

EVENT_SETFORECOL%

EVENT_SETBACKCOL%

EVENT_GETFORECOL%

EVENT_GETBACKCOL%

EVENT_SETRANGE%

EVENT_LOAD%

EVENT_SAVE%

EVENT_INSERT%

EVENT_DELETE%

EVENT_DELETEALL%

EVENT_PRIVUSER0%

EVENT_PRIVUSER1%

EVENT_PRIVUSER2%

EVENT_PRIVUSER3%

EVENT_PRIVUSER4%

EVENT_PRIVUSER5%

EVENT_PRIVUSER6%

EVENT_PRIVUSER7%

EVENT_PRIVUSER8%

EVENT_PRIVUSER9%

EVENT_PRIVUSER10%

EVENT_PRIVUSER11%

EVENT_PRIVUSER12%

EVENT_PRIVUSER13%

EVENT_PRIVUSER14%

EVENT_PRIVUSER15%

EVENT_USER0%

EVENT_USER1%

EVENT_USER2%

EVENT_USER3%

EVENT_USER4%

EVENT_USER5%

EVENT_USER6%

EVENT_USER7%

EVENT_USER8%

EVENT_USER9%

EVENT_USER10%

EVENT_USER11%

EVENT_USER12%

EVENT_USER13%

EVENT_USER14%

EVENT_USER15%

Notes

1. These constants are used with the ADDEVENT and SUBEVENT instructions and the ISINEVENTSET% function.

2. These instructions and this function are generally called by the user-defined function <Name>__GETDEFAULTEVENTS (described above) which tells NS-DK which events will be proposed in the template control's Events box.

3. They are declared internally as follows:

```
CONST EVENT_NULL%            0
CONST EVENT_INIT%            1
CONST EVENT_TERMINATE%       2
CONST EVENT_HELP%            3
CONST EVENT_CHAR%            4
CONST EVENT_MOUSEMOVE%       5
CONST EVENT_BUTTONDOWN%      6
CONST EVENT_BUTTONUP%        7
CONST EVENT_BUTTONDBLCLK%    8
CONST EVENT_TIMER%           9
CONST EVENT_PAINT%           10
CONST EVENT_CHANGED%         11
CONST EVENT_VSCROLL%         12
CONST EVENT_HSCROLL%         13
CONST EVENT_GETFOCUS%          14
CONST EVENT_LOSEFOCUS%          15
CONST EVENT_SELECTED%          16
CONST EVENT_EXECUTED%          17
CONST EVENT_CHECK%             18
CONST EVENT_GETVALUE%          19
CONST EVENT_SETVALUE%          20
CONST EVENT_GETTEXT%           21
CONST EVENT_SETTEXT%           22
CONST EVENT_GETXPOS%           23
CONST EVENT_GETYPOS%           24
CONST EVENT_GETWIDTH%          25
CONST EVENT_GETHEIGHT%          26
CONST EVENT_SETPOS%            27
CONST EVENT_SELECTION%          28
CONST EVENT_ISDISABLED%         29
CONST EVENT_ISLOCKED%          30
CONST EVENT_ISHIDDEN%          31
CONST EVENT_ISSELECTED%         32
CONST EVENT_LINECOUNT%          33
CONST EVENT_SETFOCUS%          34
CONST EVENT_DISABLE%           35
CONST EVENT_LOCK%              36
CONST EVENT_HIDE%              37
CONST EVENT_SELECT%            38
CONST EVENT_UPDATE%            39
CONST EVENT_SETFORECOL%         40
CONST EVENT_SETBACKCOL%         41
CONST EVENT_GETFORECOL%         42
CONST EVENT_GETBACKCOL%         43
CONST EVENT_SETRANGE%          44
CONST EVENT_LOAD%              45
CONST EVENT_SAVE%               46
CONST EVENT_INSERT%             47
CONST EVENT_DELETE%             48
CONST EVENT_DELETEALL%          49
CONST EVENT_DDEINITIATE%        50
CONST EVENT_DDEINITIATEACK%     51
CONST EVENT_DDETERMINATE%       52
CONST EVENT_DDEADVISE%          53
CONST EVENT_DDEUNADVISE%        54
CONST EVENT_DDEREQUEST%         55
CONST EVENT_DDEDATA%            56
CONST EVENT_DDEPOKE%            57
CONST EVENT_DDEEXECUTE%         58
CONST EVENT_PRIVUSER0%          59
CONST EVENT_PRIVUSER1%          60
CONST EVENT_PRIVUSER2%          61
CONST EVENT_PRIVUSER3%          62
CONST EVENT_PRIVUSER4%          63
CONST EVENT_PRIVUSER5%          64
CONST EVENT_PRIVUSER6%          65
CONST EVENT_PRIVUSER7%          66
CONST EVENT_PRIVUSER8%          67
CONST EVENT_PRIVUSER9%          68
CONST EVENT_PRIVUSER10%         69
CONST EVENT_PRIVUSER11%         70
CONST EVENT_PRIVUSER12%         71
CONST EVENT_PRIVUSER13%         72
CONST EVENT_PRIVUSER14%         73
CONST EVENT_PRIVUSER15%         74
CONST EVENT_USER0%              75
CONST EVENT_USER1%              76
CONST EVENT_USER2%              77
```

CONST EVENT_USER3%	78
CONST EVENT_USER4%	79
CONST EVENT_USER5%	80
CONST EVENT_USER6%	81
CONST EVENT_USER7%	82
CONST EVENT_USER8%	83
CONST EVENT_USER9%	84
CONST EVENT_USER10%	85
CONST EVENT_USER11%	86
CONST EVENT_USER12%	87
CONST EVENT_USER13%	88
CONST EVENT_USER14%	89
CONST EVENT_USER15%	90
CONST EVENT_ACTIVATE%	91
CONST EVENT_ENDMENU%	92
CONST EVENT_INITCONTEXT%	93
CONST EVENT_TERMINATECONTEXT%	94
CONST EVENT_LAST%	94

GETTAB$ Function

Returns a character string containing the name of a control in the window. Once all the controls have been returned, end-of-list is set to zero.

Syntax **GETTAB$** (*addr-tab-list-proc*, *end-of-list*)

Parameters

addr-tab-list-proc	POINTER	I	address of the function used to extract the names of the controls in the window that can get the focus after the template control.
end-of-list	INT(4)	I/O	indicates that the function has finished extracting the window's controls.

Notes

1. This instruction is particularly useful for filling a list (such as List box or Combo box) with the different fields in the window that can get the focus after the template control.
2. This instruction is generally called by the extended template or custom control's Info box to initialize the Tab field in this box.
3. Once all the controls in the window have been returned, end-of-list is set to 0

Return value INT(4)

Example

```
; INIT event for the customized Info box
; associated with the extended template or
; custom control:
; Into a Combo Box Entry Field named CBE_TAB, insert
; the various fields in the window that can get the
; focus after the template control.
local ptr%

; Initialize the field that indicates the end of
; the extraction of the fields in the window
move 0 to ptr%

; Loop to extract the fields in the window and
; insert them into the Combo Box Entry Field
Insert Ascending GetTab$( info.AddrGetTab, ptr%) to CBE_TAB
While ptr% <> 0
   Insert Ascending GetTab$( info.AddrGetTab, ptr%) to CBE_TAB
EndWhile
```

ISINEVENTSSET% Function

Checks whether the specified event appears in the events list segment passed as a parameter.

Syntax **ISINEVENTSSET%** (*events-seg-var, event-const*)

Parameters

events-seg-var	EVENTSSET	I	segment variable used to save the events added to the template control. TEVENTSSET is described below.
event-const	INT(1)	I	event to check (one of the EVENT_* constants described below).

Return value INT(4)

value	**meaning**
TRUE%	event found
FALSE%	event not found

Notes

1. This function is generally used by <Name>__GETDEFAULTEVENTS, which tells NS-DK which events will be handled by the extended template or custom control. These are the events which are proposed in the template control's Events box and can be programmed by the user of the template control.

OPENDLGEVENTS Instruction

Opens the template control's Events window, used to code the extended template or custom control's programmable events.

Syntax **OPENDLGEVENTS** *addr-event-proc, parent-window-handle*

Parameters

addr-event-proc	POINTER	I	address of the function that displays the window used to code the template control's events (Events box).
parent-window-handle	POINTER	I	handle of the Events box's parent window (usually SELF%).

Notes

1. This instruction is generally called by the Info box for the extended template or custom control and activated by the push button used to open the Events box for programmable events.

SUBEVENT Instruction

Removes an event from the list of events proposed in the template control's Events box (deletes the event passed as a parameter from the events list).

Syntax **SUBEVENT** *event-seg-var*, *event-const*

Parameters

event-seg-var	TEVENTSSET	I	segment variable used to save the events added to the template control. TEVENTSSET is described below.
event-const	INT(1)	I	event to delete (one of the EVENT_* constants described below).

Notes

1. This instruction is generally used by <Name>__GETDEFAULTEVENTS. It tells NS-DK which events will be handled by the extended template or custom control. These are the events which are proposed in the template control's Events box, and can be programmed by the user of the template control.

Example

```
; the instruction
SWITCH__GETDEFAULTEVENTS

;if EXECUTED event isn’t there, add it
if NOT ISINEVENTSSET%(EventsSet, EVENT_EXECUTED%)
             AddEvent EventsSet,EVENT_EXECUTED%
endif

; if EXECUTED event is present, delete it and add CHANGED event
if ISINEVENTSSET%(EventsSet, EVENT_EXECUTED%)
      SUBEvent EventsSet,EVENT_EXECUTED%
      AddEvent EventsSet,EVENT_CHANGED%
endif
```

TEVENTSSET Segment

This segment contains a 12-byte character field.

Syntax **TEVENTSSET** *tEventsSet*

Notes

1. It is declared internally as follows:

```
SEGMENT TEVENTSSET
  CHAR    ARRAYEVENTS[12]
ENDSEGMENT
```

2. It is used by the CLEAREVENTSSET, ADDEVENT and SUBEVENT instructions and the ISINEVENTSSET% function to store the events that will be proposed in the template control's Events box.

3. The size of the character field is set by: (EVENT_LAST% / 8) + 1

4. Each bit set in the segment indicates that the corresponding event exists.

5. This segment is generally used by <Name>__GETDEFAULTEVENTS. It tells NS-DK which events will be handled by the extended template or custom control. These are the events which are proposed in the template control's Events box, and can be programmed by the user of the template control.

Bitmaps

Specifying a default bitmap for a Custom Control

You can now specify a default bitmap for a Custom Control. Add the following instruction to the definition of the Custom Control: <CustomControlName>__BITMAP.

Here is an example:

```
INSTRUCTION <Custom>__BITMAP COMMAND%(1), POINTER @PtrBMP, @OutSize%(2)

  LOCAL POINTER TMPPBMP
  LOCAL INT SIZE(2)

%
  static unsigned char bmp1[] = {…};
  SIZE = sizeof(bmp1);
  TMPPBMP = bmp1;
%
  OutSize% = SIZE
  PtrBMP = TMPPBMP
ENDINSTRUCTION
```

This instruction specifies the bitmap to use and assigns the return values of PtrBMP and OutSize%.

There are 3 required parameters:

1. is for compatibility and not used
2. pointer to bitmap
3. size of bitmap.

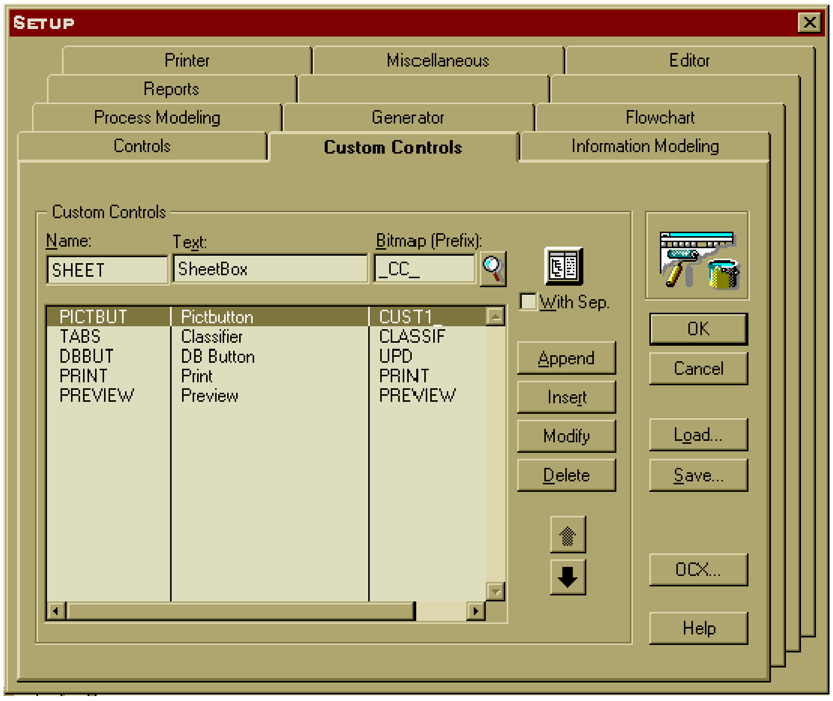
The definition of the bitmap is a byte table. You can create a table using NS-DK by inserting the bitmap to use in a window, checking the preload option, generate the window and open the associated C file. Inside the C file you will find the table;

You can use the example. All you have to do is add the name of the custom control to the instruction and specify the bmp.

Bitmaps must be 26x26 pixels and the first two left columns, the two top rows, the three right columns, and three bottom rows not used. For transparency, the upper left pixel is the background color used to specify the transparency color.

Adding a control to a project

If you want to add a Custom Control using the Option/Setup menu (Customs Controls tab), and if this control has an associated bitmap, you are not required to specify a bitmap. When you enter the name of the of the control, NS-DK automatically displays its associated bitmap. The name of the bitmap has a name similar to that of the control or "_CC_". If you have changed the name of the bitmap and want to use the default bitmap again, enter _CC_ in BitMap (Prefix).



Name: SHEET
Text: SheetBox
Bitmap (Prefix): _CC_

PICTBUT	Pictbutton	CUST1
TABS	Classifier	CLASSIF
DBBUT	DB Button	UPD
PRINT	Print	PRINT
PREVIEW	Preview	PREVIEW

Example: To insert the Custom Control Sheet Box in your project, enter the name of the Custom Control to add (SHEET) and enter the associated text. The name of the default bitmap is automatically displayed when inserting the control by choosing Append or Insert..

Displaying bitmaps

The following custom controls hav been modified to support transparent bitmaps:

1. PictureButton
2. PrintButton
3. DBButton

4. SheetBox

5. The toolbar automatically generated by menus in NS-DK applications

The modification of the DBButtons is automatic as the bitmaps are already known and therefore uses the background color of the dialog box. The border and light gray color of DBButtons is kept only when the background color is light gray (for compatibility) otherwise the border and button background color uses the system colors (depending on the OS and NSx2WPS.DLL) and the bitmaps are transparent.

Dynamic Parameters

Dynamic Parameter .TRANSPARENCY

For PictureButtons and PrintButtons, the new Dynamic Parameter `TRANSPARENCY` and the Info dialog box alow you to specify the same values so that a NS-DK bitmap control (`TRANSP_*%` ou `COL_*%`), where `TRANSP_DEFAULT%` (default value of `TRANSPARENCY`) is handled in the same manner. Buttons created before this modification are handled as if `TRANSPARENCY=TRANSP_DEFAULT%`.

Dynamic Parameter .BACKCOLOR

Specifies the transparency color replacing the current transparency color. Can be either a `COL_*%` constant or –1 to use the background color of the dialog box.

Dynamic Parameter .FORECOLOR

Specifies the transparency color of the bitmap control. Possible values are:

`TRANSP_NEVER%`	Bitmap is not transparent
`TRANSP_DEFAULT%`	If `(GETBITMAPCONTROLSDEFAULTS% BAND DEFBMPTRANSPARENT%) = 0`, the bitmap is not transparent, otherwise transparent using the value `TRANSP_TOPLEFT% by default`
`TRANSP_BOTRIGHT%`	uses the color of the bottom right pixel as transparency color.

`TRANSP_TOPRIGHT%`	uses the color of the top right pixel as transparency color.
`TRANSP_BOTLEFT%`	uses the color of the bottom left pixel as transparency color.
`TRANSP_TOPLEFT%`	uses the color of the top left pixel as transparency color.
`COL_*%`	uses the specified color as transparency color.

Dynamic Parameter .RELIEF

Specifies whether or not to draw the border of Push-Button controls with only a single bitmap (Released) or 3 identical bitmaps (Released, Pressed and Disabled)

If yes, the control is shifted one pixel down and one to the left when the control is pressed. When disabled, half of the pixels of the entire control are removed effectivly graying it out. Possible values are:

BMPF_DEFBTNBORDER%	active only when (GETBITMAPCONTROLSDEFAULTS% BAND DEFBMPBUTTONIZED%) <> 0. Default value.
BMPF_NOBTNBORDER%	always inactive
BMPF_HAVEBTNBORDER%	always active

Dynamic Parameter .INTERNALVALUE

Reads the handle of the three bitmaps (separated by commas) associated with an bitmap control as well as its corresponding modifiers such as ':' and '^'(in this order) ':' specifies that the handle must not be deleted when the button disappears or change its handle. '^' specifies that the handle corresponds to an icon (or pointer) rather than a bitmap. Use the Dynamic Parameter `TEXT` to modify these handles. `INTERNALVALUE` returns information not obtained when reading with but remains compatible.

The constants `TRANSP_*%` and `BMPF_*%` are in NSMISC.NCL.

Handling more than one bitmap in a control

Indexing a bitmap control (bmp1[0] for example) with a number from 0 to 3 allows you to display the part of indexed bitmaps of the same height that is common among them. The height of the layered bitmaps is equal to the full size bitmap.

The index 0 allows you to read or specify the width of the underlying bitmaps (if less than or equal to 0 or if wider than the bitmap, the underlying bitmap lengths are cropped to the bitmap's size)

Index numbers 1 to 3 allow you to read or specify the number of the bitmap for the staatus Released, Pressed and Disabled respectively (a negative index corresponding to 0, the width of the bitmap is adjusted to match the last complete bitmap to its right).

A bitmap having a width of 256 pixels and 32 pixels high can be divided into 8 32x32 bitmaps. If the handle of the bitmap is associated with 3 bitmaps of the bitmap control bmp1, use the following to associate the first thee bitmaps to the 3 different status:

```
bmp1[0] = 32

bmp1[1] = 0

bmp1[2] = 1

bmp1[3] = 2
```

Doing this minimizes the use of system resources (especially in Windows).

New format for bitmaps in listboxes

When specifying the handle of a bitmap to be displayed in a listbox, you can now specify the following optional attributes: "handle^transparent_color^layer_index_number^layer_width".

The default values are:

transparent_color = `TRANSP_DEFAULT%`, refer to `TRANSP_*%` or `COL_*%` for button bitmaps. TRANSP_DEFAULT% is the equivalent of TRANSP_NEVER% for listboxes but may be changed later.

layer_index_number = 0, refer to the bitmap specifications above.

layer_width = 0

Programming method

This section proposes an efficient method for programming an extended template or custom control.

Each development phase will be detailed in the next two chapters with the aid of an example.

This method is in no way compulsory but will help you develop your first few extended templates and custom controls successfully before going on to define your own method based on your experience (if required).

Specifications

To make it easier to define the functional programming requirements for your extended template, it's helpful to begin by defining the resource's specifications.

These specifications should include at least the following details:

- Development environment: data managed.
- Graphical appearance and behavior in Design mode.
- Graphical appearance and behavior in Execution mode.

It is important to distinguish between the implementation of the extended template or custom control in Execution mode and its optional implementation in Design mode.

Execution mode uses the contextual data defined in Design mode, so it's preferable, though not essential, to start by programming the Design mode.

Programming the design mode

Programming the Design mode allows you to modify the default behavior defined in NS-DK.

In the development library for your extended template or custom control, you need to:

1. Define the context segment containing:
   - Contextual data.
   - Run-time data.

2. Write the code that saves and restores the contextual data.

3. Specify whether the template control that will contain the extended template or custom control is resizable.

4. Write the code that displays the extended template or custom control in the window that will contain it and in the tool bar.

5. Define the events that can be programmed by the user of the extended template or custom control.

6. Create and manage the Info box displayed for the extended template or custom control (i.e. the Dialog window used to display and update its contextual data).

Programming the execution mode

Programming an extended template

For an extended template, you need to:

1. Initialize the template's controls (INIT event) so that values are set in some of the template's fields before the window is displayed.

2. If required:

   - Initialize the template's controls using the contextual data defined in Design mode (program the INITCONTEXT event).

   - Initialize the run-time data.

3. Assign values to the template's controls in the window that contains the template (code the SETVALUE event).

4. Retrieve the values of the controls in the window that contains the template (code the GETVALUE event).

5. If required, activate the functions and instructions used to manipulate controls by coding the events with matching names (such as DISABLE, LOCK, HIDE, ISDISABLE and LINECOUNT).

Programming a custom control

For a custom control, you need to:

1. Paint the User Control window (program the PAINT event).
2. Initialize the window (INIT event).
3. If required:
    - Initialize the template's controls using the contextual data defined in Design mode (INITCONTEXT event).
    - Initialize the run-time data.
4. Assign a value to the custom control in the window that contains it (code the SETVALUE event).
5. Retrieve the value of the custom control in the window that contains it (code the GETVALUE event).
6. If required, activate the functions and instructions used to manipulate controls by coding the events with matching names (such as DISABLE, LOCK, HIDE, ISDISABLE and LINECOUNT).
7. If required, handle the mouse (MOUSEMOVE, BUTTONDOWN and BUTTONUP event).
8. If required, handle the keyboard (CHARACTER event).

Items to supply to developers

Once you've programmed and tested your extended template or custom control, you must generate it as a DLL.

When you generate this DLL, an import file with the same name and the .LIB extension will also be generated. This LIB file is used when you link an external program that uses the DLL.

To enable a developer to use your DLL, you need to:

- Create and supply the abbreviated template that contains information about the DLL to use.
- Supply the DLL.

- Supply the import file (LIB file) that will enable developers to generate their application as a DLL or executable file.

- Supply details on the access interface for the abbreviated template:

- How to assign one or more values to the extended template or custom control (assignment operations, indexing, dynamic qualifiers, specific events to send).

- How to retrieve values from the various fields in the extended template or from the custom control.

- If events are used to interface with the extended template or custom control, these events must be specific, i.e. their names should be different from the ones used for the standard behavior events generated automatically by various NCL functions or instructions.

Chapter 3 Example of Developing an Extended Template

This chapter provides a simple example that illustrates the various phases involved in developing and using an extended template.

The programming method described in Chapter 2 of this manual will be developed to cover the creation of extended templates in general; this method will then be applied to the example.

The following information is provided for each phase:

1. Principles.
   Description of the phase for all extended templates.
2. Developing the example.
   Description of the phase for the example.
3. Coding
   Description of the coding required.

This chapter explains

Contents

Specifications

To make it easier to define the functional programming requirements for our extended template, we need to describe the specifications for the various functions that the extended template must provide and its behavior in Design mode and Execution mode.

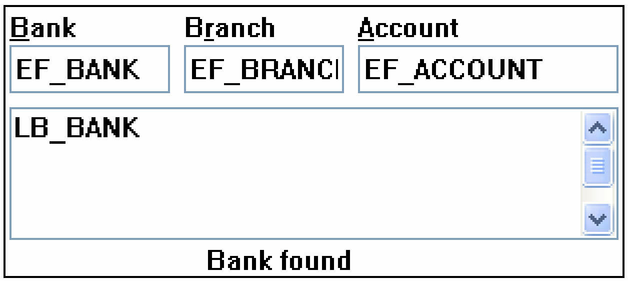
Description of the Extended Template

Our extended template will be based on the template used by the example in Chapter 2 of this manual.

The template that we're going to create will contain:

- A full bank account ID consisting of:
  - Bank code.
  - Branch code.
  - Account number.
- A List box that displays a list of banks (code and text).
- A Static text field that informs the user whether or not the bank was found in the List box.

The template will look like this.



Data being managed

Our extended template's contextual data will mainly be used to configure the bank list, i.e.:

- Set the colors for the first line (title line):
  - Foreground color.
  - Background color.

- Display a Horizontal Scroll bar in the List box:
  - TRUE% (1) Display a Horizontal Scroll bar
  - FALSE% (0) Remove the Scroll bar.

We won't manage any run-time data.

Functions and Behavior

Possible actions

➨ *Write functions*

Our template will enable us to perform the following actions:

- Split a bank account ID number into three fields:
  - Bank code
  - Branch code
  - Account number.
- Select the required bank from a List box.
- Insert a bank at a given position in the template's List box.
- Display a Horizontal Scroll bar in the bank list.

➨ *Read functions*

When the template is used in an NS-DK application, it will allow the user to:

- Compose a full bank account ID number from the 3 fields: bank code, branch code and account number.
- Retrieve the name of a bank based on its position in the list.
- Find out whether the bank List box has a Horizontal Scroll bar.

Creating and Initializing the Extended Template

Principles

An extended template is defined in a template resource.

Choose *File \ Menu* to create this resource.

Developing the example

Choose *File \ New* to create the BANK template with the following characteristics:

```
Name BANK
Width 310
Height 155
```

Add the following controls to the template:

```
Type Entry Field
Name EF_BANK
X, Y 2, 92
Width 80
Parameters Force Fill, Auto-Tab, Length=5,
           Characters='0'..'9', Shadow=Dark,
           Background=Light gray
```

```
Type Static Text
Name ST_BANK
Text ~Bank
Link EF_BANK
X, Y 2, 117
Parameters Auto-Size
```

```
Type Entry Field
Name EF_BRANCH
X, Y 89, 92
Width 80
Parameters Force Fill, Auto-Tab, Length=5,
           Characters='0'..'9', Shadow=Dark,
           Background=Light gray
```

```
Type Static Text
Name ST_BRANCH
Text B~ranch
Link EF_BRANCH
X, Y 89, 117
Parameters Auto-Size
```

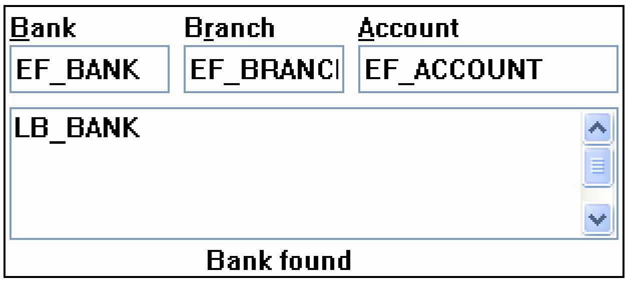
```
Type Entry Field
Name EF_ACCOUNT
X, Y 176, 92
Width 130
Parameters Force Fill, Auto-Tab, Length=5,
           Characters='0'..'9', Shadow=Dark,
           Background=Light gray
```

```
Type Static Text
Name ST_NUM
Text ~Account
Link EF_ACCOUNT
X, Y 176, 117
Parameters Auto-Size
```

```
Type Static Text
Name ST_STATUS
Text Bank found
X, Y 100, 1
Parameters Auto-Size
```

```
Type List Box
Name LB_BANK
X, Y 2, 19
Width 304
File BANK_LST.TXT Preload
Tabulations 0, 100
Separator ',' (comma)
Parameters Auto-Size
```

This produces the following template on the screen.



Managing the Context

Principles

To manage the contextual data in Design mode, we need to code the following function and instructions:

<Name>__SAVECONTEXT	saves the context segment in the window that contains the extended template.
<Name>__LOADCONTEXT	loads the context segment into memory when the window that contains the extended template is displayed in the NS-DK workspace.
<Name>__GETSIZECONTEXT	returns the size of the context segment.

where <Name> stands for the name of the extended template, i.e. the name of the template resource (.TPL).

Developing the example

Code the following functions in the BANK library:

Function

```
Name  BANK__GETSIZECONTEXT
Description  Returns the size of the context segment
Result type  integer
```

Code

```
return sizeof seg_bank_context
```

Function

```
Name  BANK__SAVECONTEXT
Description  Concatenates the contextual data items (stored in the context
segment) and separates each item with a comma in the context string.
Parameters:
      Name   string_bank_context   I/O
      Type   cstring
      Name   bank_context   I/O
      Type   seg_bank_context
      Name   x       I
      Type   integer
      Name   y       I
      Type   integer
      Name   w       I
      Type   integer
      Name   h       I
      Type   integer
```

Code

```
move bank_context.horzscrollbar% to \
     string_bank_context
move string_bank_context & ",X" &   \
     bank_context.forecol to string_bank_context
move string_bank_context & ",Y" &   \
     bank_context.backcol & "Z" to  \
     string_bank_context
```

Function

```
Name  BANK__LOADCONTEXT
Description  retrieves the information in the context string and stores it in
the context segment.
Parameters:
      Name   string_bank_context   I/O
      Type   cstring
      Name   bank_context   I/O
      Type   seg_bank_context
```

Code

```
local p1%,p2%,p3%

if length string_bank_context
   move string_bank_context(0) to \
        bank_context.horzscrollbar%

      p1%=pos%("X",string_bank_context)
      p2%=pos%("Y",string_bank_context)
      p3%=pos%("Z",string_bank_context)

      move copy$(string_bank_context,p1%+1,p2%-p1%-2) \
           to bank_context.forecol
      move copy$(string_bank_context,p2%+1,p3%-p2%-1) \
           to bank_context.backcol
   else
      fill @bank_context,sizeof seg_bank_context,0
      move col_darkgray% to bank_context.forecol
      move col_lightgray% to bank_context.backcol
endif
```

Managing the Template's Size

Principles

By default, an extended template inserted into a window can be resized with the mouse. If you want to modify this feature, you need to code the following functions:

<Name>__CHANGEWIDTH	
<Name>__CHANGEHEIGHT	indicate whether the control can be resized with the mouse and its property pane.
<Name>__MINWIDTH	
<Name>__MINHEIGHT	specify the minimum size of the control in Design mode.

Developing the example

To enable users of the extended template to view all the fields in the template when they insert it into a window, you are advised to disable resizing for the template.

You therefore need to tell NS-DK that the extended template isn't resizable by returning the value FALSE% in the BANK__CHANGEWIDTH and BANK__CHANGEHEIGHT functions (declared in the LTPLBANK.NCL library).

After this, you won't need to code the BANK__MINWIDTH and BANK__MINHEIGHT functions.

Function

```
Name  BANK__CHANGEWIDTH
Description  Indicates whether the width of the control can be resized with the
mouse
Result type  int(1)

Parameters:
      Name   bank_context   I/O
      Type   seg_bank_context
```

➧ *Code*

```
return FALSE%
```

Function

```
Name  BANK__CHANGEHEIGHT
Description  Indicates whether the height of the control can be resized with the
mouse
Result type  int(1)

Parameters:
      Name   bank_context   I/O
      Type   seg_bank_context
```

➧ *Code*

```
return FALSE%
```

Displaying the Template in a Window and in the Toolbar

Principles

You can define the graphical appearance of your extended template in Design mode when it's inserted:

- Into a window that's being edited.
- Into the tool bar.

This description is given in the <Name>__PAINT function.

If this function isn't coded, NS-DK provides a default image:

- The extended template will be displayed in the window as it was designed when defined (the .TPL resource is displayed).
- It will be represented by its name in the tool bar (.TPL name).

Developing the example

The default behavior proposed by NS-DK is adequate for our extended template, so there's no need to code the BANK__PAINT function.

Defining the Set of Programmable Events

Principles

By default, the following events are handled by an extended template:

- INIT
- TERMINATE
- CHECK.

These are the only events that can be programmed by the user of the extended template. They will be displayed in the template control's *Events* box.

If you need to provide additional programmable events, you must specify them using the <Name>__GETDEFAULTEVENTS instruction.

Developing the example

For our BANK template, we will not add any more events.

Coding read/write functions of properties

We will create in LTPLBANK library the functions allowing to recover and modify the value of each template property.

Coding

```
FUNCTION BANK_GET_HORZSCROLLBAR(seg_bank_context@ context, @retVal$, \
SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
            RETURN FALSE%
      ENDIF

      retVal$ = context.horzscrollbar%

      RETURN TRUE%
ENDFUNCTION ; BANK_GET_HORZSCROLLBAR

FUNCTION BANK_SET_HORZSCROLLBAR(seg_bank_context@ context, @val$, \
SEG_ExtendedProp@ ext) RETURN INT(1)
```

```
      IF @ext <> 0
             RETURN PSR_BADACTION% ; invalid action
      ENDIF

      IF NOT IsInt%(val$)
             RETURN PSR_BADVALUE% ; invalid value
      ENDIF

      IF context.horzscrollbar% = INT val$
             RETURN PSR_SameValue%  ; no update necessary
      ENDIF

      context.horzscrollbar% = INT val$

      RETURN PSR_OK% ; update successful

ENDFUNCTION ; BANK_SET_HORZSCROLLBAR


FUNCTION BANK_GET_FORECOLOR(seg_bank_context@ context, @retVal$, \
SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
             RETURN FALSE%
      ENDIF

      retVal$ = context.forecol + 1

      RETURN TRUE%
ENDFUNCTION ; BANK_GET_FORECOLOR

FUNCTION BANK_SET_FORECOLOR(seg_bank_context@ context, @val$, \
SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
             RETURN PSR_BADACTION% ; invalid action
      ENDIF

      IF NOT IsInt%(val$)
             RETURN PSR_BADVALUE% ; invalid value
      ENDIF

      IF context.forecol = INT val$ - 1
             RETURN PSR_SameValue%  ; no update necessary
      ENDIF

      context.forecol = INT val$- 1

      RETURN PSR_OK% ; update successful

ENDFUNCTION ; BANK_SET_FORECOLOR


FUNCTION BANK_GET_BACKCOLOR(seg_bank_context@ context, @retVal$, \
SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
             RETURN FALSE%
      ENDIF

      retVal$ = context.backcol + 1

      RETURN TRUE%
ENDFUNCTION ; BANK_GET_BACKCOLOR

FUNCTION BANK_SET_BACKCOLOR(seg_bank_context@ context, @val$, \
SEG_ExtendedProp@ ext) RETURN INT(1)
      IF @ext <> 0
```

```
               RETURN PSR_BADACTION% ; invalid action
        ENDIF

        IF NOT IsInt%(val$)
               RETURN PSR_BADVALUE% ; invalid value
        ENDIF

        IF context.backcol = INT val$ - 1
               RETURN PSR_SameValue%  ; no update necessary
        ENDIF

        context.backcol = INT val$ - 1

        RETURN PSR_OK% ; update successful

ENDFUNCTION ; BANK_SET_BACKCOLOR
```

Creation of an XML modelisation file template

Principle

The model file with XML format described the whole properties of the template.

Developing the file

Create a BANK.XML file (in the same repertory as BANK.TPL file) containing the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<Entity xmlns:NS="http://tempuri.org/Control.xsd" id="Bank Info"
visibleName="BANK" stdProperties="true" dllName="DTPLBANK.dll">
<Category stdName="MAIN">
  <Property stdName="MODEL" />
  <Property stdName="NAME" />
  <Property stdName="DESCRIPTION" />
  <Property stdName="TOOLTIP" />
  <Property stdName="TAB" />
  </Category>
  <Category stdName="EVENTS" />
  <Category stdName="LAYOUT">
  <Property stdName="X" />
  <Property stdName="Y" />
  <Property stdName="WIDTH" />
  <Property stdName="HEIGHT" />
  <Property stdName="ANCHOR" />
  </Category>
  <Category stdName="LOOK">
  <Property id="1" text="Horizontal scrollbar">
  <Description>Disabled</Description>
  <Editor kind="Bool" />
  <Callback>
  <Get>BANK_GET_HORZSCROLLBAR</Get>
  <Set>BANK_SET_HORZSCROLLBAR</Set>
  </Callback>
  </Property>
  <Property id="2" text="Foreground color">
  <Editor kind="Colors" />
```

```
<Callback>
<Get>BANK_GET_FORECOLOR</Get>
<Set>BANK_SET_FORECOLOR</Set>
</Callback>
</Property>
<Property id="3" text="Background color">
<Description>Background color</Description>
<Editor kind="Colors" />
<Callback>
<Get>BANK_GET_BACKCOLOR</Get>
<Set>BANK_SET_BACKCOLOR</Set>
</Callback>
</Property>
</Category>
</Entity>
```

Behavior Events: GetValue and SetValue

Principles

Changing the value of a NS-DK control

A value can be assigned to a NS-DK control by programming. This is usually done with the following instruction:

```
MOVE value TO control
```

After this, the value of aNS-DK control can be retrieved by using the instruction:

```
MOVE control TO variable
```

When these instructions are executed, they trigger the following events:

• SETVALUE	to set the value of a control (event generated by `MOVE value TO control`)
• GETVALUE	to obtain the value of a control (event generated by `MOVE control TO variable`)

Changing the value of an extended template

To modify or retrieve the value of the controls in an extended template by programming, you need to code the GETVALUE and SETVALUE events.

The GETVALUE event is generated by the following 3 instructions:

```
MOVE <TEMPLATE_CONTROL> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL[index]> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL.Paramdyn> TO <VARIABLE>
```

The SETVALUE event is generated by 3 instructions:

```
MOVE <value> TO <TEMPLATE_CONTROL>
MOVE <value> TO <TEMPLATE_CONTROL[index]>
MOVE <value> TO <TEMPLATE_CONTROL.Paramdyn>
```

Where:

- *index* is used to manipulate a template that contains list objects (List Boxes, Combo Boxes, CBEs and MLEs).
- *Paramdyn* is used to modify the template's parameters dynamically.

Developing the example

SETVALUE event

We will perform the following operations, depending on how the event was called:

- Initialize the three fields that form the account number.
- Insert a bank name at a specified position in the LB_BANK list.
- Control the display of a Horizontal Scroll bar in the LB_BANK list.

Therefore, we need to handle this event using 3 types of instructions:

```
MOVE <value> TO <TEMPLATE_CONTROL>
MOVE <value> TO <TEMPLATE_CONTROL[index]>
MOVE <value> TO <TEMPLATE_CONTROL.Paramdyn>
```

GETVALUE event

We will return the following information in PARAM2$, depending on how the event was called:

- A concatenation of the three fields that form the account ID number.
- The name of the bank whose index in LB_BANK was passed when the event was called.
- TRUE% or FALSE% to indicate whether or not a Horizontal Scroll bar is displayed in LB_BANK.

Therefore, we need to handle this event using 3 types of instructions:

```
MOVE <TEMPLATE_CONTROL> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL[index]> TO <VARIABLE>
MOVE <TEMPLATE_CONTROL.Paramdyn> TO <VARIABLE>
```

The type of information returned will be determined by the contents of the PARAM1% and PARAM2% parameters passed to GETVALUE.

Coding

➔ SETVALUE event for the BANK template

```
;_______________________________________________________
; Param2Value$: Value of PARAM2$
;               PARAM2$ is initialized when the
;               following instructions are executed:
;       MOVE <value> TO <TEMPLATE_CONTROL>
;       MOVE <value> TO <TEMPLATE_CONTROL[index]>
;       MOVE <value> TO <TEMPLATE_CONTROL.ParamDyn>
;
; I%:           Index used to select lines from the
;               LB_BANK List box
;
; ListLine$:    Selected line in LB_BANK
;_______________________________________________________
Local Param2Value$
Local I%
Local ListLine$


; Test how the SETVALUE event was generated:
;  1. MOVE <variable> TO <ctl_template>
;     then PARAM1% = DEFRET%
;  2. MOVE <variable> TO <ctl_template[index]
;     then PARAM2% = 0
;     and PARAM1% = index
;  3. MOVE <variable> TO <ctl_template.Paramdyn>
;     then PARAM2% <> 0
;     and PARAM1$ = "Paramdyn"
;
; The corresponding value is in PARAM2$

; Save the value passed by the MOVE instruction
Param2Value$ = PARAM2$

IF PARAM1% = DEFRET%

       ; Initialize the components of the bank account
       ; number using the full account number passed by
       ; the MOVE instruction
       EF_BANK = Copy$ (Param2Value$,1,5)
       EF_BRANCH = Copy$ (Param2Value$,6,5)
       EF_ACCOUNT = Copy$ (Param2Value$,11,11)

       ; Select the corresponding bank in the list box,
       ; if exists.
       ; If the bank does not exist in the list,
       ; display "bank not found"
       I% = 0
       NOUPDATE LB_BANK
       While I% < Linecount%(LB_BANK)
         SELECT I% FROM LB_BANK
         ListLine$ = LB_BANK
         ListLine$ = Copy$(ListLine$, \
                     1,POS%(",",ListLine$) - 1)
         IF EF_BANK = ListLine$
            I% = Linecount%(LB_BANK) + 2
            ST_STATUS = "Bank found"
         ELSE
            I% = I% + 1
         ENDIF
       EndWhile
       UPDATE LB_BANK
       IF I% = Linecount%(LB_BANK)
          ST_STATUS = "Bank not found"
          UNSELECT I% - 1 FROM LB_BANK
       ENDIF

ELSE
   IF PARAM2% <> 0
      IF PARAM1$ = "HORZSCROLLBAR"
         ; Show or hide the horizontal scroll bar
         ;(TRUE% or FALSE%)
         LB_BANK.HORZSCROLLBAR = Param2Value$
      ENDIF
   ELSE

      ; Insert a line into the list box
      IF PARAM1% = 0
         I% = 1
      ELSE
         I% = PARAM1%
      ENDIF
      INSERT AT I% Param2Value$ TO LB_BANK

   ENDIF
ENDIF
```

➔ *GETVALUE event for the BANK template*

```
;________________________________________________________
;BankName$: String variable used to extract the bank name
;           from the selected line in the LB_BANK
;           List box
;I%:        Index used to select lines from the LB_BANK
;           List box
;________________________________________________________
Local BankName$
Local I%


; Test how the GETVALUE event was generated:
;  1. MOVE <ctl_template> TO <variable>
;     then PARAM1% = DEFRET%
;  2. MOVE <ctl_template[index] TO <variable>
;     then PARAM2% = 0
;      and PARAM1% = index
;  3. MOVE <ctl_template.Paramdyn> TO <variable>
;     then PARAM2% <> 0
;     and PARAM1$ = "Paramdyn"
;
;Return the corresponding value in PARAM2$
IF PARAM1% = DEFRET%

    ; Concatenate the 3 components of a bank account
    ; number
    PARAM2$ = EF_BANK & EF_BRANCH  & EF_ACCOUNT

ELSE
    IF PARAM2% <> 0
      IF PARAM1$ = "HORZSCROLLBAR"
         ; Return TRUE% or FALSE%
         PARAM2$ = LB_BANK.HORZSCROLLBAR
      ENDIF
   ELSE
```

```
      ; Select the indexed line from the LB_BANK
      ; List box then extract the bank name
      IF PARAM1% = 0
          I% = 1
      ELSE
          I% = PARAM1%
      ENDIF

      IF I% > LINECOUNT% (LB_BANK) - 1
         PARAM2$ = "Unknown Index"
      ELSE
         SELECT I% FROM LB_BANK
         BankName$ = LB_BANK
         BankName$ = COPY$(BankName$,               \
                           POS%(",",BankName$) + 1,\
                           Length (BankName$) -  \
                           POS%(",",BankName$))
         PARAM2$ = BankName$
      ENDIF

   ENDIF
ENDIF
```