Code Conversion

This chapter covers the following topics:

- General Information
- Generating Adapters
- Structure of a Map-Based Application
- Structure of a Natural for Ajax Application
- Tasks of the Code Conversion
- DEFINE DATA Statement
- INPUT Statement
- REINPUT Statement
- PF-Key Event Handling
- SET KEY Statement
- Processing Rules
- System Variables
- Variable Names Containing Special Characters

General Information

After the Map Converter has been used to create page layouts from map extract files, the last step in the conversion process is adapting the application code to the new user interface. This step can either be performed manually or, with Natural Engineer, partly automatically. In the following, the manual code conversion is described.

Generating Adapters

First of all, it is necessary to generate HTML code and Natural adapters from the page layouts that have been created by the Map Converter. This is the same procedure as with page layouts that have been created manually with the Layout Painter. Then, the adapters are imported into the Natural development environment.

Structure of a Map-Based Application

In this context, we need not consider the application code as a whole, but only the layer that handles the user interface. Often, the user interface handling part of a map-based application is structured in the following way:

- `DEFINE DATA`
- Initialization
- `REPEAT`
  - `INPUT [USING MAP` *`map-name`*`]`
    - Includes client-side validations (processing rules)
  - Server-side validations
    - `REINPUT` or `ESCAPE TOP`
  - `DECIDE ON *PF-KEY`
    - Function key handler 1
      - Processing
      - `REINPUT` or `ESCAPE TOP`
    - Function key handler 2
      - Processing
      - `REINPUT` or `ESCAPE TOP`
    - Function key handler *n*
      - Processing
      - `ESCAPE BOTTOM`
    - ...
  - `END-DECIDE`
- `END-REPEAT`
- Cleanup
- `END`

In practice,

- the `REPEAT` loop might or might not be there, and
- there might not be a clean `DECIDE` structure for the function key handlers. Instead, checks for the pressed function key might be spread all over the code.

However, accepting these differences, the above structure should match a large number of applications.

Structure of a Natural for Ajax Application

The corresponding part of a Natural for Ajax application looks as follows:

- `DEFINE DATA`
- Initialization
- `REPEAT`
  - `PROCESS PAGE USING` *`adapter-name`*
    - Includes client-side validations
  - Server-side validations
    - `PROCESS PAGE UPDATE FULL`
  - `DECIDE ON *PAGE-EVENT`
    - Event handler 1
      - Processing
      - `PROCESS PAGE UPDATE FULL` or `ESCAPE TOP`
    - Event handler 2
      - Processing
      - `PROCESS PAGE UPDATE FULL` or `ESCAPE TOP`
    - Event handler *n*
      - Processing
      - `ESCAPE BOTTOM`
    - ...
  - `END-DECIDE`
- `END-REPEAT`
- Cleanup
- `END`

Tasks of the Code Conversion

The code conversion should achieve the following:

- It should be minimal invasive.
- It should not duplicate business code.
- The converted application should be able to run not only with the new user interface, but also in a terminal session, in a Natural Web I/O Interface session and in batch, if it did so before the code conversion.

In detail, the code conversion needs to deal with the statements and constructs mentioned below.

DEFINE DATA Statement

The `DEFINE DATA` statement must be extended because the data structures exchanged between a program and map are not fully identical to those exchanged between a program and the corresponding adapter.

The default conversion rules delivered with the Map Converter perform a data type mapping that tries to ensure that the data elements in the map interface are mapped to data elements of the same type and name in the adapter interface.

The Application Designer controls are usually not only bound to business data elements, but also to additional control fields. Which control fields these are depends on the way in which the elements of a map are mapped to Application Designer controls by the Map Converter rules. For instance, a `statusprop` can be assigned to a field, which results in an additional parameter in the parameter data area of the adapter. An array on a map can have been converted to a grid control with server-side scrolling. In this case, the additional data structures needed to control server-side scrolling need to be added to the `DEFINE DATA` statement.

statusprop

The `statusprop` is needed to control the error status or focus of a FIELD control dynamically (see example 3 for the `REINPUT` statement below where it is used to replace the `MARK *`*`field-name`* clause). The default conversion rules contain a rule that creates a `statusprop` property for each map field that is controlled by a control variable. The adapter generator creates from this property a corresponding status variable and a comment line that identifies the status variable as belonging to the field.

Example

The parameter data area of the map contains:

```
01 LIB-NAME (A8)
01 LIB-NAME-CV (C)
```

The parameter data area of the adapter will then contain:

```
* statusprop= STATUS_LIB-NAME-CV
01 LIB-NAME (A8)
01 STATUS_LIB-NAME-CV (A) DYNAMIC
```

The variable `STATUS_LIB-NAME-CV` is not yet known to the main program and must be defined there.

INPUT Statement

The replacement for the INPUT statement is the PROCESS PAGE statement. In its simplest form, the INPUT statement just references the map. In this case, it is just replaced by a PROCESS PAGE statement with the corresponding adapter.

Example 1

Main program before conversion:

```
INPUT USING MAP 'MMENU'
```

Main program after conversion:

```
IF *BROWSER-IO NE 'RICHGUI'
  INPUT USING MAP 'MMENU'
ELSE
  PROCESS PAGE USING 'AMENU'
END-IF
```

The INPUT statement can come with a message text that is displayed in the status bar. There is no direct replacement for this construction because the PROCESS PAGE statement (in contrast to the PROCESS PAGE UPDATE statement) does not support the SEND EVENT clause.

Example 2

Main program before conversion:

```
INPUT WITH TEXT MSG01 USING MAP 'MMENU'
```

Main program after conversion (no message will be displayed):

```
IF *BROWSER-IO NE 'RICHGUI'
  INPUT WITH TEXT MSG01 USING MAP 'MMENU'
ELSE
  PROCESS PAGE USING 'AMENU'
END-IF
```

REINPUT Statement

The replacement for the REINPUT statement is the PROCESS PAGE UPDATE statement. In its simplest form, the REINPUT statement comes with a message text that is displayed in the status bar. In the converted code, this is handled by the SEND EVENT clause of the PROCESS PAGE UPDATE statement.

Example 1

Main program before conversion:

```
REINPUT [FULL] WITH TEXT MSG01
```

Main program after conversion:

```
IF *BROWSER-IO NE 'RICHGUI'
  REINPUT [FULL] WITH TEXT MSG01
ELSE
  PROCESS PAGE UPDATE [FULL]
    AND SEND EVENT 'nat:page.message'
    WITH PARAMETERS
      NAME 'type' VALUE 'E'
      NAME 'short' VALUE MSG01
    END-PARAMETERS
END-IF
```

The REINPUT statement can come with a message number and replacements. In this case, the message must be created from number and replacements before it is sent to the status bar with the SEND EVENT clause.

Example 2

This example uses a subprogram GETMSTXT that builds the message text from number and replacements.

Main program before conversion:

```
REINPUT [FULL] WITH TEXT *MSGNR, REPL1, REPL2
```

Main program after conversion:

```
IF *BROWSER-IO NE 'RICHGUI'
  REINPUT [FULL] WITH TEXT *MSGNR, REPL1, REPL2
ELSE
  CALLNAT 'GETMSTXT' MSTEXT MSGNR REPL1 REPL2
  PROCESS PAGE UPDATE [FULL]
    AND SEND EVENT 'nat:page.message'
    WITH PARAMETERS
      NAME 'type' VALUE 'E'
      NAME 'short' VALUE MSTEXT
    END-PARAMETERS
END-IF
```

Example 3

The REINPUT statement can come with a MARK clause in order to put the focus on a field. This case requires that a statusprop property is created for the field during map conversion. The variable bound to the statusprop property is then used before the PROCESS PAGE UPDATE statement to set the FOCUS to the field.

Main program before conversion:

```
REINPUT [FULL] WITH TEXT MSG01 MARK *LIB-NAME
```

Main program after conversion:

```
01 STATUS_LIB-NAME-CV (A) DYNAMIC
...
IF *BROWSER-IO NE 'RICHGUI'
  REINPUT [FULL] WITH TEXT MSG01 MARK *LIB-NAME
ELSE
  STATUS_LIB-NAME-CV := 'FOCUS'
  PROCESS PAGE UPDATE FULL
    AND SEND EVENT 'nat:page.message'
    WITH PARAMETERS
```

```
      NAME 'type' VALUE 'W'
      NAME 'short' VALUE MSG01
    END-PARAMETERS
END-IF
```

PF-Key Event Handling

The original application might contain checks for the content of the system variable `*PF-KEY` at arbitrary places in the code. In order to handle function key events correctly in the converted application, several things need to be achieved:

- In response to the function keys, the converted application must raise events that are named like the possible contents of `*PF-KEY`. This can be achieved by using a page template such as *NATPAGEHOTKEYS_TEMPLATE.xml* which contains the required hot key definitions.
- A common local variable must be set up right after the `INPUT` or `PROCESS PAGE` statement that contains either the value `*PF-KEY` or `*PAGE-EVENT`, depending on the execution environment. The name of the variable can be freely chosen. In the example below, the name `XEVENT` is used.
- The event `nat:page.end` must be handled in such a way so that the program terminates. This event is raised when the user leaves the page or closes the browser session.
- A default event handler must be set up that takes care of the values of `*PAGE-EVENT` that are not expected by the original application code. These unexpected events are simply replied with a `PROCESS PAGE UPDATE FULL` statement.

Example

```
01 XEVENT (U) DYNAMIC
...
PROCESS PAGE USING ...
...
IF *BROWSER-IO = 'RICHGUI'
  DECIDE FOR FIRST CONDITION
    WHEN *PAGE-EVENT = 'nat:page.end'
      STOP
    WHEN *PAGE-EVENT = MASK ('PF'*) OR = MASK ('PA'*)
    OR = 'ENTR' OR = 'CLR'
      XEVENT := *PAGE-EVENT
    WHEN NONE
      PROCESS PAGE UPDATE FULL
  END-DECIDE
ELSE
  XEVENT := *PF-KEY
END-IF
```

All references to `*PF-KEY` in the code must then be replaced by references to `XEVENT`.

SET KEY Statement

Natural for Ajax provides two controls (NJX:BUTTONITEMLIST and NJX:BUTTONITEMLISTFIX) that represent a row of buttons. These controls can be used to replace the visual representation of the function keys from the original application. If the page template *NATPAGEPFKEYS_TEMPLATE.xml* or a similar individually adapted template is used during map conversion, each resulting page will contain a row of function key buttons. The subject of this section is how the converted application can control the

labeling and the program-sensitivity of the function keys with only little code changes.

Natural controls the labeling and program-sensitivity of the function keys in a highly dynamic way. The corresponding application code (`SET KEY` statements) can be distributed across program levels and can be lexically separated from the corresponding `INPUT` statements. Also, the `SET KEY` statement has several flavors, some affecting all keys and others affecting only individual keys. As a result, the status of the function keys at a given point in time can only be determined at application runtime.

Therefore, the following approach is chosen: Natural provides the application programming interface (API) `USR4005` that reads the current function key naming and program-sensitivity at runtime. During code conversion, a call to this API is inserted after each `SET KEY` statement or into each round trip. This call reads the function key status and passes it to the user interface.

Example

Main program before conversion:

```
SET KEY ENTR NAMED 'Enter' PF1 NAMED 'F1' PF2 NAMED 'F2'
PF3 NAMED 'Modify' PF4 NAMED 'Delete' PF5 NAMED 'F5'
PF6 NAMED 'F6' PF7 NAMED 'Create' PF8 NAMED 'Display'
PF9 NAMED 'F9' PF10 NAMED 'F10' PF11 NAMED 'F11' PF12 NAMED 'F12'
*
INPUT USING MAP "KEYS-M"
*
END
```

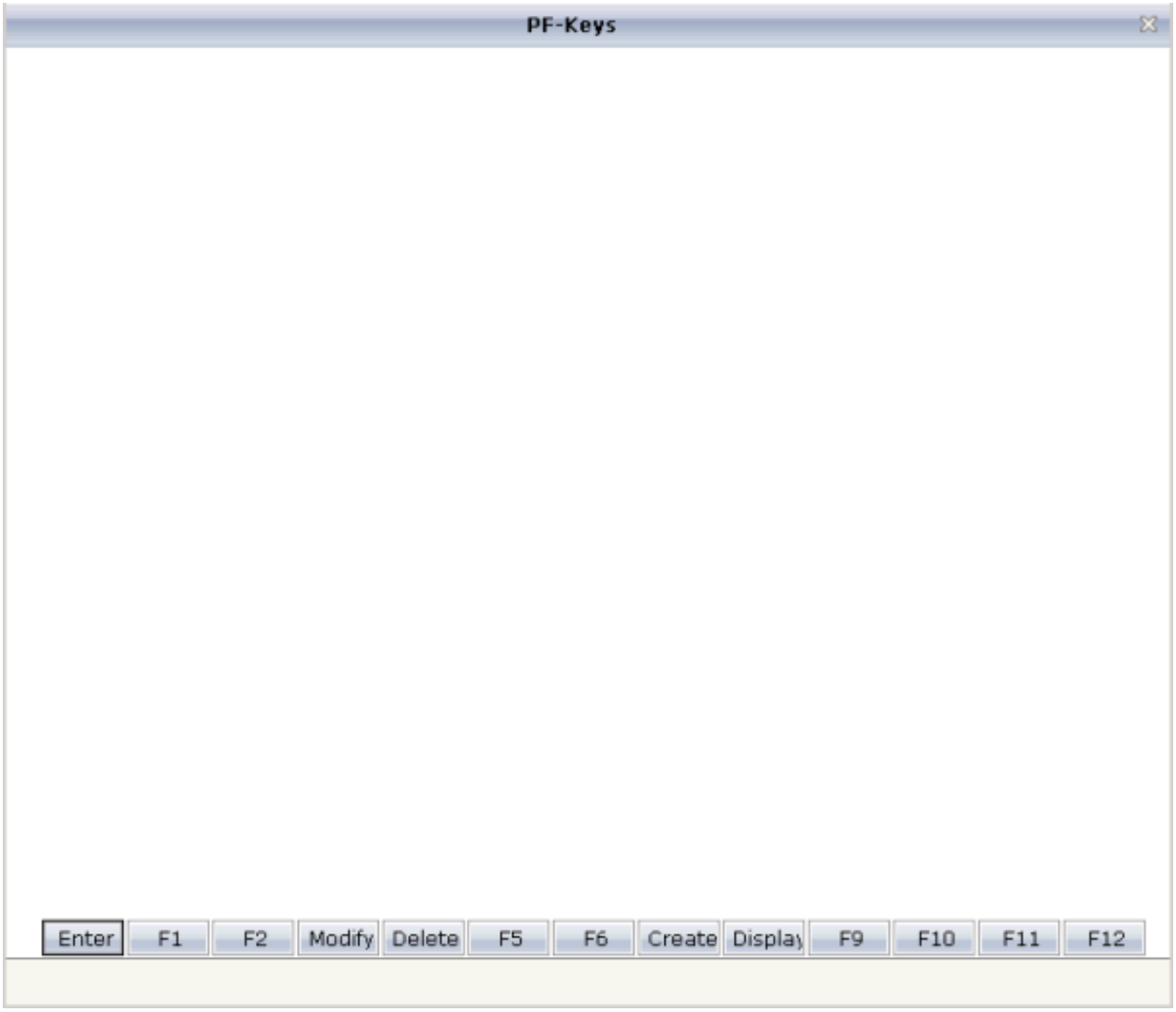
Map before conversion:

```
                                  *** PF-Keys ***









Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
Enter F1    F2    Modif Delet F5    F6    Creat Displ F9    F10   F11   F12
```

Main program after conversion:

```
DEFINE DATA LOCAL
1 PFKEY (1:*)
2 METHOD (A) DYNAMIC
2 NAME (A) DYNAMIC
2 TITLE (A) DYNAMIC
2 VISIBLE (L)
1 METHODS (A4/13) CONST <'ENTR','PF1','PF2','PF3','PF4',
'PF5','PF6','PF7','PF8','PF9','PF10','PF11','PF12'>
END-DEFINE
*
SET KEY ENTR NAMED 'Enter' PF1 NAMED 'F1' PF2 NAMED 'F2'
PF3 NAMED 'Modify' PF4 NAMED 'Delete' PF5 NAMED 'F5'
PF6 NAMED 'F6' PF7 NAMED 'Create' PF8 NAMED 'Display'
PF9 NAMED 'F9' PF10 NAMED 'F10' PF11 NAMED 'F11' PF12 NAMED 'F12'
*
IF *BROWSER-IO NE "RICHGUI"
  INPUT USING MAP "KEYS-M"
ELSE
  EXPAND ARRAY PFKEY TO (1:13)
  METHOD(1:13) := METHODS (*)
  CALLNAT "GETKEY-N" PFKEY (*)
  PROCESS PAGE USING "KEYS-A"
END-IF
*
END
```

Page after conversion:



Explanation

The structure PFKEY is generated into the Natural adapter of the page as the application interface to the BUTTONITEMLISTFIX control.

The subprogram GETKEY-N is a convenience wrapper for the API subprogram USR4005. It uses USR4005 to determine the labeling and the program-sensitivity status for a given list of function keys. Each function key is identified by the *PF-KEY value it raises. GETKEY-N returns the function key information in a data structure suitable for the application interface of the BUTTONITEMLISTFIX control. The subprogram is delivered in the library SYSEXNJX in source code and can be adapted to the needs of the application.

Processing Rules

The Natural maps in the application to be converted may contain processing rules. In the sense of a Natural for Ajax application, the processing rules are server-side validations because they are executed on the Natural server side of the application.

In order to extract processing rules from the maps and to turn them into server-side validations in the converted application, the Natural Engineer function "Separate Processing Rules from Maps" can be used.

There is currently no function available that automatically turns processing rules into client-side validations in Application Designer.

System Variables

If a map displays a system variable (for example, `*DATX`), a specific default conversion rule takes care that the necessary code for handling the system variable is generated into the Natural adapter of the resulting page layout.

Example 1

The map displays the contents of the system variables `*DATX` and `*TIMX`. The contents of these system variables are not modifiable.

The `DEFINE DATA` statement of the adapter will then contain:

```
LOCAL
01 XDATX (A8)
01 XTIMX (A8)
```

The body of the adapter will then contain:

```
XDATX := *DATX
XTIMX := *TIMX
*
PROCESS PAGE ... WITH
PARAMETERS
...
 NAME U'XDATX'
  VALUE XDATX
 NAME U'XTIMX'
  VALUE XTIMX
END-PARAMETERS
```

The main program needs no special adaptation.

Example 2

The map displays the content of the system variable `*CODEPAGE`. The content of this system variables is modifiable.

The `DEFINE DATA` statement of the adapter will then contain:

```
LOCAL
01 XCODEPAGE (A64)
```

The body of the adapter will then contain:

```
XCODEPAGE := *CODEPAGE
*
PROCESS PAGE ... WITH
PARAMETERS
...
 NAME U'XCODEPAGE'
  VALUE XCODEPAGE
...
END-PARAMETERS
*
*CODEPAGE := XCODEPAGE
```

The main program needs no special adaptation.

Variable Names Containing Special Characters

A similar procedure applies to special characters contained in variable names. These are the following special characters:

+
#
/
@
§
&
$

Note:
The hash (#) can occur only as the first character.

Variables names containing these special characters cannot be directly bound to Application Designer control attributes. A specific default conversion rule replaces the names containing these special characters with configurable replacements. The original field name is generated into the parameter data area of the Natural adapter and a corresponding mapping is generated into the `PROCESS PAGE` statement of the adapter.

Example

The map displays the variables `#FIRST` and `#LAST`.

The `DEFINE DATA` statement of the adapter will then contain:

```
DEFINE DATA PARAMETER
1 #FIRST (A16)
1 #LAST (A20)
```

The body of the adapter will then contain:

```
...
PROCESS PAGE ... WITH
PARAMETERS
...
NAME U’HFIRST’
  VALUE #FIRST
 NAME U’HLAST’
  VALUE #LAST
...
END-PARAMETERS
```

The main program needs no special adaptation.