Text System Defaults and Key Bindings

OPENSTEP 4.x has a new text system. This document reveals some tips and tricks about various defaults you can use to customize its behavior. It also describes how to customize the key bindings supported by the new text system.

Note that the new text object exists only in the Release 4.x version of the Application Kit; the following notes don't apply to NEXTSTEP Release 3.3 applications. Also note that the old (3.3) text object exists in the 4.0 Application Kit; these defaults don't apply to it nor to any OpenStep applications which use the old text object.

Heavy-duty subclassers may alter some or all of the text system's functionality, rendering some or all of these features inactive. These notes do apply to NeXT's OPENSTEP applications such as Project Builder, Interface Builder, Text Edit, and others which use the new text system.

Text System Defaults

NSModifierFlagMapping (dictionary) (Windows platform only)

This default is on OPENSTEP for Windows only. It allows you to control the mapping between physical modifier keys and logical modifier flags in OpenStep. This default is actually not specific to the text system, but its main purpose is to allow Emacs bindings to work under Windows. By default, both Control keys generate the Command key bit (for menu key equivalents) and both Alt keys generate the Alternate key bit (for mnemonics, primarily). The Control key bit is not available in the default setup which means it is not possible to invoke Emacs-style commands in the text system. This default can be used to re-map the available keys to generate what you want. The value of the default is a dictionary with four possible keys, each of which can have one of three possible values. The dictionary keys are: “LeftControl”, “RightControl”, “LeftAlt”, and “RightAlt”. The valid values are: “Command”, “Alt”, and “Control”. So the default setup is like this:

```
{
    "LeftControl" = "Command";
    "RightControl" = "Command";
    "LeftAlt" = "Alt";
    "RightAlt" = "Alt";
}
```

One possible setup that allows you to use Emacs keys would be:

```
{
    "LeftControl" = "Command";
    "RightControl" = "Command";
```

```
        "LeftAlt" = "Control";
        "RightAlt" = "Alt";
    }
```

This would make it so the left Alt key acts like a control key for Emacs. The right Alt key is still used for Alt.

Currently this default has a limitation that only a real Alt key (left or right) can be used for the Alt bit. Therefore it is not valid to assign "LeftControl" = "Alt".

NSMnemonicsWorkInText ("YES" or "NO")

This default controls whether the text system accepts key events with the Alt key down. The default value is NO on Mach and YES on Windows. A value of YES means that any key event with the Alt bit on will be passed up the responder chain to eventually be treated as a mnemonic instead of being accepted by the text as textual input or a key binding command. If this default is set to NO then the key events with the Alt bit set will be passed through the text system's normal key input sequence. This will allow any key bindings involving Alt to work (such as Emacs-style bindings like Alt-f for word forward) and, on Mach it allows typing of special international and Symbol font characters.

NSRepeatCountBinding (key binding style string)

This default controls the numeric argument binding. The default is for numeric arguments not to be supported. If you provide a binding for this default you enable the feature. This allows you to repeat a keyboard command a given number of times. For instance "Control-U 10 Control-F" means move forward ten characters.

NSQuotedKeystrokeBinding (key binding style string)

This default controls the quote binding. The default is for this to be "^q" (that's Control-Q). This is the binding that allows you to literally enter characters that would otherwise be interpreted as commands. For instance "Control-Q Control-F" would insert a Control-F character into the document instead of performing the command **moveForward:**.

NSTextShowsInvisibleCharacters ("YES" or "NO")

The default controls whether a text object will by default show invisible characters like tab, space, and carriage return using some visible glyph. By default it is NO. It only controls the default setting for NSLayoutManagers (which can be modified programmatically). In order for this to work, the rule book generating the glyphs must support the feature. Currently our rule books do not support this feature, so currently this default is not very useful.

NSTextShowsControlCharacters ("YES" or "NO")

The default controls whether a text object will by default show control characters visibly (usually by representing Control-C as "^C" in the text). By default it is NO. It only controls the default setting for NSLayoutManagers (which can be modified programmatically). In order for this to work, the rulebook(s) generating the glyphs must support the feature. This feature carries a cost. It will increase the memory needed for documents that contain control characters by quite a lot. Use it with care.

NSTextSelectionColor (color)

This default controls the background color of selected text. By default this is light gray. Kit defaults that accept colors accept them in one of three ways. Either as an archived NSColor object, or as three RGB components, or as a string that can be resolved to a factory selector on NSColor that will return the desired color (for example, "redColor"). Note that NSTextFields and other controls that use field editors to edit their text control their own selection attributes to conform with the platform UI.

NSMarkedTextAttribute and NSMarkedTextColor (color or "underline")

This default controls the way that marked text is displayed. The NSMarkedTextAttribute can be either "Background" or "Underline". If it is "Background" then NSMarkedTextColor indicates the background color to use for marked text. If NSMarkedTextAttribute is "Underline", NSMarkedTextColor indicates the foreground color to use for marked text (the marked text will be drawn in the indicated color and underlined). By default, marked text is drawn with a yellow-ish background color. Kit defaults that accept colors accept them in one of three ways. Either as an archived NSColor object, or as three RGB components, or as a string that can be resolved to a factory selector on NSColor that will return the desired color (for example, "redColor"). For compatibility with the way this default worked in 4.0, if the NSMarkedTextAttribute default contains a color instead of one of the strings "Background" or "Underline" then that color is used as the background color for marked text and the NSMarkedTextColor attribute is ignored.

NSTextKillRingSize (number string)

This default controls the size of the kill ring (as in Emacs Control-Y). The default value is 1 (not really a ring at all, just a single buffer). If you set this to a value larger than one, you also need to rebind Control-Y to "yankAndSelect:" instead of "yank:" for things to work properly (note that **yankAndSelect:** is not listed in any headers). See below for more info on bindings.

Key bindings

The new text system uses a generalized key binding mechanism which is completely re-mappable by the user. The standard bindings for can always be found in

NextLibrary/Frameworks/AppKit.framework/Resources/StandardKeyBinding.dict or in **NextLibrary/Frameworks/AppKit.framework/Resources/StandardKeyBinding-winnt.dict**. On both platforms these standard bindings include a large number of Emacs-compatible control key bindings, all the various arrow key bindings, bindings for making field editors and some keyboard UI work, and backstop bindings for many function keys. On Windows the standard bindings also include a number of Emacs-compatible Alt key bindings (like Alt-f, Alt-b).

All these bindings are customizable by the user. You can create a file in **~/Library/KeyBindings/DefaultKeyBinding.dict** to augment or replace the standard bindings. Use the standard bindings files as templates. Modifier flags are specified using special characters: "^" for control, "~" for Alt, "$" for Shift, and "#" for numeric keypad. Multiple keystroke bindings are supported through nested binding dictionaries. For instance, Escape could be bound to "cancel:" or it could be bound to a whole dictionary which would then contain bindings for the next keystroke after Escape.

Here are a couple sample binding files that you might use:

1. The first one adds Alt-key bindings for some common Emacs stuff. This might be useful on Mach where the Alt-key bindings are not standard. With these bindings it would be necessary to type "Control-Q, Alt-f" in order to type a florin character instead of moving forward a word. This sample also explicitly binds Escape to "complete:". On Mach, this is the default so this override changes nothing, but on Windows, Escape is bound to "cancel:" by default, so this example changes it so Escape will mean **complete:** when a text object is key (it will still mean **cancel:** if some non-textual thing, like an NSButton, is key).

```
/* ~/Library/KeyBindings/DefaultKeyBinding.dict */

{
    /* Additional Emacs bindings */
    "~f" = "moveWordForward:";
    "~b" = "moveWordBackward:";
    "~<" = "moveToBeginningOfDocument:";
    "~>" = "moveToEndOfDocument:";
    "~v" = "pageUp:";
    "~d" = "deleteWordForward:";
    "~^h" = "deleteWordBackward:";
    "~\010" = "deleteWordBackward:";  /* Alt-backspace */
    "~\177" = "deleteWordBackward:";  /* Alt-delete */

    /* Escape should really be complete: */
    "\033" = "complete:";  /* Escape */
```

```
}
```

2. This example shows how to have multi-keystroke bindings. It binds a number of Emacs meta bindings using Escape as the meta key instead of the Alt modifier. So Escape followed by f means **moveWordForward:** here. This sample binds Esc-Esc to “complete:”. Note the nested dictionaries.

```
/* ~/Library/KeyBindings/DefaultKeyBinding.dict */

{
    /* Additional Emacs bindings */
    "\033" = {
        "\033" = "complete:";  /* ESC-ESC */
        "f" = "moveWordForward:";  /* ESC-f */
        "b" = "moveWordBackward:";  /* ESC-b */
        "<" = "moveToBeginningOfDocument:";  /* ESC-< */
        ">" = "moveToEndOfDocument:";  /* ESC-> */
        "v" = "pageUp:";  /* ESC-v */
        "d" = "deleteWordForward:";  /* ESC-d */
        "^h" = "deleteWordBackward:";  /* ESC-Ctrl-H */
        "\010" = "deleteWordBackward:";  /* ESC-backspace */
        "\177" = "deleteWordBackward:";  /* ESC-delete */
    };
}
```

With the right combination of key bindings and default settings, it should be possible to tailor the text system to your preferences.

Key Bindings in Project Builder

ProjectBuilder uses the Application Kit's key binding mechanism as described above to define key bindings for its code editor. These key-binding definitions are in **NextDeveloper/Apps/ProjectBuilder.app/Resources/KeyBindings.dict**. You may define your own key bindings to supplment or replace these default Project Builder bindings in a private dictionary of key bindings, **PBKeyBinding.dict** in **~/Library/KeyBindings**.

Project Builder merges all key-binding dictionaries to create a composite dictionary. The order of merging is:

1. Text system's default dictionary
2. Text system's user dictionary
3. Project Builder's default dictionary
4. Project Builder's user dictionary

Because the merge processes replaces earlier bindings with later corresponding ones, the bindings in your personal dictionary take precedence over the other bindings. The merge process affects bindings of multi-key sequences, like the Control-X family of bindings, where the binding for the first key of the sequence is another dictionary which contains the bindings for the subsequent keys in the sequence. If you want to add your own Control-X binding to the existing bindings, first copy the Control-X dictionary from ProjectBuilder's **KeyBindings.dict** file and add it to your own **PBKeyBinding.dict** file. Then modify the bindings as needed. .