# NXTUIKit v1.1 Documentation

Contents

# Button APIs and Related Constants

Constants:

- #define NoButton

  A constant representing when no buttons are pressed.

APIs:

- byte PressedButton()

  Returns one of the button constants representing which button is currently pressed (BTNCENTER, BTNEXIT, BTNLEFT, or BTNRIGHT). If no buttons are pressed, returns NoButton.

- bool ButtonLongPressed(byte button)

  Returns a boolean value representing whether or not a given button has been pressed for a prolonged period of time (determined by the button state constant BTNSTATE_LONG_PRESSED_EV). Does not return until either a long button press has been determined or until the button is released.

Sample code:

```
while (true) {
	while (PressedButton() != NoButton); // Wait until no buttons are
pressed
	int button;
	while((button = PressedButton()) == NoButton); // Wait until a button is
pressed

	switch (button) {
		case BTNLEFT:
			TextOut(0, LCD_LINE1, "Left button", true);
			break;
		case BTNRIGHT:
			TextOut(0, LCD_LINE1, "Right button", true);
			break;
		case BTNCENTER:
			if (ButtonLongPressed(button))
				TextOut(0, LCD_LINE1, "Center long", true);
			else
				TextOut(0, LCD_LINE1, "Center short", true);
			break;
		case BTNEXIT:
			TextOut(0, LCD_LINE1, "Exit button", true);
			break;
	}
}
```

# Index Set APIs

Introduction:

An *index set* is an array of integers containing indexes to another array. For example, say we have an array of strings with the contents, “Larry”, “Bob”, “Madame Blueberry”, and “Junior.” A sample index set for this array might include the indexes 1 and 3, specifying the second and fourth elements of the array (in this case “Bob” and “Junior”). Note that an index set never has the same value more than once (because a value refers to a specific index of another array, of which there can only be one). Therefore, if you try to add an index to an index set, and the index set already contains that index, a duplicate is not added to the set.

APIs:

- bool IndexSetContainsIndex(int indexSet[], int index)

  Returns true if *indexSet* contains *index*, otherwise, false.

- void AddIndexToIndexSet(int &indexSet[], int index)

  Adds *index* to *indexSet*. If *indexSet* already contains *index*, it is not added to the set.

- void RemoveIndexFromIndexSet(int &indexSet[], int index)

  Removes *index* from *indexSet*, if *indexSet* contains *index*.

- void RemoveAllIndexesFromIndexSet(int &indexSet[])

  Removes all indexes from *indexSet*.

Sample code:

```
int indexSet[] = {0, 3};
string veggietales[] = {“Larry”, “Bob”, “Madame Blueberry”, “Junior”};

// Displays “Larry” and “Junior” on the screen
for (int i = 0; i < ArrayLen(indexSet); i++)
      TextOut(0, LCD_LINE1 – i * 8, veggietales[indexSet[i]], false);
Wait (SEC_2);

RemoveIndexFromIndexSet(indexSet, 3);
AddIndexToIndexSet(indexSet, 1);
ClearScreen();

// Displays “Larry” and “Bob” on the screen
for (int i = 0; i < ArrayLen(indexSet); i++)
      TextOut(0, LCD_LINE1 – i * 8, veggietales[indexSet[i]], false);
Wait (SEC_2);
```

# Text Display APIs

APIs:

- void ClearTextLine(byte y)

  Clears a line of text on the display specified by one of the LCD_LINE* constants (LCD_LINE1, LCD_LINE2, etc.).

- void CenterTextOut(byte y, string msg, byte cls)

  Displays *msg* centered on the screen.

- void RightTextOut(byte y, string msg, byte cls)

  Displays *msg* right-aligned on the screen.

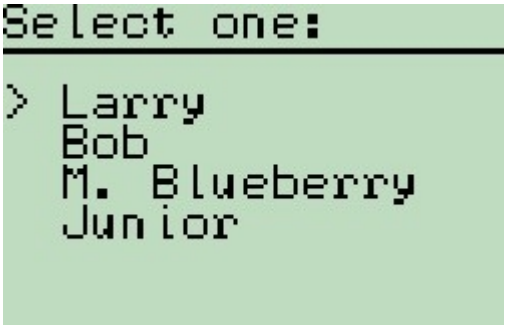
Sample code:

```
TextOut(0, LCD_LINE1, "Hello, world!", false);
TextOut(0, LCD_LINE2, "foo", false);
Wait (SEC_2);

ClearTextLine(LCD_LINE1);
CenterTextOut(LCD_LINE1, "Centered text", false);
Wait (SEC_2);

ClearTextLine(LCD_LINE2);
RightTextOut(LCD_LINE2, "Right-aligned", false);
Wait (SEC_2);
```

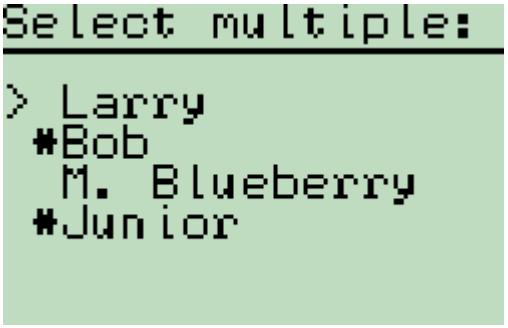
# Text Menu APIs and Related Constants and Structures

Introduction:

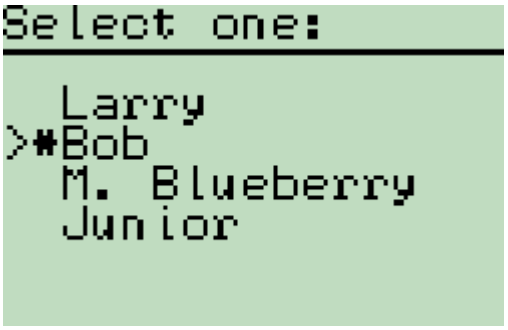
Text menus are simply selectable menus of text. Their appearance looks something like this:




They consists of a prompt (in this case, "Select one:") and a list of selectable text items. The default behavior allows you to select only one item on the list, but text menus also allow multiple selection and something called static selection. Multiple selection looks like this:




It allows the selection of multiple items by pressing the center orange button for a prolonged period of time (about two seconds, determined by the LongButtonPressed API). This takes you into multiple selection mode. To cancel the selection of multiple items, simply press the gray back button. You will then be taken back into single selection mode. To confirm a selection in multiple selection mode, press and hold the center button for about two seconds (again, determined by LongButtonPressed). *Note that this will also select whichever item the arrow is currently positioned at.* Text menus also support something called static selection. Static selection behaves similarly to but differently than both single and multiple selection. It looks like this:



Static selection allows only one item to be selected. Unlike single selection, it does not exit when an item is selected, and like multiple selection, it shows the selection indicator (an asterisk). The reason it is called "static" is because the selection is meant to be preserved. In other words, when you select an item of a static text menu and come back to it later, the same item should still be selected.

Structures:

- TextMenu
    - string prompt

      The prompt to display above the text menu. Note that this is limited to a length of 16 characters.

    - bool allowsMultipleSelection

      A boolean value determining if multiple selection is allowed.

    - bool showsSelectionIndicator

      A boolean value determining if the selection indicator (asterisk) is shown for menus that do not support multiple selection. Note that this value really determines if a text menu supports static selection (in which case, true) or regular single selection (false).

    - int selectedRowIndexes[]

      An index set, which, upon return, contains the indexes of the selected items in the text menu. For static selection and multiple selection, it is perfectly valid to populate this array beforehand with values you already want to be selected.

    - string stringValues[]

      The string array whose contents will be displayed as selectable text items on the screen.

Constants:

- TextMenuOutConfirmedReturn

  A constant representing that a selection was made.

- TextMenuOutCancelledReturn

  A constant representing that a selection was not made.

- TextMenuOutAlternateReturn

  A special constant used by static menus representing that a selection was made with a long button press.

APIs:

- TextMenuOutReturnType TextMenuOut(TextMenu &textMenu)

Displays a text menu on the screen, returning one of the TextMenuOutReturnType constants (TextMenuOutConfirmedReturn, etc.).

Sample code:

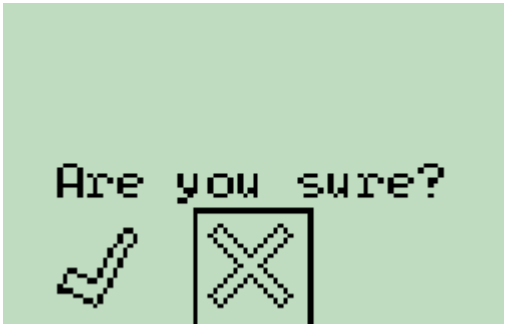
```
TextMenu singleSelectionTextMenu, multiSelectionTextMenu, staticSelectionTextMenu;
string menuOptions[] = {"Larry", "Bob", "M. Blueberry", "Junior"};

singleSelectionTextMenu.prompt = “Select one:”;
singleSelectionTextMenu.stringValues = menuOptions;

if (TextMenuOut(singleSelectionTextMenu)) {
      NumOut(0, LCD_LINE1, singleSelectionTextMenu.selectedRowIndexes[0], true);
      Wait (SEC_2);
}

multiSelectionTextMenu.allowsMultipleSelection = true;
multiSelectionTextMenu.prompt = "Select multiple:";
multiSelectionTextMenu.stringValues = menuOptions;

if (TextMenuOut(multiSelectionTextMenu)) {
      ClearScreen();
      for (int i = 0; i < ArrayLen(multiSelectionTextMenu.selectedRowIndexes); i++)
            NumOut(0, LCD_LINE1 - i * 8,
multiSelectionTextMenu.selectedRowIndexes[i], false);
      Wait (SEC_2);
}

staticSelectionTextMenu.showsSelectionIndicator = true;
staticSelectionTextMenu.prompt = “Select one:”;
staticSelectionTextMenu.stringValues = menuOptions;
int alreadySelectedIndexes[] = {1};
staticSelectionTextMenu.selectedRowIndexes = alreadySelectedIndexes;

TextMenuOutReturnType retVal = TextMenuOut(staticSelectionTextMenu);
if (retVal == TextMenuOutConfirmedReturn) {
      NumOut(0, LCD_LINE1, staticSelectionTextMenu.selectedRowIndexes[0], true);
      Wait (SEC_2);
} else if (retVal == TextMenuOutAlternateReturn) {
      TextOut(0, LCD_LINE1, “Static alternate”, true);
      Wait (SEC_2);
}
```

# Various UI APIs

APIs:

- bool ConfirmationDialogOut(string messageText)

  A confirmation dialog looks like the following:




It is the same screen that appears when turning off the NXT or deleting a file. The text displayed above the dialog is arbitrary and can be set to a custom string value. The function returns a boolean value reflecting the user's selection.

- bool TextViewOut(int y1, int y2, string msg)

A text view is a portion of the screen in which text is rendered across multiple lines. Text views support multiple pages of text when one page is not enough. To navigate through a text view, use the left and right buttons on the NXT. Note that text views do *not* clear the entire screen. Rather, they clear only the portion of the screen in which text is displayed. TextViewOut returns a boolean value reflecting if the center button was pressed to exit the view (in which case, true) or if the gray button was pressed (false).

Sample code:

```
string text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque
laoreet, quam vel accumsan feugiat, mi est ultricies quam, in iaculis diam velit ac
tortor. Sed sit amet rhoncus dui. Fusce ut massa eu nisl aliquam viverra. Phasellus
lacinia, leo ut cursus mattis, tortor turpis ullamcorper augue, eget tincidunt
massa erat ut libero. Nulla accumsan nisl mi. Vestibulum justo turpis, aliquam a
suscipit et, vehicula a quam.";


if (TextViewOut(LCD_LINE3, LCD_LINE7, text)
      TextOut(0, LCD_LINE1, "Center button", true);
else
      TextOut(0, LCD_LINE1, "Back button", true);


Wait (SEC_2);
```

- void ProgressIndicatorOut(int y, int numerator, int denominator)

A progress indicator, or what is better known as a loading bar, is used to indicate the progress of a given task. It looks like the following:

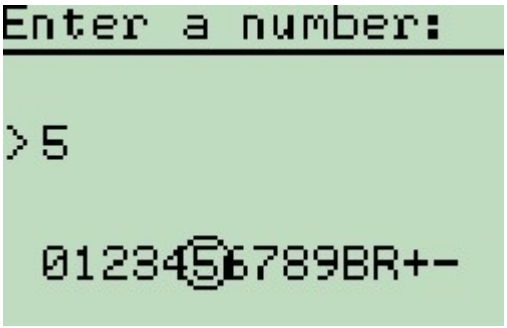

It is meant to be used within a loop where the current loop iteration and the number of total iterations is known. These two values are used to compute the percentage of the task that has been performed.

Sample code:

```
// Starting position
ProgressIndicatorOut(LCD_LINE3, 0, 7);


for (int i = 0; i < 7; i++) {
	// Execute some task (in this case, wait 1 second)
	Wait(SEC_1);
	// Update progress indicator to reflect the number of tasks that have been performed
	ProgressIndicatorOut(LCD_LINE3, i + 1, 7);
}


Wait(SEC_2);
```

- bool NumberInputDialogOut(string prompt, int startingLocation, unsigned long &number)

A number input dialog allows the user to input a number. It looks like this (credit to Brian Davis for the interface):



To input a number, move the circular cursor to the next digit of the number (using the NXT's left and right buttons), and then press the center button. The four last symbols (*B*, *R*, +, -) are used to erase the last digit of the number, return the inputted number (confirm the input), increment the number, and decrement the number, respectively.

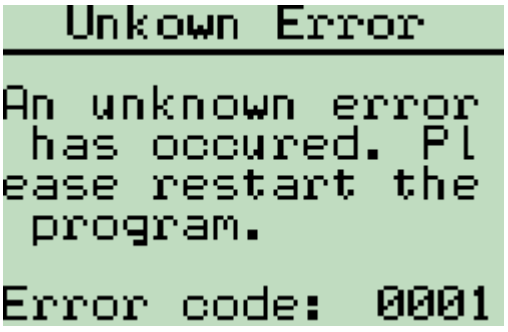
Besides the obvious *number* parameter, which contains the result of the dialog and which can be set to an initial value, NumberInputDialogOut takes two other parameters, *prompt* and *startingLocation*. *prompt* is simply the line of text above the dialog ("Enter a number:"), while *startingLocation* represents the cursor's initial location. Valid constants for *startingLocation* are the numbers 0 through 9 (which represent the same digits on the screen) and NumberInputDialogStartingLocationBack, NumberInputDialogStartingLocationReturn, NumberInputDialogStartingLocationPlus, and NumberInputDialogStartingLocationMinus (which represent *B*, *R*, +, and -, respectively). Returns a boolean value representing whether the input was confirmed (true) or canceled (false).

Sample code:

```
unsigned long number = 0;

if (NumberInputDialogOut("Enter a number:",
NumberInputDialogStartingLocationReturn, number))

      NumOut(0, LCD_LINE3, number, true);

else

      TextOut(0, LCD_LINE3, "Canceled!", true);


Wait (SEC_2);
```

- void ErrorAlertOut(string header, string messageText, unsigned int errorCode)

An error alert provides for the display of error content on the screen as an alert. It takes the following form:




ErrorAlertOut has three components: a header, message text, and error code. To accommodate for long message texts, a text view is used to display that portion of the alert (which, of course, allows paging).

Sample code:

```
bool errorHasOccured = true;

unsigned int errorCode = 1;


if (errorHasOccured) {

      ErrorAlertOut(“Unknown Error”, “An unknown error has occurred. Please restart the program. If that is not an option, continue to run the program in an inconsistent state.”, errorCode);

} else {

      // Continue normal execution

}
```