Developer’s Guide for BlackBerry™ and RIM Wireless Handhelds™

UI Engine API
Version 2.1
Contents

Chapter 1
Introduction ........................................................................................................ 5
About this guide .......................................................................................... 5

Chapter 2
Classes ................................................................................................................. 7
Base classes ............................................................................................. 9
Field ....................................................................................................... 9
FieldManager ...................................................................................... 10
Menu .................................................................................................... 10
Screen ................................................................................................... 10
Status .................................................................................................... 11
UIEngine .............................................................................................. 11

Chapter 3
API descriptions .............................................................................................. 13
BeepChoice class ................................................................................. 13
BusyStatus class .................................................................................. 15
Choice class .......................................................................................... 18
ClickDialog class ................................................................................. 26
DecimalEdit class ............................................................................... 29
Dialog class ........................................................................................... 30
Edit class ............................................................................................... 34
Field class .............................................................................................. 45
FieldManager class ............................................................................ 48
Label class ............................................................................................. 53
List class ................................................................................................. 54
Menu class ............................................................................................. 60
OKDialog class .................................................................................... 65
RichText class ...................................................................................... 67
Structures ............................................................................................ 69
Screen class ........................................................................................... 70
Separator class ..................................................................................... 77
Status class ............................................................................................ 79
StringList class .................................................................................... 83
Table class ............................................................................................. 85
Title class ............................................................................................. 100
TopBottomChoice class .................................................................. 102
UIEngine class ................................................................................... 104
Index of APIs ..............................................................111
Index ..............................................................113
1

Introduction

The User Interface Engine (UI Engine) API is a set of classes that provides a common look and feel for all applications developed for the RIM Wireless Handheld. Features include menu and screen handling, user fields and dialog boxes. This functionality works to complement and enhance the base functionality of the Operating System (OS) API.

In using this guide, it is assumed that you are familiar with the RIM Wireless Handheld OS API.

About this guide

This guide includes general information on the UI Engine API, as well as a complete listing of all its functions. The functions appear in alphabetical order.

<table>
<thead>
<tr>
<th>Note</th>
<th>Notes provide additional information to help complete a task. Tips offer an alternative method of performing an action, or background information you might find interesting.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning!</td>
<td>Warnings follow any procedure or paragraph containing instructions that, if followed improperly, could result in damaging the device or software.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names of program groups, icons, folders, windows, menu items, and labels of other interface objects</td>
<td>Research In Motion program group</td>
</tr>
<tr>
<td>Key names</td>
<td>ALT</td>
</tr>
<tr>
<td>Programming elements such as returns, events, constants, structures, parameters</td>
<td>COMM_RX_ERROR</td>
</tr>
<tr>
<td>Function names when set apart from a class name or code snippet.</td>
<td>RimGetMessage</td>
</tr>
</tbody>
</table>
2

Classes

The following classes are defined in the UI Engine. Public member functions are defined in the header files by the heading “public: // used by client.” Indentation in this table indicates subclassing.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Start on page</th>
<th>Header file</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIEngine class</td>
<td>104</td>
<td>uiengine.h</td>
</tr>
<tr>
<td>Menu class</td>
<td>60</td>
<td>menu.h</td>
</tr>
<tr>
<td>Status class</td>
<td>79</td>
<td>status.h</td>
</tr>
<tr>
<td>BusyStatus class</td>
<td>15</td>
<td>busystatus.h</td>
</tr>
<tr>
<td>FieldManager class</td>
<td>48</td>
<td>manager.h</td>
</tr>
<tr>
<td>Screen class</td>
<td>70</td>
<td>screen.h</td>
</tr>
<tr>
<td>Dialog class</td>
<td>30</td>
<td>dialog.h</td>
</tr>
<tr>
<td>ClickDialog class</td>
<td>26</td>
<td>clickdialog.h</td>
</tr>
<tr>
<td>OKDialog class</td>
<td>65</td>
<td>okdialog.h</td>
</tr>
<tr>
<td>Field class</td>
<td>45</td>
<td>field.h</td>
</tr>
<tr>
<td>Choice class</td>
<td>18</td>
<td>choice.h</td>
</tr>
<tr>
<td>BeepChoice class</td>
<td>13</td>
<td>choice.h</td>
</tr>
<tr>
<td>TopBottomChoice class</td>
<td>102</td>
<td>choice.h</td>
</tr>
<tr>
<td></td>
<td>109</td>
<td>choice.h</td>
</tr>
<tr>
<td>Edit class</td>
<td>34</td>
<td>edit.h</td>
</tr>
<tr>
<td>DecimalEdit class</td>
<td>29</td>
<td>edit.h</td>
</tr>
<tr>
<td>Label class</td>
<td>53</td>
<td>label.h</td>
</tr>
<tr>
<td>Title class</td>
<td>100</td>
<td>label.h</td>
</tr>
<tr>
<td>List class</td>
<td>54</td>
<td>list.h</td>
</tr>
<tr>
<td>StringList class</td>
<td>83</td>
<td>list.h</td>
</tr>
</tbody>
</table>
### Classes

<table>
<thead>
<tr>
<th>Classes</th>
<th>Start on page</th>
<th>Header file</th>
</tr>
</thead>
<tbody>
<tr>
<td>RichText class</td>
<td>67</td>
<td>richtext.h</td>
</tr>
<tr>
<td>Separator class</td>
<td>77</td>
<td>separator.h</td>
</tr>
<tr>
<td>Table class</td>
<td>85</td>
<td>table.h</td>
</tr>
</tbody>
</table>

The base classes are described in the next section.
Base classes

The UI Engine contains seven base classes. Some of the base classes have no derived classes.

Field

A Field is a screen object that accepts user data. Fields are peers to one another, and they are always contained within a screen.

A number of classes inherit from the Field class, dealing with different types of user data and different ways of handling the data.
**FieldManager**

The FieldManager class is an abstract base class, introduced in v2.0 of the SDK. It defines methods and functions for managing fields. Since fields are common throughout the user interface, a number of other classes inherit from FieldManager.

![Class Diagram]

**Menu**

The Menu class handles menu interactions. It has no derived classes.

![Menu Class]

**Screen**

A screen is a rectangular container for a list of fields. It isn't actually a base class—it inherits from the FieldManager class—but it was a base class in editions of the SDK before 2.0.

The Screen class has no classes derived from it.

![Screen Class]
Status
The Status class supports a single derived class, BusyStatus:

```
.Status -> BusyStatus
```

UIEngine
The UIEngine root class has no derived classes. The UIEngine class is responsible for handling the display of screen elements; an instance of the UIEngine class must be initialized for an application to use the UI engine.
Classes

Base classes
3

API descriptions

The API descriptions are organized by class, alphabetically.

BeepChoice class

BeepChoice is a class derived from Choice. It is tied directly to the OS function RimTestAlert.

BeepChoice

BeepChoice::BeepChoice ......................................................................................................... 13
BeepChoice::GetBeep ............................................................................................................. 13
BeepChoice::GetFieldType ................................................................................................. 14
BeepChoice::SetBeep ............................................................................................................ 14

BeepChoice::BeepChoice

Constructor for the BeepChoice class

BeepChoice( 
    char * Label,
    int CurrentChoice )

Parameters

Label
    The label associated with the choice box.

CurrentChoice
    The same as the first parameter in RimTestAlert

Remarks

These are constructors.

BeepChoice::GetBeep

Gets the value displayed in the BeepChoice field

int GetBeep()

Return value

The same as the first parameter in RimTestAlert
BeepChoice::GetFieldType

Returns the type of the current field

```cpp
FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL ) ;
```

**Parameters**

- `pDerived`: A pointer to a `FIELDTYPE` variable that will receive the specific derived type of the current field. This parameter is optional. It can be `NULL`. If the parameter is `NULL`, the function returns no information about the specific derived type.

**Return value**

Returns the type of the base class of the current field (i.e. the type of the first class derived from the `Field` object)

**Remarks**

For a `BeepChoice` object, the function will return `CHOICE_FIELD`, and `pDerived` will be filled in with `BEEP_CHOICE_FIELD`.

BeepChoice::SetBeep

Sets the value displayed in the BeepChoice field

```cpp
SetBeep(int newBeepNumber)
```

**Parameters**

- `newBeepNumber`: The same as the first parameter in `RimTestAlert`
BusyStatus class

A BusyStatus object can be used during long operations to inform the user that work is being done. It looks like a regular status box, but it contains an animated hourglass bitmap, and can be held on the screen until the operation is complete rather than for a fixed period of time.

Using BusyStatus helps prevent the OS from throwing the watchPuppy exception. For example, you might first declare a busy status:

BusyStatus myBusyStatus("Calculating pi, Captain...", 100);

And then, in a time-consuming loop, call Draw():

for (...) {
    timeconsuming code
    myBusyStatus.Draw()
}

Note: BusyStatus is optimized so that it does not display every time you call draw, only every 500 ms (1/2 a second). If you just declare one of these and draw it once, it doesn't show up, because there is no delay between calls.

BusyStatus

BusyStatus::BusyStatus (char * InitString = NULL, int InitTime = 10)

Parameters

InitString

The string that should be displayed in the busy status box. If this parameter is NULL, no string will be displayed.

InitTime

The length of time that the busy status box will remain on the screen after a call to Draw() before being removed by the UI Engine (in 1/100 second intervals)
BusyStatus class

Remarks
The BusyStatus object will be displayed when Draw() is called, and it will remain on the screen until the specified time interval elapses, or until Draw() is called again to render something else. Typically, an application will call Draw() many times throughout the course of an intensive operation. The busy status box remains displayed throughout.

**BusyStatus::Disable**
Disables the BusyStatus box

```cpp
void Disable()
```

Remarks
The busy status box will remain disabled until a subsequent call to Reset(), ResetBitmap(), or SetText() is made.

**BusyStatus::Draw**
Draws the busy status box for the first time or holds it on the screen

```cpp
void Draw()
```

Remarks
The BusyStatus object will be displayed when Draw() is called, and it will remain on the screen until the specified time interval elapses or until Draw() is called again. Typically, an application will call Draw() many times throughout the course of an intensive operation. The busy status box remains displayed throughout. Each time Draw() is called, the UI Engine decides whether or not to advance the bitmap to the next bitmap in the animation sequence. This is based on the time elapsed since the last time the bitmap was advanced.

**BusyStatus::Reset**
Resets the busy status box with a new string and a new display time

```cpp
void Reset(
    char *InitString = NULL,
    int InitTime = 10)
```

Parameters
- **InitString**
  The string that should be displayed in the busy status box. If this parameter is NULL, no string will be displayed.
- **InitTime**
  The length of time that the busy status box will remain on the screen after a call to Draw() before being removed by the UI Engine (in 1/100 second intervals)
Remarks  The BusyStatus object will be displayed when Draw() is called, and it will remain on the screen until the specified time interval elapses, or until Draw() is called again. Typically, an application will call Draw() many times throughout the course of an intensive operation. The busy status box remains displayed throughout.

**BusyStatus::ResetBitmap**

Resets the bitmap to the first bitmap in the animation sequence

```cpp
void ResetBitmap()
```
Choice class

The functions on the following pages are listed in alphabetical order.

Choice

Choice::ChangeChoice ................................................................. 19
Choice::ChangeChoiceEx ............................................................. 20
Choice::ChangeChoiceList ............................................................ 21
Choice::Choice ........................................................................... 18
Choice::GetFieldType ................................................................. 21
Choice::GetLabel ....................................................................... 22
Choice::GetNumEntries ............................................................... 22
Choice::GetSelectedIndex .......................................................... 22
Choice::GetSelectedValue ............................................................ 22
Choice::GetValueAtIndex ............................................................. 23
Choice::SetChoices ................................................................... 23
Choice::SetLabel ....................................................................... 23
Choice::SetNumEntries ............................................................... 24
Choice::SetNumericString ........................................................... 24
Choice::SetSelectedIndex ........................................................... 24
Choice::UpdatePtr ..................................................................... 25

Choice::Choice

Constructors for the Choice class

Form 1: Choice()

Form 2: Choice (char const * const pnewLabel,
char const * const * const newStringArray,
int const newIndex=0, int const Justify=LCD_RIGHT_JUSTIFIED)

Form 3: Choice(char const * const pnewLabel,
unsigned int const newStartValue,
unsigned int const newEndValue,
int const newIndex=0,
int const Increment=1,
int const Justify=LCD_RIGHT_JUSTIFIED)

Parameters

pnewLabel
The pointer to the choice box label. Labels are left justified.

pnewStringArray
A pointer to a string array. This string is displayed right justified.

NewStartValue
The integer values to start a range

NewEndValue
The integer value to end a range

NewIndex
The starting value within the above range
Increment
The increment value in the above range

Justify
The display justification of the data portion of a choice box

Remarks
These are constructors.

Choice boxes are effectively one line text boxes. There are two ways to trigger the UI Engine to change the displayed data: simultaneously pressing the ALT key and rolling, or using the ChangeChoice member function. The application passes data to the UI Engine in 2 ways:

- Pass a pointer to a string array and the offset in the array for the initial value. When the user performs ALT+ROLL, the next value in the array will be displayed (rolling down will set the next value, rolling up will display the previous value).

- Pass an integer range and initial value. When the user performs ALT+ROLL, the next value in the range will be displayed (rolling down will set the next value, rolling up will display the previous value).

Choice::ChangeChoice
Displays a dialog box on the screen that allows the user to use the trackwheel to modify the current setting in the Choice box

RESULT ChangeChoice(
    UIEngine &newCallingUI,
    void ((*CallbackFunction)(int Value)) = NULL)

Parameters
NewCallingUI
The UIEngine object associated with this choice box

(*CallbackFunction)(int Value)
If the value is not NULL, this function is called each time the associated ChoiceBox value is changed. (The Value parameter of the callback function is the index of the currently selected choice item).

(*func)(int Value, void * tag)
A callback function called when the choice value changes. The Value parameter is passed in as the tag value (and hence can be used for context).

Return value
One of UNHANDLED (implies that the BACKSPACE key was hit) or CLICKED (the user hit either the ENTER key or clicked the trackwheel).
Choice class

Remarks
One way the user changes the Choice Box choice is by hitting ALT+ROLL. ChangeChoice is an alternative which, when invoked, places a Dialog on the screen that uses another Choice Box (using the same data) as the field in the Dialog. Rolling up and down changes the data associated with the Choice Box. Each time a change is made, the UI Engine calls the CallBack Function.

**Choice::ChangeChoiceEx**
Displays a dialog box on the screen that allows the user to use the trackwheel to modify the current setting in the Choice box

```
RESULT ChangeChoiceEx(
    UIEngine &newCallingUI,
    void (*func)(int Value, void * tag),
    void * tagValue)
```

**Parameters**

- **NewCallingUI**
  The UIEngine object associated with this choice box

- **(*CallbackFunction)(int Value)**
  If the value is not NULL, this function is called each time the associated Choice box value is changed. (The Value parameter of the callback function is the index of the currently selected choice item).

- **(*func)(int Value, void * tag)**
  A callback function called when the choice value changes. The Value parameter is passed in as the tag value (and hence can be used for context).

- **TagValue**
  Passed to the callback function as the tag value

**Return value**
One of UNHANDLED (implies the BACKSPACE key was hit) or CLICKED (implies the user hit either the ENTER key or clicked the trackwheel).

**Remarks**
One way the user changes the Choice Box choice is by hitting ALT+ROLL. ChangeChoiceEx is an alternative which, when invoked, places a Dialog on the screen that uses another Choice Box (using the same data) as the field in the Dialog. Rolling up and down changes the data associated with the Choice Box. Each time a change is made, the UI Engine calls the CallBack Function.
**Choice::ChangeChoiceList**

Displays a dialog box on the screen that allows the user to use the trackwheel to modify the current setting in the Choice box.

```
RESULT ChangeChoiceList(
    UIEngine &newCallingUI,
    void (*func)(int Value, void * tag),
    void * tagValue)
```

- **Parameters**
  - `NewCallingUI`: The UIEngine object associated with this choice box.
  - `(*CallbackFunction)(int Value)`
    - If the value is not NULL, this function is called each time the associated ChoiceBox value is changed. (The Value parameter of the callback function is the index of the currently selected choice item).
  - `(*func)(int Value, void * tag)`
    - A callback function called when the choice value changes. The Value parameter is passed in as the tag value (and hence can be used for context).
  - `TagValue`
    - Passed to the callback function as the tag value.

- **Return value**
  - One of UNHANDLED (implies the BACKSPACE key was hit) or CLICKED (implies the user hit either the ENTER key or clicked the trackwheel).

- **Remarks**
  - One way the user changes the Choice Box choice is by hitting ALT+ROLL. ChangeChoiceList is an alternative which, when invoked, places a Dialog on the screen that uses another Choice Box (using the same data) as the field in the Dialog. Rolling up and down changes the data associated with the Choice Box. Each time a change is made, the UI Engine calls the CallbackFunction.

**Choice::GetFieldType**

Returns the type of the current field.

```
FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL )
```

- **Parameters**
  - `pDerived`: A pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. This parameter is optional. It can be NULL. If the parameter is NULL, the function returns no information about the specific derived type.

- **Return value**
  - Returns the type of the base class of the current field (i.e. the type of the first class derived from the Field object).
API descriptions

Choice class

Remarks
For a Choice object, the function will return CHOICE_FIELD, and pDerived will be filled in with NULL_FIELD (since Choice is derived directly from Field).

Choice::GetLabel
Returns a pointer to the text in the label currently displayed beside the Choice field
char const * const GetLabel()

Return value
The label associated with choice box

Choice::GetNumEntries
Retrieves the number of entries in the choice field
int GetNumEntries() const

Return value
Returns the number of entries associated with the choice field.

Choice::GetSelectedIndex
Returns the index of the currently selected item in the Choice field
int GetSelectedIndex() const

Return value
The index of either the string array or index of the value, depending on which constructor is used

Remarks
This function returns the value selected by the user.

Choice::GetSelectedValue
Returns the numerical value or a pointer to the string currently selected in the Choice box
Form 1: int GetSelectedValue (void)
Form 2: void GetSelectedValue (const char * &String)

Parameters
String
A reference to a character pointer variable that will receive a pointer to the currently selected string

Return value
In Form 1, it returns the value of the currently-selected integer. In Form 2, no return value
**Choice::GetValueAtIndex**

Returns the string associated with the index

```
Char const * GetValueAtIndex(int IndexValue)
```

**Parameters**

- **IndexValue**
  - The index to query for its value

**Return value**

Returns the character string associated with the index

**Choice::SetChoices**

Sets the list of choices that should be displayed in the Choice field either as a range of numeric values or as an array of NULL-terminated strings

**Form 1**:
```
void SetChoices (char const * const * const newStringArray,
                int const newIndex=0)
```

**Form 2**:
```
void SetChoices (unsigned int const newStartValue,
                unsigned int const newEndValue,
                int const newIndex=0,
                int const Increment = 1)
```

**Parameters**

- **newStringArray**
  - The string array containing choice box entries
- **newIndex**
  - The index into the currently selected entry associated with **newIndex**
- **newStartValue**
  - The start value
- **newEndValue**
  - The end value
- **Increment**
  - The increment value in the range between the start value and the end value

**Choice::SetLabel**

Sets the text in the label that will be displayed beside the Choice field

```
void SetLabel(char const * const pnewLabel)
```

**Parameters**

- **pnewLabel**
  - The label associated with the choice box
**Choice::SetNumEntries**
Sets the number of entries available in the choice box

```cpp
void SetNumEntries ( int const NumEntries ) ;
```

**Parameters**
- **NumEntries**
  The number of entries that should be available to the user

**Remarks**
Calling this function will not change the values of the entries in the choice box. **NumEntries** specifies the number of entries that should be available, starting with the start of the range of integer values specified, or with the first string specified. For example, if the range specified is 2-10, and this function is called with **NumEntries** set to 3, the values available in the choice box will be 2, 3, and 4.

**Choice::SetNumericString**
Allows the user to enter a number without pressing control.

```cpp
void SetNumericString(bool Numeric=true)
```

**Parameters**
- **Numeric**
  If **TRUE**, the control key is not required to enter a number

**Remarks**
If **numeric** is set to **TRUE**, then a user may select a choice by entering the associated number without first activating the control key.

**Choice::SetSelectedIndex**
Instructs the UI Engine to display the item with the specified index in the Choice field

```cpp
void SetSelectedIndex(int const newIndex)
```

**Parameters**
- **newIndex**
  Sets the index of either the string array or the index of the value, depending on what constructor is used

**Remarks**
This function sets the index of the choice box
**Choice::UpdatePtr**

Updates RAM pointers associated with flash data

```cpp
void UpdatePtr(
    char const * const * const pData,
    char const * const pLabel = NULL)
```

**Parameters**

- **pData**
  
  The data to update in flash. If NULL, the data is assumed the same as before and is updated accordingly.

- **pLabel**
  
  A label associated with the choice field. If NULL, the data is assumed the same as before and is updated accordingly.
ClickDialog class

ClickDialog is a class derived from Dialog. A ClickDialog object is a dialog box that is dismissed automatically when the user clicks the trackwheel or presses any key.

ClickDialog

ClickDialog::ClickDialog ....................................................................................................... 26
ClickDialog::Go ................................................................................................................ ........... 27
ClickDialog::SetAnchorPoint .................................................................................................... 27
ClickDialog::SetBitmap ......................................................................................................... ..... 27
ClickDialog::SetDisplayString .................................................................................................. 28

ClickDialog::::ClickDialog

Constructors for the ClickDialog class

Form 1: ClickDialog ()

Form 2: ClickDialog (  
char const * const newpDisplayString, BitMap  
const * const newpBitmap = NULL )

Form 3: ClickDialog(  
char const * const newpDisplayString,  
Bitmaps::PREDEFINED_BITMAP const PredefinedBitmap

Parameters

newpDisplayString  
Pointer to the text string displayed in the dialog box

newpBitmap  
Pointer to a bitmap structure that is to be displayed in the dialog box

PredefinedBitmap  
One of the predefined bitmap shapes specified in PredefinedBitmaps.h:
- BITMAPS::INFORMATION
- BITMAPS::QUESTION
- BITMAPS::EXCLAMATION
- BITMAPS::HOURGLASS

Remarks

The display string and bitmap are placed in an application modal dialog box on the display when the member function DisplayDialog() is called. The dialog will be cleared automatically when the user clicks the trackwheel or presses any key.
**ClickDialog::Go**
Activates the dialog

```cpp
int Go (UIEngine & UIEngineToUse)
```

**Parameters**
- **UIEngineToUse**
  The UIEngine to use for rendering the component

**Return value**
The return value is always zero

**Remarks**
The specified display string and bitmap will be placed in an application modal dialog box on the display. The dialog will be cleared automatically when the user clicks the trackwheel or presses any key.

**ClickDialog::SetAnchorPoint**
Sets the number of pixels from the top of the screen at which the dialog box should be displayed

```cpp
void SetAnchorPoint ( int AnchorPoint ) ;
```

**Parameters**
- **AnchorPoint**
  The number of pixels from the top of the screen at which the dialog box should be displayed

**Remarks**
If there is not enough room on the display to show the entire dialog box starting at AnchorPoint, nothing will be displayed.

**ClickDialog::SetBitmap**
Sets the bitmap that will be displayed in the click dialog box

- **Form 1:** void SetBitmap (BitMap const * const Bitmap)
- **Form 2:** void SetBitmap (BitMaps::PREDEFINED_BITMAP const PredefinedBitmap)

**Parameters**
- **Bitmap**
  The pointer to a bitmap (custom or predefined)
- **PredefinedBitmap**
  One of the predefined bitmap shapes specified in PredefinedBitmaps.h:
  - BITMAPS::INFORMATION
  - BITMAPS::QUESTION
  - BITMAPS::EXCLAMATION
  - BITMAPS::HOURGLASS

**Remarks**
The Bitmap is placed in a dialog box on the display.
**ClickDialog::SetDisplayString**

Sets the text that will be displayed in the click dialog box

```cpp
void SetDisplayString(const char * const newpDisplayString)
```

**Parameters**

- `newpDisplayString`
  - The pointer to the string associated with the status box
DecimalEdit class

DecimalEdit is a class derived from Edit.

**Note**
If the ALT key is held while an alpha key is typed when this field has the focus, and the key is numeric, the UI Engine will display the secondary function character (the character displayed above the numeric key).

The functions on the following pages are listed in alphabetical order.

**DecimalEdit**
DecimalEdit::DecimalEdit ................................................................. 29
DecimalEdit::GetNumber ................................................................. 29
DecimalEdit::SetNumber ................................................................. 29

**DecimalEdit::DecimalEdit**
Constructor for the DecimalEdit class

Form 1: DecimalEdit()
Form 2: DecimalEdit (char * pLabel,
int Current,
int const Justify=LCD_RIGHT_JUSTIFIED)

**Parameters**
- pLabel: The label associated with the edit buffer
- Current: The initial number associated with the buffer
- Justify: The justification

**DecimalEdit::GetNumber**
Returns the numeric value currently displayed in the DecimalEdit field

int GetNumber ()

**Return value**
The current number in the DecimalEdit field

**DecimalEdit::SetNumber**
Sets the numeric value to be displayed in the DecimalEdit field

void SetNumber(int Number )

**Parameters**
- Number: The number to be placed in the DecimalEdit field
Dialog class

The Dialog class inherits from the FieldManager class.

**Dialog**

Dialog::ClearDialog ............................................................................................................ 31  
Dialog::Dialog ................................................................................................................... 30  
Dialog::DisplayDialog ....................................................................................................... 31  
Dialog::SetAnchorPoint ................................................................................................. 31  
Dialog::SetBitmap ........................................................................................................... 32  
Dialog::SetDisplayString ............................................................................................... 32  
Dialog::SetField ............................................................................................................. 32  
Dialog::UpdatePtr ............................................................................................................. 33

### Dialog::Dialog

Constructors for the Dialog class

Form 1: Dialog ()

Form 2: Dialog ( 
  char const * const newpDisplayString,  
  BitMap const * const newpBitmap=NULL,  
  Field * const pField=NULL )

Form 3: Dialog ( 
  char const * const newpDisplayString,  
  Bitmaps::PREDEFINED_BITMAP const PredefinedBitmap  
  Field * const pField=NULL )

Form 4: Dialog ( const Dialog & src )

**Parameters**

- **newpDisplayString**
  The pointer to the string associated with the dialog box

- **newpBitmap**
  The pointer to a bitmap (custom or predefined)

- **pField**
  A pointer to a field (typically edit or list)

- **PredefinedBitmap**
  One of the predefined bitmap shapes specified in
  PredefinedBitmaps.h:
  - BITMAPS::INFORMATION
  - BITMAPS::QUESTION
  - BITMAPS::EXCLAMATION
  - BITMAPS::HOURGLASS

**src**
An initialized Dialog object

**Remarks**
These are the constructors.
The `DisplayString`, `Bitmap` and `Field` are placed in an application modal dialog box on the display. The following is an example of a dialog box with the relative position of the `DisplayString`, `Bitmap`, and a `Field`.

Form 4 instantiates a new `Dialog` object that’s a duplicate of the `src` parameter.

**Dialog::ClearDialog**
Removes the dialog from the display

```cpp
void ClearDialog ()
```

Remarks: This function clears the dialog box from the display.

**Dialog::DisplayDialog**
Instructs the UI Engine to draw the dialog on the display

```cpp
void DisplayDialog ()
```

Remarks: This function displays the dialog box on the display. If you wish, it can be overridden to provide a custom display method, but the implementation must call the line:

```cpp
uiengine.ProcessDialog(mydialog)
```

where `mydialog` is the dialog you want to display.

**Dialog::SetAnchorPoint**
Sets the number of pixels from the top of the screen at which the dialog box should be displayed

```cpp
void SetAnchorPoint ( int AnchorPoint ) ;
```

Parameters: `AnchorPoint`

The number of pixels from the top of the screen at which the dialog box should be displayed

Remarks: If there is not enough room on the display to show the entire dialog box starting at `AnchorPoint`, nothing will be displayed.
Dialog::SetBitmap
Sets the bitmap that will be displayed in the dialog box
Form 1: void SetBitmap (BitMap const * const Bitmap)
Form 2: void SetBitmap ( 
    BitMaps::PREDEFINED_BITMAP
    const PredefinedBitmap)

Parameters
   Bitmap
      The pointer to a bitmap (custom or predefined)
   PredefinedBitmap
      One of the predefined bitmap shapes specified in 
      PredefinedBitmaps.h:
      • BITMAPS::INFORMATION
      • BITMAPS::QUESTION
      • BITMAPS::EXCLAMATION
      • BITMAPS::HOURGLASS

Remarks
   The Bitmap is placed in a dialog box on the display.

Dialog::SetDisplayString
Sets the text that will be displayed in the dialog box
void SetDisplayString(char const * const newpDisplayString )

Parameters
   newpDisplayString
      The pointer to the string associated with the status box

Dialog::SetField
Sets the field that will be displayed in the dialog box
void SetField(
    Field &newpField ,
    int const newFieldHeight = -1,
    int const newFieldWidth = -1)

Parameters
   newpField
      The field (typically Edit or List) that is part of the dialog
   newFieldHeight
      The height of the field
   newFieldWidth
      The width of the field
Dialog::UpdatePtr

Updates the RAM pointers associated with flash data

```cpp
void UpdatePtr(
    char const * const newpDisplayString,
    BitMap const * const newpBitmap = NULL)
```

**Parameters**
- **newpDisplayString**
  - The new display string
- **newpBitmap**
  - The bitmap to display. If NULL, any existing bitmap data is used.
API descriptions

Edit class

The Edit class is derived from the Field class; the DecimalEdit class derives from Edit.

Edit

Edit::Edit ..................................................................................................................... 35
Edit::operator= ................................................................................................................ 36
Edit::AddProperties ............................................................................................................ 36
Edit::Append ................................................................................................................... 36
Edit::ClearProperties ..................................................................................................... 36
Edit::Delete ................................................................................................................... 37
Edit::FocusAtEndOfData ................................................................................................... 37
Edit::GetBookmark ......................................................................................................... 37
Edit::GetBuffer ................................................................................................................ 38
Edit::GetBufferLength ..................................................................................................... 38
Edit::GetBufferSize ......................................................................................................... 38
Edit::GetCursorOffset ..................................................................................................... 38
Edit::GetFieldType .......................................................................................................... 38
Edit::GetLabel ................................................................................................................. 39
Edit::GetProperties ........................................................................................................... 39
Edit::GetStringLength ..................................................................................................... 39
Edit::Insert ................................................................................................................... 39
Edit::ReplaceSelection ..................................................................................................... 40
Edit::SetBookmark .......................................................................................................... 40
Edit::SetBuffer ............................................................................................................... 41
Edit::SetCursorOffset ...................................................................................................... 41
Edit::SetLabel .................................................................................................................. 42
Edit::SetProperties ......................................................................................................... 42
Edit::SetSelection ......................................................................................................... 44
**Edit::Edit**

Constructors for the Edit class

Form 1: Edit()

Form 2: Edit(
    char const * const pnewLabel,
    char * const pnewBuffer,
    int const newLengthofBuffer,
    int const newCharacterOffset=0,
    int const Justify=LCD_LEFT_JUSTIFIED);

Form 3: Edit(
    char const * const pnewLabel,
    int const InitialSize,
    int const MaximumSize,
    int const Justify=LCD_LEFT_JUSTIFIED,
    bool const TruncateIfFull=false)

Form 4: Edit ( const Edit & src )

**Parameters**

- **pnewLabel**
  The label associated with the field
- **pnewBuffer**
  The pointer to the application defined buffer
- **newLengthofBuffer**
  The length of the buffer
- **newCharacterOffset**
  The cursor offset
- **InitialSize**
  The initial size of UI Engine allocated buffer
- **MaximumSize**
  The maximum size of UI Engine allocated buffer
- **Justify**
  The justification of the edit buffer with respect to the screen
- **TruncateIfFull**
  The desired behaviour when the Edit field reaches the specified maximum size. If this parameter is TRUE, when the user inserts new text before the end of the buffer, the text at the end of the buffer will be removed to make room for the new text. If this parameter is FALSE, when the user tries to insert new text before the end of the buffer, the device will beep and the new text will not be added.

**src**
An initialized Edit object; in this case, the constructor returns a duplicate of the src object

**Remarks**
Two ways to use the Edit class are:
- Using an application-defined buffer
API descriptions

Edit class

- Using a UI Engine defined buffer. The UI Engine allocates memory in this case.

**Edit::operator=**
Sets one Edit object equal to another Edit object

```cpp
Edit& operator = ( const Edit & src )
```

**Parameters**

- `src` An initialized Edit object

**Return value**

An Edit object which is a duplicate of `src`

**Remarks**

Sets the left side parameter to be a duplicate of `src`

**Edit::AddProperties**
Adds the specified properties to the Edit field without clearing the properties that have already been set

```cpp
void AddProperties (unsigned long FlagstoSet)
```

**Parameters**

- `FlagstoSet` The properties to add; see `Edit::SetProperties` for the definitions.

**Remarks**

This call leaves the current properties intact.

**Edit::Append**
Appends text to the end of the buffer

```cpp
bool Append ( char const * const pInsertionString,
              int const Length = -1 );
```

**Parameters**

- `pInsertionString` The pointer of the insertion string to append
- `Length` The length of string. The default is the length of the insertion string.

**Return value**

TRUE if successful or FALSE if it was not successful

**Edit::ClearProperties**
Removes all special properties from the Edit field

```cpp
void ClearProperties (unsigned long FlagstoClear)
```

**Parameters**

- `FlagstoSet` The properties to clear
Remarks See Edit::SetProperties for the definitions. This leaves all other properties intact.

**Edit::Delete**

Removes text from the buffer

void Delete (  
    int const Offset,  
    int const Length)

**Parameters**

Offset  
The offset in the edit buffer

Length  
The length of string in the edit buffer to delete

**Edit::FocusAtEndOfData**

Determines if cursor is currently at end of the text in the buffer

virtual bool FocusAtEndOfData()  

**Return value**  
TRUE if the cursor is at the end of the entered text in the buffer, and FALSE otherwise

**Edit::GetBookmark**

Retrieves the current position of the character on which a bookmark was set

int GetBookmark (int const BookmarkNumber =USER_DEFINED_BOOKMARK_NUMBER ) const

**Parameters**

BookmarkNumber  
A value between 0 and Buffer::NUMBER_OF_BOOKMARKS indicating the index number of the bookmark that is to be used. Always use USER_DEFINED_BOOKMARK_NUMBER, because that is the only bookmark number not used internally by the UI Engine. You should only call this function with that value.

**Return value**  
The current position of the character on which the specified bookmark was set

**Remarks**  
A bookmark may be used to keep track of a character in the buffer. It will remain with the character on which it was set until that character is removed from the buffer.
**Edit::GetBuffer**

Returns a pointer to the buffer that stores the characters typed into the Edit field

```c
char const * GetBuffer() const
```

**Return value**

A pointer to a buffer containing the current contents of the edit buffer. Returns NULL value if the buffer has no characters in it.

**Edit::GetBufferSize**

Retrieves the buffer size

```c
int GetBufferSize() const
```

**Return value**

The number of bytes available to the edit buffer

**Edit::GetCursorPosition**

Sets the position of the cursor in the Edit field

```c
int GetCursorOffset() const
```

**Return value**

The current offset in the edit buffer of the cursor

**Edit::GetFieldType**

Returns the type of the current field

```c
FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL ) ;
```

**Parameters**

- `pDerived` A pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. This parameter is optional. It can be NULL. If the parameter is NULL, the function returns no information about the specific derived type.

**Return value**

Returns the type of the base class of the current field (i.e. the type of the first class derived from the Field object)

**Remarks**

For an Edit object, the function will return EDIT_FIELD, and pDerived will be filled in with NULL_FIELD (since Edit is derived directly from Field).
**Edit::GetLabel**
Returns a pointer to the text in the label currently displayed beside the Edit field
```cpp
char const * const GetLabel()
```
Return value: The label associated with current edit field

**Edit::GetProperties**
Returns the properties that the Edit field is currently using
```cpp
unsigned long GetProperties()
```
Return value: The edit buffer properties
Remarks: See SetProperties for the definitions.

**Edit::GetStringLength**
Returns the length of the string currently stored in the buffer
```cpp
int GetStringLength() const
```
Return value: The length of the current string in the buffer

**Edit::Insert**
Inserts text into the buffer
```cpp
bool Insert (  
    char const * const pInsertionString,  
    int const Position = -1,  
    int const Length = -1)
```
Parameters:
- **pInsertionString**: The pointer of the insertion string
- **Position**: The position of insertion
- **Length**: The length of string. The default is the length of the insertion string.
Return value: TRUE if successful or FALSE if it was not successful
**Edit:: ReplaceSelection**
Replaces the current selection with the specified string

```cpp
void ReplaceSelection ( 
    char const * const InsertionString, 
    int const Length = -1);
```

**Parameters**
- **InsertionString**
  The string with which to replace the current selection
- **Length**
  The length of the replacement string. If Length is –1, the entire string will be used up to a terminating NULL character.

**Remarks**
This function should be used together with SetSelection to replace one large block of text with another. First, set the selection using SetSelection and then replace it using ReplaceSelection.

**Edit:: SetBookmark**
Sets a bookmark on a specific character in the edit buffer

```cpp
int SetBookmark ( 
    int const Offset, 
    int const BookmarkNumber = USER_DEFINED_BOOKMARK_NUMBER)
```

**Parameters**
- **Offset**
  The offset of the character on which to set the bookmark
- **BookmarkNumber**
  A value between 0 and Buffer::NUMBER_OF_BOOKMARKS indicating the index number of the bookmark that is to be used. Currently, the only bookmark number not used internally by the UI Engine is USER_DEFINED_BOOKMARK_NUMBER. You should only call this function with that value.

**Remarks**
A bookmark may be used to keep track of a character in the buffer. It will remain with the character on which it was set until that character is removed from the buffer.
**Edit::SetBuffer**

Sets the buffer that will be used to store characters typed into the Edit field.

**Form 1:**
```c
void SetBuffer(
    char * const pnewBuffer,
    int const newLengthofBuffer)
```

**Form 2:**
```c
void SetBuffer(
    char const * const pnewBuffer,
    int const newLengthofBuffer)
```

**Form 3:**
```c
bool SetBuffer(
    int const InitialSize,
    int const MaximumSize,
    bool const TruncateIfFull=false)
```

**Parameters**
- `pnewBuffer`: The pointer to application buffer used in the edit field.
- `newLengthofBuffer`: The length of above buffer.
- `InitialSize`: The initial size of UI Engine allocated buffer.
- `MaximumSize`: The maximum size of UI Engine allocated buffer.
- `TruncateIfFull`: The desired behaviour when the Edit field reaches the specified maximum size. If this parameter is TRUE, when the user inserts new text before the end of the buffer, the text at the end of the buffer will be removed to make room for the new text. If this parameter is FALSE, when the user tries to insert new text before the end of the buffer, the device will beep and the new text will not be added.

**Return value**
Returns TRUE if memory could be allocated for the buffer and FALSE if not.

**Remarks**
Use the first `SetBuffer` when the buffer is allocated by the application and is read/write. Use the second when the buffer is allocated by the application and is read-only. Use the third when the buffer is allocated by the UI Engine and is read/write.

**Edit::SetCursorOffset**

Sets the position of the cursor in the Edit field.

```c
void SetCursorOffset (int newCharacterOffset=0)
```

**Parameters**
- `newCharacterOffset`: The offset in the edit buffer where the character cursor is set.

This is a draft document. Its contents are proprietary and confidential, and copyright © 2001 Research In Motion Limited.
**Edit class**

**Remarks**
The cursor will advance one character as keys are typed; the cursor will go back a character each time the BACKSPACE key is pressed.

**Edit::SetLabel**
Sets the text in the label that will be displayed beside the Edit field

```cpp
void SetLabel(char const * const pnewLabel)
```

**Parameters**
- `pnewLabel` The pointer to the label

**Edit::SetProperties**
Sets the properties of the Edit field, including what types of input the field will accept

```cpp
void SetProperties (  
    unsigned long FlagstoSet,  
    unsigned long FlagstoClear = 0xFF)
```

**Parameters**
- `FlagstoSet` The field and character properties to set. The upper 24 bits is the field property definitions. The lower 8 bits define the character properties.
- `FlagstoClear` The field and character properties to clear. By default all character properties are cleared.

**Remarks**
When first instantiated, there are no field properties and all characters are accepted.

Edit objects have two kinds of properties, field properties and character properties, as listed below.

<table>
<thead>
<tr>
<th>Field Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO_SPACE_SUBSTITUTION</td>
<td>Internal to the UI Engine.</td>
</tr>
<tr>
<td>CR_TO_ROLL_DOWN</td>
<td>Changes the meaning of ENTER from “insert a new line” to “roll down.” If the cursor is at the end of a field and the ENTER key is pressed, the cursor will go to the next field.</td>
</tr>
<tr>
<td>DISABLE_AUTO_FORMATTING</td>
<td>Bitwise OR of DISABLE_POSITION_AUTO_CAP and DISABLE_POSITION_AUTO_PUNC.</td>
</tr>
</tbody>
</table>
## API descriptions

### Edit class

<table>
<thead>
<tr>
<th>Field Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE_POSITION_AUTO_CAP</td>
<td>Disables the property of capitalizing the word after a punctuation mark followed by a space.</td>
</tr>
<tr>
<td>DISABLE_POSITION_AUTO_PUNC</td>
<td>Disables the property of substituting 2 spaces with period then space.</td>
</tr>
<tr>
<td>DISABLE_PRESS_HOLD_AUTO_CAP</td>
<td>Disables the property of pressing and holding an alpha key down to automatically capitalize it.</td>
</tr>
<tr>
<td>EMAIL_FIELD</td>
<td>Bitwise OR of AUTO_SPACE_SUBSTITUTION, CType::EMAIL_ADDRESS, and DISABLE_AUTO_FORMATTING.</td>
</tr>
<tr>
<td>HANGING_INDENT</td>
<td>Indents the edit buffer.</td>
</tr>
<tr>
<td>PASSWORD_FIELD</td>
<td>Substitutes every character entered with an asterisk.</td>
</tr>
<tr>
<td>READ_ONLY</td>
<td>Edit buffer is read only. If the field has the focus and is given keyboard input (using the UI Engine member function HandleInput), the UI Engine returns UNHANDLED.</td>
</tr>
</tbody>
</table>

### Character property

<table>
<thead>
<tr>
<th>Character property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHABETIC</td>
<td>A-Z, a-z</td>
</tr>
<tr>
<td>ALPHANUMERIC</td>
<td>ALPHABETIC + DECIMAL_NUMERIC</td>
</tr>
<tr>
<td>DECIMAL_NUMERIC</td>
<td>0-9</td>
</tr>
<tr>
<td>EMAIL_ADDRESS</td>
<td>All characters except specials, CRs, and SPACE (except '@' and '.')</td>
</tr>
<tr>
<td>EMAIL_WORD</td>
<td>All characters except specials, CRs, and SPACE.</td>
</tr>
<tr>
<td>HEXDECIMAL_NUMERIC</td>
<td>0-9,a-f</td>
</tr>
<tr>
<td>PHONE</td>
<td>0-9, (,), +, -, x, X</td>
</tr>
<tr>
<td>URL</td>
<td>URL characters (a-z,0-9,@,,#,?,=)</td>
</tr>
</tbody>
</table>
**Edit::SetSelection**

Sets a block of text that can be replaced with another block of text

```c
void SetSelection (  
    int Offset,  
    int Length );
```

**Parameters**

- **Offset**
  - The character position at which to start the selection

- **Length**
  - The length of the selection; a **Length** of 0 clears the selection.

**Remarks**

This function can be used together with `ReplaceSelection` to replace one large block of text with another. Set the selection using `SetSelection`, and then replace it using `ReplaceSelection`. The selection is not visible in the UI.
Field class

The Field class is a base class for most of the components of the UI architecture.

Field

Field::GetFieldType ............................................................................................................ 45
Field::IsDirty .................................................................................................................... 46
Field::IsFocusVisible ........................................................................................................ 46
Field::MarkAsClean ......................................................................................................... 46
Field::MarkAsDirty ........................................................................................................... 46
Field::Redraw .................................................................................................................. 46
Field::SetJustification ..................................................................................................... 46

Field::GetFieldType

Returns the type of field; pure virtual and must be overridden

virtual FIELDTYPE GetFieldType (FIELDTYPE * pDerived = NULL) = 0

Parameters

pDerived

An optional pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. Optional. If the parameter is NULL, the function returns no information about the specific derived type.

Return value

The type of the base class of the current field (that is, the type of the first class derived from the Field object), as a FIELDTYPE constant. One of:

- NULL_FIELD
- EDIT_FIELD = 1
- DECIMAL_EDIT_FIELD
- LIST_FIELD
- STRING_LIST_FIELD
- SEPARATOR_FIELD
- CHOICE_FIELD
- BEEP_CHOICE_FIELD
- YES_NO_CHOICE_FIELD
- TOP_BOTTOM_CHOICE_FIELD
- TABLE_FIELD
- DATE_FIELD
- TIME_FIELD
- RICHTEXT_FIELD
Field::IsDirty
Determines whether or not the component has changed since the last redraw
virtual bool IsDirty() const

Return value
TRUE if the component has changed since the last redraw and therefore requires a redraw to display new information, and FALSE otherwise.

Field::IsFocusVisible
Determines if the cursor is visible; can be overridden
virtual bool IsFocusVisible()

Return value
TRUE if the cursor is visible, FALSE otherwise.

Field::MarkAsClean
Marks the field clean
virtual void MarkAsClean()

Remarks
IsDirty() will return FALSE after this call has been made

Field::MarkAsDirty
Marks the field dirty
virtual void MarkAsDirty()

Remarks
IsDirty() will return TRUE after this call has been made

Field::Redraw
Draws the component
virtual void Redraw()

Remarks
The method is responsible for displaying the component. It will call the Draw and Paint methods to actually render the component.

Field::SetJustification
Sets the justification mode for the text in the field.
Form 1: void SetJustification(int const newJustify=LCD_LEFT_JUSTIFIED)
Form 2: void SetJustification(JUSTIFICATION const justification)

Parameters
newJustify
One of:
Field class

- LCD_LEFT_JUSTIFIED
- LCD_RIGHT_JUSTIFIED
- LCD_CENTERED

Justification
One of:
- JUST_CENTER
- JUST_LEFT
- JUST_RIGHT
- JUST_FULL

Remarks: Form 1 is deprecated. Developers should use form 2, which was introduced in version 2.0 of the SDK.
FieldManager class

The FieldManager class, introduced in version 2.0 of the SDK, is an abstract base class. It defines methods and functionality associated with managing fields. Classes that inherit from field manager include Screen and Dialog.

FieldManager

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldManager::AddField</td>
<td>48</td>
</tr>
<tr>
<td>FieldManager::BottomOfDisplay</td>
<td>49</td>
</tr>
<tr>
<td>FieldManager::FocusEvents</td>
<td>49</td>
</tr>
<tr>
<td>FieldManager::GetFieldWithFocus (Deprecated)</td>
<td>49</td>
</tr>
<tr>
<td>FieldManager::GetFirstField</td>
<td>49</td>
</tr>
<tr>
<td>FieldManager::GetHeight</td>
<td>50</td>
</tr>
<tr>
<td>FieldManager::GetLastField</td>
<td>50</td>
</tr>
<tr>
<td>FieldManager::GetWidth</td>
<td>50</td>
</tr>
<tr>
<td>FieldManager::Invalidate</td>
<td>50</td>
</tr>
<tr>
<td>FieldManager::OnFieldUpdate</td>
<td>50</td>
</tr>
<tr>
<td>FieldManager::PutFieldAtBottomOfDisplay</td>
<td>51</td>
</tr>
<tr>
<td>FieldManager::PutFieldATopOfDisplay</td>
<td>51</td>
</tr>
<tr>
<td>FieldManager::RemoveAllFields</td>
<td>51</td>
</tr>
<tr>
<td>FieldManager::RemoveField</td>
<td>51</td>
</tr>
<tr>
<td>FieldManager::SetDimensionDependentData</td>
<td>51</td>
</tr>
<tr>
<td>FieldManager::SetFieldwithFocus</td>
<td>52</td>
</tr>
<tr>
<td>FieldManager::TopOfDisplay</td>
<td>52</td>
</tr>
</tbody>
</table>

FieldManager::AddField

Adds a field to the field manager

virtual void AddField(
    Field &newField,
    Field * pFieldAfterNewField = NULL)

Parameters
d newField
   A field object (list, choice box, edit box, table)

d pFieldAfterNewField
   Pointer to the existing field you want to follow your new field, or NULL (the default value for this parameter)

Remarks
If you specify a field pointer for the second pointer, this function inserts your new field just before this existing field; otherwise, if you specify NULL for the second parameter, the function adds your new field to the end of the associated field manager’s list of fields.
**FieldManager::BottomOfDisplay**
Sets the focus to the field at the bottom of the field manager

```cpp
void BottomOfDisplay()
```

**FieldManager::FocusEvents**
Monitors focus changes for a given field

```cpp
virtual void OnFocusEvent(
    FOCUS_EVENT e,
    Field * f)
```

**Parameters**
- `e` One of FOCUS_GAINED or FOCUS_LOST.
- `f` The field to be monitored.

**Remarks**
The `OnFocusEvent` function is called when a focus event takes place. It is intended for application writers to monitor focus changes as a way to trigger input verification. Currently the function is called after the Field has been notified of the change.

Do not rely on the return value of `GetFieldWithFocus()` during the processing of this call.

**FieldManager::GetFieldWithFocus**
Returns a pointer to the field that currently has focus

```cpp
Field * GetFieldWithFocus()
```

**Return value**
The pointer to the field on the screen that currently has the focus

**FieldManager::GetFieldwithFocus** *( Deprecated )*  
**Deprecated** – use `GetFieldWithFocus` instead

```cpp
Field * GetFieldwithFocus()
```

**Return value**
The pointer to the field on the screen that currently has the focus

**Remarks**
This version is included only for backward compatibility

**FieldManager::GetFirstField**
Returns a pointer to the field at the top of the field manager

```cpp
Field * GetFirstField()
```

**Return value**
The pointer to the field that is the first in the list of field in the screen
FieldManager::GetHeight
Retrieves the height of the field manager in pixels
int GetHeight() const

Return value
The height of the field manager in pixels

See also
FieldManager::GetWidth

FieldManager::GetLastField
Get the last field in the field manager’s field list
Field * GetLastField()

Return value
A pointer to the last field in the field manager

FieldManager::GetWidth
Retrieves the width of the field manager in pixels
int GetWidth() const

Return value
The width of the field manager in pixels

See also
FieldManager::GetHeight

FieldManager::Invalidate
Invalidates the entire field manager
virtual void Invalidate()

Remarks
Invalidate will cause a repaint on the screen component if Process() is used as the message loop.

FieldManager::OnFieldUpdate
Notification that a field’s data has changed; can override this method
virtual void OnFieldUpdate(Field & field)

Parameters
field
A reference to the field whose data has changed
FieldManager::PutFieldAtBottomOfDisplay
Instructs the UI Engine to display the specified field at the bottom of the display

void PutFieldAtBottomOfDisplay( Field * newpBottomField )

Parameters
newpBottomField
The field that should be positioned at the bottom

FieldManager::PutFieldAtTopOfDisplay
Instructs the UI Engine to display the specified field at the top of the display

void PutFieldAtTopOfDisplay ( Field * newpTopField )

Parameters
newpTopField
The field that should be positioned at the top

FieldManager::RemoveAllFields
Removes all the fields from the field manager

void RemoveAllFields()

FieldManager::RemoveField
Removes a field from the field manager

virtual void RemoveField (Field &FieldToRemove)

Parameters
FieldToRemove
A field object (menu, list, choice box, edit box, table)

Remarks
This function removes the field object from the list of fields in the associated Field Manager.

FieldManager::ResetDimensionDependentData
Walks the list of fields associate with the manager and calls the recalculation methods of all child components

virtual void ResetDimensionDependentData()

Remarks
Call this method when system display parameters have changed. For instance, when system fonts have been changed from small to large.
FieldManager class

FieldManager::SetFieldwithFocus
Instructs the UI Engine to give the focus to the specified field

```cpp
bool SetFieldwithFocus( Field * pField)
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pField</td>
<td>A pointer to a field object (menu, list, choice box, edit box, table)</td>
</tr>
</tbody>
</table>

Return value

TRUE if successful or FALSE if the field does not exist (i.e. you did not invoke AddField member function)

Remarks

If the field has more than one item that can be set or selected, the previous item that was set before that field lost the focus will be re-selected by default.

**Note**

This function will fail if the pointer isn’t in the list of fields; for this reason, the call is deprecated. You should use SetFieldWithFocus instead.

FieldManager::TopOfDisplay
Sets the focus to the field at the top of the field manager

```cpp
void TopOfDisplay ()
```

See also

FieldManager::BottomOfDisplay
Label class

A Label object is a box that contains static text. The Label class is derived from the Field class, and the Title class derives from Label.

Label

Label::Label ................................................................................................................... 53
Label::SetText ................................................................................................................. 53

Label::Label
Constructors for the Label class

Form 1: Label()

Remarks When using form 1, you must call SetText before the Label is displayed.

See also Label::SetText

Label::SetText
Set the text to be displayed

Form 1: void SetText(char const * const pnewText)

Form 2: void SetText( char const * const text,
int length)

Parameters pnewText

The pointer to the text that appears in the Label

length

The length of the text

Remarks If form 2 is used then all ‘\0’ characters in the string are ignored
List class

The functions on the following pages are listed in alphabetical order.

List

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List::List</td>
<td>54</td>
</tr>
<tr>
<td>List::Delete</td>
<td>55</td>
</tr>
<tr>
<td>List::GetColumnFont</td>
<td>55</td>
</tr>
<tr>
<td>List::GetFieldType</td>
<td>55</td>
</tr>
<tr>
<td>List::GetNumEntries</td>
<td>55</td>
</tr>
<tr>
<td>List::GetSelectedIndex</td>
<td>56</td>
</tr>
<tr>
<td>List::GetTopLineEntry</td>
<td>56</td>
</tr>
<tr>
<td>List::Insert</td>
<td>56</td>
</tr>
<tr>
<td>List::NowDisplaying</td>
<td>56</td>
</tr>
<tr>
<td>List::PutColumn</td>
<td>57</td>
</tr>
<tr>
<td>List::Redraw</td>
<td>57</td>
</tr>
<tr>
<td>List::SetColumnFont</td>
<td>58</td>
</tr>
<tr>
<td>List::SetEnableMultipleSelection</td>
<td>58</td>
</tr>
<tr>
<td>List::SetNumEntries</td>
<td>58</td>
</tr>
<tr>
<td>List::SetSelectedIndex</td>
<td>58</td>
</tr>
<tr>
<td>List::SetTopLineEntry</td>
<td>59</td>
</tr>
</tbody>
</table>

List::List

Constructor for the List class

Form 1: List()
Form 2: List (int const NumEntries)

Parameters

- NumEntries
  - The number of entries in the list

Remarks

When the UI Engine needs to display the string associated with index in the number of entries, the UI Engine calls the pure virtual member function NowDisplaying once for every line.

Note

It's your job to write the NowDisplaying member function.

NowDisplaying would then call the PutColumn member function, perhaps a number of times, which tells the UI Engine the associated string(s) to display. (This is useful if the displayed list needs its information columnized.)
List::Delete
Removes an entry from the list

void Delete(
  int const Index,
  bool Redraw = true )

Parameters
  Index
    The index of the item deleted
  Redraw
    An indicator to the UI Engine that the list, if present on the screen, should be redrawn

List::GetColumnFont
Retrieves the font associated with a particular column

Font const * GetColumnFont(int column)

Parameters
  column
    The index of the column for which to retrieve the font

Return value
  Returns the font object associated with the column

List::GetFieldType
Returns the type of the current field

FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL );

Parameters
  pDerived
    A pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. This parameter is optional. It can be NULL. If the parameter is NULL, the function returns no information about the specific derived type.

Return value
  Returns the type of the base class of the current field (i.e. the type of the first class derived from the Field object)

Remarks
  For a List object, the function will return LIST_FIELD, and pDerived will be filled in with NULL_FIELD (since List is derived directly from Field).

List::GetNumEntries
Returns the number of entries currently in the list

int GetNumEntries()}

Return value
  The number of entries in the list
**List::GetSelectedIndex**  
Returns the index of the currently selected list entry  

```cpp
int GetSelectedIndex()
```

**Return value**  
The index of the selected item in the list  

**List::GetTopLineEntry**  
Returns the specified list entry currently displayed at the top of the display  

```cpp
int GetTopLineEntry()
```

**Return value**  
The index of the first entry being displayed on the display  

**List::Insert**  
Inserts an entry into the list  

```cpp
void Insert(
    int const Index,
    bool Redraw = true)
```

**Parameters**  
- **Index**  
The index where the item is inserted  
- **Redraw**  
An indicator to the UI Engine that the list, if present on the screen, should be redrawn  

**Remarks**  
The UI Engine will call `NowDisplaying` for the inserted item.  

**List::NowDisplaying**  
A virtual function called by the UI Engine when a particular entry in the list needs to be drawn  

```cpp
void NowDisplaying(int const Index)
```

**Parameters**  
- **Index**  
The index in the string array of item being updated  

**Remarks**  
This is a pure virtual function (written by you, called by the UI Engine). The UI Engine calls this function when the application invokes one of the drawing functions. This triggers the application to call the `PutColumn` member function to update the string associated with this handle. The `PutColumn` member function can be called more than once, normally if the application wants columnized strings on the associated line.
**List::PutColumn**

Called from within `NowDisplaying` to display the text associated with the list entry currently being drawn.

Form 1: `void PutColumn(
    char const * const pnewStr,
    int const ColumnWidth = -1,
    int const Flags = DEFAULT,
    int const StrLength = INT_MAX,
    Font const * font = NULL)`

Form 2: `void PutColumn(
    BitMap const * pBitmap,
    int const ColumnWidth = 0,
    int const Flags = DEFAULT)`

**Parameters**

- `pnewStr`  
  The pointer to the string associated with this index

- `ColumnWidth`  
  The maximum width of the location (in pixels)

- `Flag`  
  Reserved for future enhancement

- `StrLength`  
  Reserved for future enhancement

- `pBitmap`  
  A pointer to a BitMap resource to display

- `font`  
  Pointer to the font to be used for display; if NULL or omitted, the currently-selected font is used

**Remarks**

This function can be called more than once. Subsequent calls after the first call are based on the previous `ColumnWidth`. The text is left justified in `ColumnWidth`.

**List::Redraw**

Redraws the visible portion of the list by calling `NowDisplaying` for each visible list entry.

Form 1: `void Redraw()`  

Form 2: `void Redraw(int const Index)`

**Parameters**

- `Index`  
  The index in the string array of item being updated
List class

Remarks
Form 1 is an override of the Field::Redraw() method. This function causes the UI Engine to redraw the line associated with the item in the string array (perhaps the string associated with the line has changed).

Note
The UI Engine will call NowDisplaying for the item.

List::SetColumnFont
Sets a font for a particular column

void SetColumnFont(
    int column,
    Font const * font)

Parameters
column
The index of the column for which to set the font
font
The font object to set for the column

List::SetEnableMultipleSelection
Enables multiple selection mode for the List

void EnableMultipleSelection()

List::SetNumEntries
Sets the number of entries that should be displayed in the list

void SetNumEntries (int const NumEntries)

Parameters
NumEntries
Sets the number of entries in the list

List::SetSelectedIndex
Instructs the UI Engine to show the specified list entry as highlighted

void SetSelectedIndex( int const Index)

Parameters
Index
The index of the item in list the application wants selected
List::SetTopLineEntry

Instructs the UI Engine to display the specified list entry at the top of the display

void SetTopLineEntry ( int const Index )

Parameters

Index

The index of the item to place at the top of the display
Menu class

The Menu class is a base class for handling menus.

**Menu**

Menu::Menu ..................................................................................................................... 60
Menu::GetSelectedIndex ........................................................................................................ 61
Menu::GetTopIndex .............................................................................................................. 61
Menu::HideItem ................................................................................................................. 61
Menu::HideItems ................................................................................................................ 62
Menu::SetMenuItems ........................................................................................................... 62
Menu::SetSelectedIndex ..................................................................................................... 63
Menu::SetTopIndex .............................................................................................................. 63
Menu::ShowItem ................................................................................................................. 64

**Menu::Menu**

Constructors for the Menu class

Form 1: Menu()

Form 2: Menu (char const *const *const newMenuStringArray, int const NumberOfEntries)

Parameters

newMenuStringArray

A pointer to the address of a string array, containing user-defined menu items

NumberOfEntries

The number of application defined menu items in the newMenuStringArray (maximum 32)

Remarks

This is the constructor. An item string that begins with two underscore marks (“__”) is a menu separator.

**Note**
The user can never select menu separators and GetSelectedIndex (below) never returns its offset in the newMenuStringArray.

Menus are composed of 4 kinds of elements:

**HIDE_MENU**

This always appears at the beginning. The UI Engine will return a status code of hide_menu from HandleInput().

**Application defined menu items**

These are the menu items passed in from the application. The preferred form for a menu item is Action-Object, such as “Open message” or “Destroy record”.

**Note**
These can be hidden or shown individually by the associated member function.
System menu items
There is one system menu item defined, “Hide menu.” This function returns an integer that the application can use as input to `SetSelectedIndex`.

Task Items
These items are added by the UI Engine. They are the application’s tasks that are currently running. The displayed name is the name in the `VersionPtr` variable in the application’s `PagerMain()` function.

**Menu::GetSelectedIndex**
Returns the index of the currently selected menu item

```cpp
int GetSelectedIndex() const
```

**Return value**
The offset of the user-selected menu item

**Remarks**
The value returned is an integer, not a bitmap.

**Note**
Menu separator item strings that begin with two underscore characters (“_”) can never be returned.

**See also**
`Menu::SetSelectedIndex`

**Menu::GetTopIndex**
Gets the offset to the item to be displayed at the top of the menu

```cpp
int GetTopIndex() const
```

**Return value**
The index or offset into the menu list to be displayed as the top item in the menu

**Menu::HideItem**
Instructs the UI Engine not to display the specified menu item the next time the menu is displayed

```cpp
void HideItem(int const Item)
```

**Parameters**

- **Item**
The offset in `newMenuStringArray` of the menu item to be hidden. Bit 0 represents item 0, bit 1 represents item 1, and so forth.
### Menu::HideItems

Instructs the UI Engine not to display the specified menu items the next time the menu is displayed.

**Form 1**: `void HideItems (int const BitMaptoHide)`

**Parameters**
- `BitMaptoHide`: A bitmap relationship with the associated `newMenuStringArray`. A value of 1 hides the corresponding menu item; a value of 0 displays it. In form 1, this bitmap is expressed as an `int`.

**Remarks**
This function allows you **not** to display certain menu items that are passed in the `newMenuStringArray`. One purpose for this would be to allow the user to use the same string array for more than one menu, where one menu would display certain items and another menu would not. In order to display the effects of this function, the application must perform a `ProcessMenu`.

Form 1 is limited to addressing the first 32 items in the associated `newMenuStringArray`.

### Menu::SetMenuItems

Sets the strings that will be shown on each line of the menu.

```cpp
void SetMenuItems (
    char const * const * const newMenuStringArray,
    int const NumberofEntries = -1)
```

**Parameters**
- `newMenuStringArray`: The pointer to the address of a string array, containing menu items
- `NumberofEntries`: The number of menu items in the `newMenuStringArray`. The maximum number is 32.

**Remarks**
An item string that begins with two underscore marks ("__") is a menu separator.

**Note**
The user can never select menu separators and `GetSelectedIndex` (below) never returns a separator’s offset in the `newMenuStringArray`.
Menus are composed of 4 kinds of elements:

**HIDE_Menu**
This always appears at the beginning. The UI Engine will return a status code of **HIDE_Menu** from `HandleInput()`.

**Application defined menu items**
These are the menu items passed in from the application.

**System menu items**
There is one system menu item defined, “Hide menu.” This function returns an integer that the application can use as input to `SetSelectedIndex`.

**Task Items**
These items are added by the UI Engine. They are the applications’ tasks that are currently running. The displayed name is the name in the VersionPtr variable in the applications’ `PagerMain()` function.

**Menu::SetSelectedIndex**
Instructs the UI Engine to show the specified menu item as highlighted

```cpp
void SetSelectIndex( int const Item )
```

**Parameters**
Item

The offset in the newMenuStringArray

**See also**
`Menu::GetSelectedIndex`

**Menu::SetTopIndex**
Sets the top menu item

```cpp
void SetTopIndex( int const Item )
```

**Parameters**
Item

The index of the first menu item to display

**Remarks**
This function sets the item offset to display at the top of the screen. When the menu is rendered for the screen, the item specified is drawn at the top of the list. To access previous items, the user must scroll up.
Menu::ShowItem

Instructs the UI Engine to display the specified menu item the next time the menu is displayed

```c
void ShowItem(int const Item)
```

**Parameters**

- **Item**
  The offset in `newMenuStringArray` of the item requested to be made visible

**Remarks**

This function is the opposite of the `HideItem` function.
OKDialog class

OKDialog is a class derived from Dialog. An OKDialog box can be used to present important information to the user. The information will be displayed in a dialog box with an Information bitmap and field containing the string ‘OK’. The box will be dismissed when the user clicks the track-wheel or presses the ENTER key on the ‘OK’ field.

**OKDialog**

- **OKDialog::Go**
- **OKDialog::OKDialog**
- **OKDialog::SetQuestion**

**OKDialog::OKDialog**

Constructor for the OKDialog class

Form 1: OKDialog (char const * const pStatement = NULL)
Form 2: OKDialog ( const OKDialog & src )

**Parameters**
- **pStatement**
  - The statement to be displayed in the dialog box
- **src**
  - An initialized OKDialog object

**Remarks**

- The statement will be displayed in a dialog box with an Information bitmap and a field containing the string ‘OK’. The box will be dismissed when the user clicks the track-wheel or presses the ENTER key on the OK field.

Form 2 creates an OKDialog object that is a duplicate of the src parameter.

**OKDialog::operator =**

Sets one OKDialog object equal to another OKDialog object

OKDialog& operator = ( const OKDialog& src )

**Parameters**
- **src**
  - An initialized OKDialog object

**Return value**
- An OKDialog object that is a duplicate of src.
**OKDialog::Go**
Displays the dialog and clears it when the user clicks the trackwheel or presses the ENTER key

RESULT Go (UIEngine & UIEngineToUse)

**Parameters**
UIEngineToUse
The UIEngine object that should be used to display the dialog box

**Return value**
Returns CLICKED when the user clicks the trackwheel or presses the ENTER key

**OKDialog::SetQuestion**
Sets the text that should be displayed in the dialog box

bool SetQuestion (char const *const pStatement)

**Parameters**
pStatement
The statement to be displayed in the dialog box

**Return value**
TRUE if the statement was set successfully and FALSE otherwise

**Remarks**
The statement will be displayed in a dialog box with an Information bitmap and a field containing the string ‘OK’.
RichText class

The RichText component, introduced in version 2.0, is a child of the Field class. It defines advanced text layout operations, such as underlined and bold text, as well as different font sizes within a stream of text. RichText is a read-only buffer, initialized with a block of text that cannot be modified.

RichText

RichText::RichText

Constructor

RichText()

RichText::GetAttributes

Retrieves attributes of the RichText field

void GetAttributes(
    TextRangeAttribute const * &AttributeArray,
    int & NumberOfAttributes
)

Parameters

AttributeArray
    Storage for an array of attributes associated with the RichText field.

NumberOfAttributes
    The number of attributes in the array

RichText::GetBuffer

Retrieves the buffer associated with the RichText component

char const * GetBuffer() const

Return value

The rich text string as pure text
### RichText::GetBufferLength
Retrieves the buffer length associated with the RichText component buffer

```cpp
int GetBufferLength() const
```

**Return value** The length of the RichText buffer

---

### RichText::GetCursorOffset
Retrieves the current cursor offset

```cpp
int GetCursorOffset() const
```

**Return value** The offset of the cursor inside the RichText buffer

---

### RichText::SetAttributes
Sets attributes for a rich text component

```cpp
void SetAttributes(
    TextRangeAttribute const * AttributeArray,
    int NumberOfAttributes)
```

**Parameters**
- `AttributeArray` An array of attributes to set for the RichText buffer
- `NumberOfAttributes` The number of attributes in the array

---

### RichText::SetBuffer
Sets the buffer for the rich text component

```cpp
void SetBuffer(
    char const * pNewBuffer,
    int LengthOfBuffer)
```

**Parameters**
- `pNewBuffer` The new buffer to set for the RichText component
- `LengthOfBuffer` The length of the buffer

---

### RichText::SetCursorOffset
Sets the cursor offset into the rich text buffer

```cpp
void SetCursorOffset(int NewCharacterOffset)
```

**Parameters**
- `NewCharacterOffset` The offset into the buffer
RichText::SetLabel
Sets the label for the RichText field

```c
void SetLabel(char const * label)
```

Parameters

- `label`: The label for the RichText field

### Structures

**RichText::TextRangeAttribute**
A structure containing text range attribute information; declared in RichText.h

```c
struct TextRangeAttribute
{
    int Offset;
    int Length;
    unsigned char Attribute;
}
```

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>The offset into the text buffer indicating the start of the text range</td>
</tr>
<tr>
<td>Length</td>
<td>The length of the affected text</td>
</tr>
<tr>
<td>Attribute</td>
<td>A font index indicating the font used for the text range; a number in the range of 0..4</td>
</tr>
</tbody>
</table>
Screen class

The screen class inherits from the FieldManager class.

Screen::Screen
Constructor for the Screen class

Form 1: Screen()

Form 2: Screen(UIEngine & uiEngine)

Parameters

uiEngine
A reference to a UIEngine component to use for rendering the screen.

Remarks
This constructor instantiates a screen object. A screen object is the container of the field objects and a title object.

Screen::AddLabel
Adds a title to the top of the screen

void AddLabel(Label &newTitle)

Parameters

newTitle
A title object

Remarks
See “Title class” on page 100 for details on title objects.
**Screen::ClearScreen**  
Removes all fields from the screen component  
void ClearScreen()

**Screen::Close**  
Called to exit the Process() message loop  
void Close()

**Screen::DidSave**  
Determines whether a user has requested a save  
bool DidSave()  
Return value  
Returns TRUE if the user has requested a save for the screen component  
Remarks  
If Save() is not overridden, then this method can be used to determine if users have requested a save on the screen component.

**Screen::GetUIEngine**  
Gets the UIEngine associated with the screen  
UIEngine & GetUIEngine()  
Return value  
Returns a reference to the UIEngine component associated with the screen

**Screen::IsDisabled**  
Determines whether the screen is disabled  
bool IsDisabled()  
Return value  
Returns TRUE if the screen is disabled and FALSE otherwise.

**Screen::IsFieldDirty**  
Returns TRUE if any field on the screen is marked as dirty  
bool IsFieldDirty()  
Return value  
TRUE if any of the fields associated with the screen have changed  
Remarks  
The application must invoke the member function MarkFieldAsClean in order for the UI Engine to clear this state.
Screen::MarkFieldAsClean
Marks every field on the screen as clean
void MarkFieldAsClean()

Remarks
A clean field is one that has had no changes made to it. This function marks all the fields associated with the screen as clean.

Screen::MarkFieldAsDirty
Marks every field on the screen as dirty
void MarkFieldAsDirty()

Remarks
A dirty field is one that has had changes made to it. This function marks all the fields associated with the screen as dirty.

Screen::OnChangeOptions
Used to implement the “Change Options” for choice fields; can override this method
virtual bool OnChangeOptions()

Return value
Returns TRUE if the Change Option was successful and returns FALSE otherwise.

Remarks
This method is called when the field in focus is a choice field and the user selects the “change options” menu item.

Screen::OnClose
Called when the user exits the screen; can override this method
virtual void OnClose()

Remarks
The default implementation of this method will exit the screen if IsFieldDirty() returns FALSE. Otherwise, a default dialog is displayed, indicating that screen component data has changed. Override this method to display a custom dialog.
Screen::OnKey
Handles key input for the screen; can override this method

virtual void OnKey(
    int event,
    char character,
    int flags)

Parameters

  event
    One of KEY_DOWN, KEY_UP or KEY_REPEAT

  character
    The character pressed

  flags
    Modification flags, such as ALT, CAPS etc.

Remarks

  Override this method to provide input handling at the screen level. This
  method is only called if the field in focus does not accept the input.

Screen::OnMenuItem
Called when a menu item has been selected; can override this method

virtual void OnMenuItem(int item)

Parameters

  item
    The menu item selected

Remarks

  Note that OnMenuItem() returns the same value as (GetSelectedIndex() - 1).
  The OS ignores the mandatory leading menu separator (see Screen::SetMenu),
  which accounts for the off-by-one value.

Screen::OnMessage
Called when a message has been received; can override this method

virtual void OnMessage(const MESSAGE & message)

Parameters

  message
    The RIM message structure populated with newly received message
    information

Remarks

  This method is called to handle RIM messages received while the screen
  is active. The default implementation routes the message to either the
  screen’s or menu’s message handler, depending on which is in focus.
**Screen::OnSave**
Checks if the screen has changed and calls `Save();` can override this method

```
virtual void OnSave()
```

**Remarks**
This method checks if anything in the screen has been modified since the last save, and if so, prompts the user with a dialog saying the contents have changed. The `Save()` method is subsequently called.

**Screen::OnRoll**
Called when the trackwheel has moved; can override this method

```
virtual void OnRoll(
    int directionMagnitude,
    int flags)
```

**Parameters**
- `directionMagnitude`
  A value indicating the magnitude of the roll. Positive values indicate a roll down. A value of 1 should cause the screen to display one new field or one new line of text.
- `flags`
  Key modifiers such as `ALT_STATUS` or `SHIFT_STATUS`, etc.

**Remarks**
Normally this method is called by `OnMessage()` or `Process()`, so an application does not need to call it explicitly. You should only need to override this function if your application bypasses the normal message loop.

**Screen::Process**
Activates the screen and provides a message loop

```
int Process()
```

**Return value**
If the menu is activated during the process call, the return value is the item selected, otherwise the return is −1.

**Screen::RemoveLabel**
Removes a title from the screen

```
void RemoveLabel (Label &TitleToRemove)
```

**Parameters**
- `TitleToRemove`
  A title object

**Remarks**
This function removes the title object from the list of titles in the associated screen.
See also  Screen::AddLabel

**Screen::ResetScreen**
Removes all titles and fields from the screen

```cpp
void ResetScreen()
```

**Remarks**
This function effectively does what ClearScreen in the UI Engine does. It also removes all the links between the fields (linked when AddField is invoked) and all the titles (linked when AddLabel is invoked), and it physically blanks out the display.

**Screen::Save**
Saves the contents of the screen; can override this method

```cpp
virtual void Save()
```

**Remarks**
You should override this method and save to flash any data that is to persist. This method is called by OnSave() if fields have changed and need to be saved.

This method is inherited and overridden from FieldManager.

**Screen::SetMenu**
Sets the menu for use with the default menu handling

```cpp
void SetMenu(
    char const * const * ppItems,
    int nItems,
    int iMenuDefaultSelection,
    int fMenuMask)
```

**Parameters**

- `ppItems`  
  An array of strings to use as the menu items. The first item in every menu should be a menu separator of two underscores, “__”. Menu items should be in the form `verb object`, such as “Open mail” or “Cancel compose” or “Quit game”.

- `nItems`  
  The number of items in the ppItems array

- `iMenuDefaultSelection`  
  The index of the default menu item. This parameter is passed onto the OnMenu(…) method as the selection parameter

- `fMenuMask`  
  A 32 bit bitmask indicating which items are to be disabled. An ‘on’ bit indicates that the corresponding item at that index should be disabled.
Screen::SetScreenDisabled

Disables the screen

void SetScreenDisabled()

Remarks

Use this method to disable the screen. A disabled screen (and all contained components) will not be redrawn during a repaint operation.
Separator class

A separator field is used to separate 2 fields for readability purposes. If no parameters are passed to it, the string displayed is a line.

Note A separator field can never have the focus. Therefore, if you are scrolling over a separator field, the subsequent field will get the focus.

The functions on the following pages are listed in alphabetical order.

Separator
- Separator::Separator
- Separator::GetFieldType
- Separator::SetText

Separator::Separator
Constructor for the Separator class

Form 1: Separator (char const * const pnewSeparatorString = NULL, int newSeparatorStringLength = -1, int const newJustification = LCD_LEFT_JUSTIFIED, bool FontHeight = true)

Form 2: Separator (BitMap const * const pnewSeparatorBitMap, int const newXOffset = 0)

Parameters
- pnewSeparatorString: The pointer to the label associated with the separator; if this is NULL, the Separator object will be displayed as a black line across the width of the screen.
- newSeparatorStringLength: The length associated with the separator
- pnewSeparatorBitMap: The pointer to bitmap associated with the separator
- newJustification: The justification associated with the separator
- newXOffset: The horizontal offset of the bitmap
- FontHeight: If set to TRUE, the Separator object will have the same height as a single line of text. If set to FALSE, the Separator object will be shorter.

Remarks Form 3 returns a duplicate of src
Separator::GetFieldType

Returns the type of the current field

\[
\text{FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL );}
\]

- **Parameters**
  - *pDerived*: A pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. This parameter is optional. It can be NULL. If the parameter is NULL, the function returns no information about the specific derived type.

- **Return value**: Returns the type of the base class of the current field (i.e. the type of the first class derived from the Field object)

- **Remarks**: For a Separator object, the function will return SEPARATOR_FIELD, and *pDerived* will be filled in with NULL_FIELD (since Separator is derived directly from Field).

Separator::SetText

Sets the text that should be displayed by the Separator object

\[
\text{void SetText (char const * const pnewText);}\]

- **Parameters**
  - *pnewText*: The text that should be displayed

- **Remarks**: If *pnewText* is NULL, the Separator object will be displayed as a black line across the width of the screen.
**Status class**

The status box displays system modal informational messages for a pre-defined period of time. For any other kinds of messages, use a dialog box.

The following is an example of a status box with the relative position of the `MessageString` and the Bitmap (which are both optional). `DisplayTime` is always in 1/100 second intervals.

![A status box]

---

**Status**

<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>80</td>
</tr>
<tr>
<td>Status::ClearModalStatus</td>
<td>81</td>
</tr>
<tr>
<td>Status::DisplayModalStatus</td>
<td>81</td>
</tr>
<tr>
<td>Status::GetDisplayTime</td>
<td>81</td>
</tr>
<tr>
<td>Status::SetBitmap</td>
<td>81</td>
</tr>
<tr>
<td>Status::SetDisplayTime</td>
<td>81</td>
</tr>
<tr>
<td>Status::SetText</td>
<td>82</td>
</tr>
</tbody>
</table>
**Status::Status**

Constructors for the Status class

Form 1: Status()

Form 2: Status(
    char const * const pDisplayString,
    int const DisplayTime = STATUS_DEFAULT_TIME)

Form 3: Status(
    BitMap const * const pBitmap,
    int const DisplayTime = STATUS_DEFAULT_TIME)

Form 4: Status(
    Bitmaps::PREDEFINED_BITMAP const pdb,
    int const DisplayTime = STATUS_DEFAULT_TIME)

Form 5: Status(
    char const * const pDisplayString,
    BitMap const * const pBitmap,
    int const DisplayTime = STATUS_DEFAULT_TIME)

Form 6: Status(
    char const * const pDisplayString,
    Bitmaps::PREDEFINED_BITMAP const pdb,
    int const DisplayTime = STATUS_DEFAULT_TIME)

**Parameters**

- **pDisplayString**
  The string to be displayed in the status box. Word-wrapping is done automatically, so there is no need to manually line break the string.

- **pBitmap**
  The pointer to a bitmap

- **pdb**
  One of the predefined bitmap shapes specified in PredefinedBitmaps.h:
  - BITMAPS::INFORMATION
  - BITMAPS::QUESTION
  - BITMAPS::EXCLAMATION
  - BITMAPS::HOURGLASS

**DisplayTime**

The amount of time (in 1/100 second intervals) the status is displayed

**Remarks**

These functions are the constructors. The status box is displayed immediately.

If STATUS_EXPLICIT_CLEAR is specified for the display time, the status box will remain on the display until ClearModalStatus() is called. If a different display time was specified (in 1/100 second intervals), the status box will automatically be removed after the specified period of time.
**Status::ClearModalStatus**
Removes the current status box from the display

```cpp
void ClearModalStatus();
```

**Remarks**
This function should be used only if the status box was created with a value of `STATUS_EXPLICIT_CLEAR` for the display time. If a different display time was specified, the UI Engine will automatically remove the status object from the screen when the time expires.

**Status::DisplayModalStatus**
Instructs the UI Engine to paint the status box on the screen and removes it after the specified display time

```cpp
void DisplayModalStatus();
```

**Remarks**
Used if the default constructor was invoked. This function displays the status box system-modally for the length of time specified. After the time has expired, the dialog box will be cleared from the screen.

**Status::GetDisplayTime**
Returns the length of time that the status box will be held on the display

```cpp
int GetDisplayTime() const;
```

**Return value**
The amount of time (in 1/100 second intervals) that the status will be displayed

**Remarks**
This function is used if the default constructor was invoked.

If `STATUS_EXPLICIT_CLEAR` is specified for the display time, the status box will remain on the display until `ClearModalStatus()` is called. If a different display time was specified (in 1/100 second intervals), the status box will automatically be removed after the specified period of time.

**Status::SetBitmap**
Sets the bitmap that will appear in the status box

- Form 1: `void SetBitmap(Bitmap const * const pBitmap)`
- Form 2: `void SetBitmap(Bitmap::PREDEFINED_BITMAP const pdb)`

**Parameters**
- `pBitmap` The pointer to a bitmap
- `pdb` One of the predefined bitmap shapes specified in `PredefinedBitmaps.h`
### Status class

- BITMAPS::INFORMATION
- BITMAPS::QUESTION
- BITMAPS::EXCLAMATION
- BITMAPS::HOURGLASS

**Remarks**
This function is used if the default constructor was invoked.

#### Status::SetDisplayTime

Sets the length of time that the status box will be held on the display

```cpp
void SetDisplayTime(int const DisplayTime)
```

**Parameters**
- `DisplayTime`
  
  The amount of time (in 1/100 second intervals) the status is displayed

**Remarks**
Used if the default constructor was invoked.

If STATUS_EXPLICIT_CLEAR is specified for the display time, the status box will remain on the display until `ClearModalStatus()` is called. If a different display time was specified (in 1/100 second intervals), the status box will automatically be removed after the specified period of time.

#### Status::SetText

Sets the text that will appear in the status box

```cpp
void SetText(char const * const pDisplayString)
```

**Parameters**
- `pDisplayString`
  
  The string to be displayed in the status box. Word wrapping is done automatically, so there is no need to manually line break the string.

**Remarks**
This function is used if the default constructor was invoked.
StringList class

StringList is a class derived from List. This is a list in which the members are constant strings that never change. Unlike List, the application does not need to implement the NowDisplaying member function.

StringList
StringList::StringList ................................................. 83
StringList::GetFieldType ............................................. 83
StringList::SetStringList ........................................... 84

StringList::StringList
 Constructors for the StringList class
Form 1: StringList()
Form 2: StringList ( char const *const * StringArray)
Parameters
StringArray
A pointer to an array of strings. The last entry must be 0.
src
An initialized StringList object.
Remarks
Form 3 of the constructor creates a new StringList object that is a duplicate of src.

StringList::GetFieldType
Returns the type of the current field
FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL );
Parameters
pDerived
A pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. This parameter is optional. It can be NULL. If the parameter is NULL, the function returns no information about the specific derived type.
Return value
The type of the base class of the current field (i.e. the type of the first class derived from the Field object)
Remarks
For a StringList object, the function will return LIST_FIELD, and pDerived will be filled in with STRING_LIST_FIELD.
See also
Field class
StringList::SetStringList

Sets the strings that will be shown in the list

Form 1: void SetStringList (char const * const * StringArray)
Form 2: void SetStringList(
    char const * const * StringArray,
    int nEntries)

Parameters

StringArray
    A pointer to an array of strings. In Form 1, the last entry must be 0
nEntries
    The number of entries in the string list
**Table class**

The **Table class** inherits from the **Field class**.

### Table Class Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table::Table</td>
<td>86</td>
</tr>
<tr>
<td>Table::AllowBorders</td>
<td>86</td>
</tr>
<tr>
<td>Table::DeleteColumnTable</td>
<td>87</td>
</tr>
<tr>
<td>Table::DeleteRowTable</td>
<td>87</td>
</tr>
<tr>
<td>Table::FindNextCell</td>
<td>88</td>
</tr>
<tr>
<td>Table::GetCaption</td>
<td>88</td>
</tr>
<tr>
<td>Table::GetFieldType</td>
<td>89</td>
</tr>
<tr>
<td>Table::GetNumCells</td>
<td>89</td>
</tr>
<tr>
<td>Table::GetNumColumns</td>
<td>89</td>
</tr>
<tr>
<td>Table::GetNumRows</td>
<td>89</td>
</tr>
<tr>
<td>Table::GetSelectedCell</td>
<td>90</td>
</tr>
<tr>
<td>Table::GetSelectedColumn</td>
<td>90</td>
</tr>
<tr>
<td>Table::GetSelectedRow</td>
<td>90</td>
</tr>
<tr>
<td>Table::GetTopColumn</td>
<td>90</td>
</tr>
<tr>
<td>Table::GetTopRow</td>
<td>90</td>
</tr>
<tr>
<td>Table::InsertColumn</td>
<td>91</td>
</tr>
<tr>
<td>Table::InsertRow</td>
<td>91</td>
</tr>
<tr>
<td>Table::IsCellSelectable</td>
<td>92</td>
</tr>
<tr>
<td>Table::NowDisplaying</td>
<td>92</td>
</tr>
<tr>
<td>Table::PutCell</td>
<td>93</td>
</tr>
<tr>
<td>Table::PutCell</td>
<td>94</td>
</tr>
<tr>
<td>Table::Redraw</td>
<td>94</td>
</tr>
<tr>
<td>Table::SelectingCell</td>
<td>95</td>
</tr>
<tr>
<td>Table::SetColumnWidth</td>
<td>95</td>
</tr>
<tr>
<td>Table::SetDrawFlags</td>
<td>96</td>
</tr>
<tr>
<td>Table::SetHighlight</td>
<td>96</td>
</tr>
<tr>
<td>Table::SetJustification</td>
<td>96</td>
</tr>
<tr>
<td>Table::SetProperties</td>
<td>97</td>
</tr>
<tr>
<td>Table::SetRowHeight</td>
<td>97</td>
</tr>
<tr>
<td>Table::SetSelectedCell</td>
<td>98</td>
</tr>
<tr>
<td>Table::SetTableSize</td>
<td>98</td>
</tr>
<tr>
<td>Table::SetTableStructure</td>
<td>98</td>
</tr>
<tr>
<td>Table::SetTopCell</td>
<td>99</td>
</tr>
<tr>
<td>Table::ShowCaptions</td>
<td>99</td>
</tr>
<tr>
<td>Table::UpdateCaption</td>
<td>99</td>
</tr>
</tbody>
</table>
Table::Table

Constructors for the Table class

Form 1: Table (void)

Form 2: Table (int const NumColumns, int const NumRows, int const ColumnWidth = DEF_COLUMN_WIDTH)

Form 3: Table (int const NumColumns, int const NumRows, bool AllowBorders, bool ShowCaptions = false, int const ColumnWidth = DEF_COLUMN_WIDTH)

Parameters

NumColumns
The number of columns in the table

NumRows
The number of rows in the table

ColumnWidth
The width of each column

AllowBorders
If set to TRUE, the UI Engine will leave a space around the edge of each cell for an optional border. The border style for every cell in the table can be specified by calling SetDrawFlags(); alternatively, each call to PutCell() can specify a border style for an individual cell.

ShowCaptions
If TRUE, the caption for the current cell will be displayed beneath the table

Remarks
Each column in the table is the same width. The default width, DEF_COLUMN_WIDTH, places four columns on the LCD at one time. By default, the row height is set to allow one line of text per row; it can be changed by calling SetRowHeight().

Table::AllowBorders

Instructs the UI Engine to leave a space around the edge of each cell for an optional border

void AllowBorders (bool AllowBorders = true)

Parameters

AllowBorders
If TRUE, the UI Engine will leave a space for a border, otherwise it will not
Remarks: The border style for every cell in the table can be specified by calling `SetDrawFlag();` alternatively, each call to `PutCell()` can specify a border style for an individual cell.

**Table::DeleteColumnTable:**

Removes a column from a table

```cpp
void DeleteColumn (  
    int const Column,  
    bool Redraw = true)
```

**Parameters**

- **Column**
  - The column to remove
- **Redraw**
  - If this parameter is set to TRUE, the UI Engine will redraw the portion of the table that is visible after the deletion is complete. Otherwise, it will not redraw the table.

**Remarks**

If many columns or rows are inserted or deleted at one time, it is more efficient to avoid redrawing the table until the insertions and deletions are all finished than to redraw after each insertion or deletion. It is, therefore, best to set Redraw to TRUE only on the last call to one of the insert or delete functions.

**Table::DeleteRowTable:**

Removes a row from a table

```cpp
void DeleteRow (  
    int const Row,  
    bool Redraw = true)
```

**Parameters**

- **Row**
  - The row to remove
- **Redraw**
  - If this parameter is set to TRUE, the UI Engine will redraw the portion of the table that is visible after the deletion is complete. Otherwise, it will not redraw the table.

**Remarks**

If many columns or rows are inserted or deleted at one time, it is more efficient to avoid redrawing the table until the insertions and deletions are all finished than to redraw after each insertion or deletion. It is, therefore, best to set Redraw to TRUE only on the last call to one of the insert or delete functions.
Table::FindNextCell
Find the next selectable cell

void FindNextCell (  
    int& Column,  
    int& Row,  
    MESSAGE const &msg)

Parameters

Column
When the function is called, this parameter specifies the column of the currently selected cell. When the function returns, this parameter specifies the column of the next selectable cell.

Row
When the function is called, this parameter specifies the row of the currently selected cell. When the function returns, this parameter specifies the row of the next selectable cell.

msg
The message that caused the table to search for a selectable cell

Remarks
This virtual function will be called whenever a trackwheel roll message is received by the UI Engine so the selection in the table can change appropriately. The default implementation will calculate the next selectable cell based on the properties of the table set in SetProperties () and the direction of the roll. You can override this function if a more specialized behaviour is required.

Table::GetCaption
A virtual member function that allows you to dynamically choose the caption to be displayed for the selected cell

char const * const GetCaption (int const SelectedColumn, int const SelectedRow)

Parameters

SelectedColumn
The column of the selected cell

SelectedRow
The row of the selected cell

Return value
The caption for the specified cell that should be displayed beneath the table

Remarks
This virtual function will be called each time a cell is selected.
Table::GetFieldType
Returns the type of the current field

FIELDTYPE GetFieldType (FIELDTYPE * pDerived = NULL)

Parameters
- pDerived: A pointer to a FIELDTYPE variable that will receive the specific derived type of the current field. This parameter is optional; it can be NULL. If the parameter is NULL, the function returns no information about the specific derived type.

Return value
The type of the base class of the current field (i.e. the type of the first class derived from the Field object)

Remarks
For a Table object, the function will return TABLE_FIELD, and pDerived will be filled in with NULL_FIELD (since Table is derived directly from Field).

Table::GetNumCells
Returns the number of cells in the table

void GetNumCells (  
    int *oNumColumns,  
    int *oNumRows)  

Parameters
- oNumColumns: A pointer to an integer variable which will receive the number of columns in the table
- oNumRows: A pointer to an integer variable which will receive the number of rows in the table

Table::GetNumColumns
Returns the number of columns in the table

int GetNumColumns (void) const

Return value
Number of columns in the table

Table::GetNumRows
Returns the number of rows in the table

int GetNumRows (void) const

Return value
Number of rows in the table
Table::GetSelectedCell
Returns the row and column of the selected cell

void GetSelectedCell (  
  int *Column,  
  int *Row)

Parameters  
Column
A pointer to an integer variable which will receive the column of the selected cell

Row
A pointer to an integer variable which will receive the row of the selected cell

Table::GetSelectedColumn
Returns the index of the selected column

int GetSelectedColumn (void) const

Return value
One-based index of the selected column

Table::GetSelectedRow
Returns the index of the selected row

int GetSelectedRow (void) const

Return value
One-based index of the selected row

Table::GetTopColumn
Returns the index of the column at the left side of the display

int GetTopColumn (void) const

Return value
One-based index of the column at the left of the display

Table::GetTopRow
Returns the index of the row at the top of the display

int GetTopRow (void) const

Return value
One-based index of the row at the top of the display.
Table::InsertColumn
Inserts a column into a table

```cpp
void InsertColumn(
    int const Column,
    bool Redraw = true)
```

**Parameters**
- **Column**
  The column before which to insert the new column

- **Redraw**
  If this parameter is set to `TRUE`, the UI Engine will redraw the portion of the table that is visible after the insertion is complete. Otherwise, it will not redraw the table.

**Remarks**
If many columns or rows are inserted or deleted at one time, it is more efficient to avoid redrawing the table until the insertions and deletions are all finished than to redraw after each insertion or deletion. It is, therefore, best to set `Redraw` to `TRUE` only on the last call to one of the insert or delete functions.

Table::InsertRow
Inserts a row into a table

```cpp
void InsertRow(
    int const Row,
    bool Redraw = true)
```

**Parameters**
- **Row**
  The row before which to insert the new column

- **Redraw**
  If this parameter is set to `TRUE`, the UI Engine will redraw the portion of the table that is visible after the insertion is complete. Otherwise, it will not redraw the table.

**Remarks**
If many columns or rows are inserted or deleted at one time, it is more efficient to avoid redrawing the table until the insertions and deletions are all finished than to redraw after each insertion or deletion. It is, therefore, best to set `Redraw` to `TRUE` only on the last call to one of the insert or delete functions.
**Table::IsCellSelectable**
Determines whether a cell in a table is selectable (can be highlighted)

```cpp
bool IsCellSelectable ( 
    int const Column, 
    int const Row)
```

**Parameters**
- Column
  The column of the cell that is being asked about
- Row
  The row of the cell that is being asked about

**Return value**
- TRUE if the specified cell is selectable, FALSE if not

**Remarks**
- By default, every cell in the table is selectable. This is a virtual function that allows you to specify which cells can receive a highlight and which cannot. The default implementation of `FindNextCall()` will skip over cells that return FALSE from `IsCellSelectable()`.

**Table::NowDisplaying**
A virtual member function that allows you to dynamically choose the information to be displayed in each cell

```cpp
void NowDisplaying ( 
    int const Column, 
    int const Row)
```

**Parameters**
- Column
  The column of the cell being displayed
- Row
  The row of the cell being displayed

**Remarks**
- This virtual function will be called each time a cell is displayed on the screen. In response to this function being called, you should call `PutCell()` to display the correct data.

**Remarks**
- The default implementation does nothing.
### Table::PutCell

Called from the `NowDisplaying()` function in order to show text inside a cell that is being drawn on the screen.

```cpp
void PutCell(
    char const * const Str,
    int const ColumnWidth = LCD_WIDTH,
    int const JustifyFlags = JF_DEFAULT,
    int const StrLength = INT_MAX,
    int const BorderFlags = DF_DEFAULT)
```

**Parameters**

- **Str**: The text string to display.
- **ColumnWidth**: The width of the current column being written to the cell.
- **JustifyFlags**: A combination of the flags `JF_HOR_LEFT`, `JF_HOR_CENTERED`, `JF_HOR_RIGHT`, `JF_VER_TOP`, `JF_VER_CENTERED`, `JF_VER_LEFT`, `JF_VER_RIGHT`, and `JF_VER_BOTTOM`. At most one of `JF_HOR_LEFT`, `JF_HOR_CENTERED`, or `JF_HOR_RIGHT` should be specified to display the text in the cell horizontally left, centered or right justified, respectively. At most, one of `JF_VER_LEFT`, `JF_VER_CENTERED`, or `JF_VER_RIGHT` should be specified to display the text in the cell vertically left, centered or right justified, respectively.
- **StrLength**: The maximum number of characters that should be copied from `Str`. The default value will display the entire string until a terminating `NULL` is reached, or until the edge of the cell is reached.
- **BorderFlags**: A combination of the flags `DF_LEFT`, `DF_RIGHT`, `DF_TOP`, `DF_BOTTOM`, and `DF_OUTLINE`. The flags `DF_LEFT`, `DF_RIGHT`, `DF_TOP`, and `DF_BOTTOM` instruct the UI Engine to display a border on the left, right, top, and bottom of every cell, respectively. The flag `DF_OUTLINE` is a combination of all four of the other values, and instructs the UI Engine to display a border around each cell.

**Remarks**

`PutCell()` can be called multiple times for the same cell; the specified text will be placed at the end of the last specified column width, and will extend to the edge of the cell or to the end of the specified column width.
Table::PutCell

Called from the `NowDisplaying()` function in order to show a bitmap inside a cell that is being drawn on the screen.

```cpp
void PutCell (
    BitMap const * const BitmapPtr,
    int const XPos = 0,
    int const YPos = 0,
    int const BorderFlags = DF_DEFAULT)
```

**Parameters**
- **BitmapPtr**
  Pointer to a bitmap structure that should be displayed in the cell
- **XPos**
  The x-coordinate of the position where the bitmap should be displayed, relative to the left edge of the cell
- **YPos**
  The y-coordinate of the position where the bitmap should be displayed, relative to the top edge of the cell
- **BorderFlags**
  A combination of the flags `DF_LEFT`, `DF_RIGHT`, `DF_TOP`, `DF_BOTTOM`, and `DF_OUTLINE`

**Remarks**
`PutCell()` can be called multiple times for the same cell. The specified bitmap will be placed at the specified position each time. If the bitmap does not fit inside the cell, it will be truncated on the right and the bottom. In the `BorderFlags` parameter, the flags `DF_LEFT`, `DF_RIGHT`, `DF_TOP`, `DF_BOTTOM`, instruct the UI Engine to display a border on the left, right, top, and bottom of every cell, respectively. The flag `DF_OUTLINE` is a combination of all four of the other values, and instructs the UI Engine to display a border around each cell.

Table::Redraw

Redraws the table

**Form 1:**
```cpp
void Redraw (void)
```

**Form 2:**
```cpp
void Redraw (
    int const Column,
    int const Row)
```

**Parameters**
- **Column**
  The column of the cell to redraw
- **Row**
  The row of the cell to redraw

**Remarks**
Form 1 redraws the entire visible portion of the table and Form 2 redraws only the specified cell in the table.
As the table is redrawn, the virtual member function `NowDisplaying()` will get called each time a cell needs to be drawn on the screen. A developer should derive a class from `Table` and define that function to insert the correct data by calling `PutCell()`.

**Table::SelectingCell**
Called immediately before the specified cell is selected

```cpp
void SelectingCell (int const Column, int const Row)
```

**Parameters**

- **Column**
  The column of the cell that is about to be selected

- **Row**
  The row of the cell that is about to be selected

**Remarks**
This is a virtual function that could be overridden if you need to perform any operations (such as updating a label) each time a cell is selected.

**Table::SetColumnWidth**
Sets the width of each column in the table

```cpp
void SetColumnWidth (int const ColumnWidth = DEF_COLUMN_WIDTH)
```

**Parameters**

- **ColumnWidth**
  The width of each column

**Remarks**
Each column in the table is the same width. The default width, `DEF_COLUMN_WIDTH`, places four columns on the LCD at one time.

**Table::SetDrawFlags**
Sets the border style for the entire table

```cpp
void SetDrawFlags (int const DrawFlags)
```

**Parameters**

- **DrawFlags**
  A combination of the flags `DF_LEFT`, `DF_RIGHT`, `DF_TOP`, `DF_BOTTOM`, and `DF_OUTLINE`

**Remarks**
These flags control the border style of every cell in the table. The flags `DF_LEFT`, `DF_RIGHT`, `DF_TOP`, and `DF_BOTTOM` instruct the UI Engine to display a border on the left, right, top, and bottom of every cell, respectively. The flag `DF_OUTLINE` is a combination of all four of the other values, and instructs the UI Engine to display a border around each cell.
**Table::SetHighlight**

Sets the highlight mode for the table.

```cpp
void SetHighlight (int const Highlight)
```

**Parameters**

- **Highlight**
  
  One of the constants:
  - HF_HIGHLIGHT_NONE
  - HF_HIGHLIGHT_OUTLINE
  - HF_HIGHLIGHT_CELL
  - HF_HIGHLIGHT_CORNERS
  - HF_HIGHLIGHT_ELBOW
  - HF_HIGHLIGHT_CROSS

**Remarks**

- HF_HIGHLIGHT_NONE will not highlight the selected cell in any way.
- HF_HIGHLIGHT_OUTLINE will highlight the selected cell by drawing an outline around the edge of the cell.
- HF_HIGHLIGHT_CELL will highlight the selected cell by inverting the entire area of the cell.
- HF_HIGHLIGHT_CORNERS will highlight the selected cell by placing a dot in each corner.
- HF_HIGHLIGHT_ELBOW will highlight the selected cell by drawing L-shaped brackets in each corner.

**Table::SetJustification**

Sets the justification mode for the entire table. Form 1 is deprecated, developers should use form 2.

```cpp
Form 1: void SetJustification (int const justify)
```

```cpp
Form 2: void SetJustification(JUSTIFICATION const justification)
```

**Parameters**

- **Justify**

  A combination of the flags:
  - JF_HOR_LEFT
  - JF_HOR_CENTERED
  - JF_HOR_RIGHT
  - JF_VER_TOP
  - JF_VER_CENTERED
  - JF_VER_LEFT
  - JF_VER_RIGHT
  - JF_VER_BOTTOM

**justification**

- One of: JUST_CENTER, JUST_LEFT, JUST_RIGHT, JUST_FULL

**Remarks**

- These flags control the justification of the text in every cell in the table. At most, one of JF_HOR_LEFT, JF_HOR_CENTERED, and JF_HOR_RIGHT should be specified to display the text horizontally left, center, or right justified, respectively. At most, one of JF_VER_LEFT, JF_VER_CENTERED, or JF_VER_RIGHT should be specified to display the text vertically left, centered, or right justified, respectively. The justification mode for each cell can be controlled individually in the call to PutCell().

Form 1 has been deprecated, developers should use Form 2.
**Table::SetProperties**
Sets the properties for the entire table

```cpp
void SetProperties (int const Properties)
```

**Parameters**
- **Properties**
  A combination of the flags `PF_TOP_BOTTOM_CHECK`, `PF_LEFT_RIGHT_CHECK`, `PF_ALIGN_CENTER`, and `PF_ALIGN_RIGHT`

**Remarks**
If `PF_TOP_BOTTOM_CHECK` is not specified and the user scrolls down past the bottom of the table, the cursor will appear at the top of the next column in the table. If the user scrolls up past the top of the table the cursor will appear at the bottom of the previous column. If `PF_TOP_BOTTOM_CHECK` is specified, the cursor will stop at the top and bottom of the table.

If `PF_LEFT_RIGHT_CHECK` is not specified and the user scrolls past the right edge of the table, the cursor will appear at the left side of the next row. If the user scrolls left past the left edge of the table, the cursor will appear at the right side of the previous row.

Only one of `PF_ALIGN_CENTER` and `PF_ALIGN_RIGHT` should be specified. They specify that the entire table should be displayed in the center or against the right side of the screen, respectively.

**Table::SetRowHeight**
Sets the height of each row in the table

```cpp
void SetRowHeight (int const RowHeight)
```

**Parameters**
- **RowHeight**
  The height of each row

**Remarks**
Each row in the table is the same height. The height should be no larger than `LCD_HEIGHT`, if no captions are used, or `LCD_HEIGHT` minus the height of the font, if captions are used.

**Table::SetSelectedCell**
Selects a particular cell in the table

**Form 1**
```cpp
void SetSelectedCell (int const Column,
                     int const Row)
```

**Form 2**
```cpp
void SetSelectedCell (int Offset)
```

**Parameters**
- **Column**
  The column of the cell to select
### Table Class

**Row**
- The row of the cell to select

**Offset**
- The offset of the cell to select

**Remarks**
- The cell in the upper left corner of the table is cell 0. The offset increases across the rows of the table. When the end of a row is reached, the next cell in the offset order is the leftmost cell in the row directly below.

#### Table::SetTableSize

Sets the number of rows and columns in the table

```cpp
void SetTableSize(
    int const NumColumns,
    int const NumRows)
```

**Parameters**
- NumColumns
  - The number of columns in the table
- NumRows
  - The number of rows in the table

#### Table::SetTableStructure

Sets the number of rows and columns in the table, and the width of each column

```cpp
void SetTableStructure(
    int const NumColumns,
    int const NumRows,
    int const ColumnWidth = DEF_COLUMN_WIDTH)
```

**Parameters**
- NumColumns
  - The number of columns in the table
- NumRows
  - The number of rows in the table
- ColumnWidth
  - The width of each column

**Remarks**
- Each column in the table is the same width. The default width, DEF_COLUMN_WIDTH, places four columns on the LCD at one time.
**Table::SetTopCell**

Specifies which cell should be placed in the top left corner of the display

```cpp
void SetTopCell (int const Column, int const Row)
```

**Parameters**

- **Column**
  - The column of the cell to place in the top left corner of the display
- **Row**
  - The row of the cell to select

**Table::ShowCaptions**

Instructs the UI Engine to leave a space below the table for a caption

```cpp
void ShowCaptions (bool ShowCaptions = true)
```

**Parameters**

- **ShowCaptions**
  - If TRUE, the UI Engine will leave a space for a caption, otherwise it will not

**Remarks**

The caption can be associated dynamically with each cell. In order to display the caption for a particular cell, the UI Engine will call the virtual member function `GetCaption()` specifying the currently selected row and column. You should derive a class from `Table` and define that function to return the correct caption.

**Table::UpdateCaption**

Virtual method called when the user has moved focus or when a paint method is called.

```cpp
virtual void UpdateCaption(int const column, int const row)
```

**Parameters**

- **column**
  - The new column with focus
- **row**
  - The new row with focus

**Remarks**

Use this function to display extra information about the cell that the user has cursored over.
Title class

The Title class is derived from the Label class. A Title is a Label that is displayed on the top line of the display.

The functions on the following pages are listed in alphabetical order.

Title

Title::Title ................................................................................................................... 100
Title::SetLocation ............................................................................................................. 101
Title::SetText ................................................................................................................. 101

Title::Title

Constructors for the Title class

Form 1: Title (
    int const XPosition = 0,
    int const Width = -1,
    char const * const pnewText = NULL,
    JUSTIFICATION justify = JUST_LEFT)

Form 2: Title (    char const * const pnewText,
    int const Xposition = 0,
    int const Width = -1,
    JUSTIFICATION justify = JUST_LEFT)

Parameters

pnewText
    The position where text starts (in pixels)
XPosition
    The position where text starts (in pixels) from top left hand corner
Width
    The width of the text (in pixels)
justify
    The justification of the text. One of JUST_CENTER, JUST_LEFT, JUST_RIGHT, JUST_FULL

Remarks

Titles always appear on the first line of the display. As indicated by the above parameters, there can be more than one title object on the top line. The RIM Wireless Handheld is 132 pixels wide and the average width of system font characters is 5 pixels.

Form 1 ignores the justify settings.
**Title::SetLocation**

Sets the location of the title bar on the top line of the display

```cpp
void SetLocation(
    int const XPosition,
    int const Width)
```

**Parameters**

- **XPosition**
  - The position where text starts (in pixels)
- **Width**
  - The width of the text (in pixels)

**Remarks**

Titles always appear on the first line of the display. As indicated by the above parameters, there can be more than one title object on the top line. The display on the pager-sized handheld device is 132 pixels wide and the average width of system font characters is 5 pixels.

**Title::SetText**

Sets the text that will appear on the top line of the display

**Form 1:**
```cpp
void SetText (char  const * const pnewText)
```

**Form 2:**
```cpp
void SetText (char const * const text,
              int length)
```

**Parameters**

- **PnewText**
  - The pointer to the text that appears in title text
- **text**
  - Pointer to the text that appears in the title
- **length**
  - Length of the text

**Remarks**

Titles always appear on the first line of the display. The text will be displayed in the associated offset and location that is passed in the `SetLocation` member function.

Form 2 ignores all ‘\0’ characters in the string.
TopBottomChoice class

**TopBottomChoice** is a class derived from Choice. It fixes the string array to be either Top or Bottom.

### TopBottomChoice

- **TopBottomChoice::TopBottomChoice**
  - Constructor for the TopBottomChoice class
  - Form 1: TopBottomChoice( char * pLabel, bool CurrentChoice )
  - Form 2: TopBottomChoice( const TopBottomChoice & src )

#### Parameters
- **pLabel**
  - The pointer to the label associated with the choice
- **CurrentChoice**
  - If this parameter is TRUE, the current value in the choice box is Top. If this parameter is FALSE, the current value in the choice box is Bottom.
- **src**
  - An initialized TopBottomChoice object.

#### Remarks
- Form 1 is a standard constructor.
- Form 2 creates a duplicate of the one indicated by src. This form is new for version 2.1.

### TopBottomChoice::operator=

- Sets one TopBottomChoice object equal to another TopBottomChoice object
- TopBottomChoice & operator = ( const TopBottomChoice & src )

#### Parameters
- **src**
  - An initialized TopBottomChoice object

#### Return value
- A TopBottomChoice object which is a duplicate of src

#### Remarks
- Sets the left side parameter to be a duplicate of src
- This operator is new for release 2.1.
TopBottomChoice::GetFieldType
Returns the type of the current field

FIELDTYPE GetFieldType ( FIELDTYPE * pDerived = NULL );

Parameters
- pDerived
  A pointer to a FIELDTYPE variable that will receive the specific
derived type of the current field. This parameter is optional. It can be
NULL. If the parameter is NULL, the function returns no information
about the specific derived type.

Return value
Returns the type of the base class of the current field (i.e. the type of the
first class derived from the Field object)

Remarks
For a TopBottomChoice object, the function will return CHOICE_FIELD,
and pDerived will be filled in with TOP_BOTTOM_CHOICE_FIELD.

TopBottomChoice::GetFlag
Gets the value displayed in the Top/Bottom Choice field

bool GetFlag()

Return value
TRUE if Top and FALSE if Bottom

TopBottomChoice::SetFlag
Sets the value displayed in the Top/Bottom Choice class

void SetFlag(bool newFlagValue)

Parameters
- newFlagValue
  If this parameter is TRUE, the value in the choice box is set to Top. If
this parameter is FALSE, the value in the choice box is set to Bottom.
UIEngine class

The UIEngine class is a required component for using the UI engine application or the APIs. This class is used to handle user input from the keyboard and trackwheel and to manage the display context.

UIEngine

UIEngine::UIEngine ................................................................. 104
UIEngine::ClearDialog ............................................................ 105
UIEngine::ClearMenu ............................................................. 105
UIEngine::ClearScreen ........................................................... 105
UIEngine::DisableScreen ....................................................... 105
UIEngine::GetUIVersion ....................................................... 105
UIEngine::HandleInput ......................................................... 106
UIEngine::Initialize ............................................................. 107
UIEngine::ProcessDialog ...................................................... 107
UIEngine::ProcessMenu ......................................................... 107
UIEngine::ProcessScreen ...................................................... 108
UIEngine::RestoreDisplayContext ......................................... 108
UIEngine::SetFont ............................................................... 108

Remarks

There should only be one UIEngine object per application. All input (typically from the keyboard device) to the UI Engine from a field object, a menu object, or a dialog object should come through the UIEngine object.

Where the input goes is determined by the following: If a Dialog is on the display (the user has performed a ProcessDialog()), the input will go there. Otherwise, if a menu is on the display (the user has performed a ProcessMenu()), it will go there. Otherwise, the input will go to the screen (the user has performed a ProcessScreen()). Remember a screen is composed of fields; therefore, the input will go to the field with focus.

Note

The only way for the UI Engine to receive input is through an application.

The UI Engine doesn’t retain the previous contents of the display when outputting to the screen. If the application puts up a screen (ProcessScreen()) then puts up a menu, (ProcessMenu()), the application must perform a ProcessScreen() again to see the original screen, even if the application clears the menu (ClearMenu()) or the menu object goes out of scope.
**UIEngine::ClearDialog**
Instructs the UI Engine to remove the current dialog box from the display
void ClearDialog()

Remarks
This function removes the application instantiated dialog on the display. A `ProcessDialog` must be called in order for the UI Engine to be aware of the dialog again.

**UIEngine::ClearMenu**
Instructs the UI Engine to remove the current menu from the display
void ClearMenu()

Remarks
This function removes the application instantiated menu on the display. A `ProcessMenu` must be invoked in order for the UI Engine to be aware of the menu again. While the `ClearMenu` function removes the menu from the display context, foreground applications will need to call `ProcessScreen` to remove the image of the menu from the screen.

**UIEngine::ClearScreen**
Instructs the UI Engine to remove the current screen from the display and to reset the fields on the screen
void ClearScreen()

Remarks
This function clears the application instantiated screen from the UI Engine. A `ProcessScreen` must be invoked in order for the UI Engine to be aware of it again.

**UIEngine::DisableScreen**
Prevents any updates to the screen from appearing on the display
void DisableScreen()

**UIEngine::GetUIVersion**
Returns information about the version of the UI Engine
void GetUIVersion(
    int *Major,
    int *Minor,
    int *Revision)

Parameters
Major
A pointer to the UI Engine major version number
Minor
A pointer to the UI Engine minor version number

Revision
A pointer to the UI Engine revision number

Remarks
This function returns the version of the UI Engine.

**UIEngine::HandleInput**
Call this function to handle any keypad messages received from the Operating System

RESULT HandleInput (MESSAGE &InputMessage)

Parameters

**Message**
A RIM MESSAGE which contains device events

Return value
This function returns one of five RESULT values:

<table>
<thead>
<tr>
<th>Return value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLICKED</td>
<td>This implies the user has clicked the trackwheel. Typically clicking the trackwheel implies the application will bring up a menu, or if there is a menu or dialog already up, it will disappear.</td>
</tr>
</tbody>
</table>
| CONTINUE                    | This implies the UI Engine handled this input. Typical examples of input the UI Engine handles include:  
|                             | • Input that is navigational in nature (such as rolling the trackwheel)  
|                             | • New key input if an edit field has the focus.                       |
| HIDE_MENU                   | The user has selected this menu item. Typically an application would go back to processing the screen under the menu. |
| UI_SWITCH_TO_BACKGROUND     | The UI Engine has switched the application to the background. This happens when the user selects another application from the menu (See “Menu class” on page 60.). Typically an application would clear the menu and go back waiting for another RIM message, when receiving this. |
**UIEngine::Initialize**
Sets up the UIEngine object and initializes it for first use

void Initialize ()

**Remarks**
This is the first function that should be called after the UI Engine is instantiated.

**UIEngine::ProcessDialog**
Instructs the UI Engine to display a dialog box on the screen

void ProcessDialog (Dialog &newDialog)

**Parameters**
- newDialog
  The application-instantiated dialog

**Remarks**
This dialog remains intact until either the application calls ClearDialog or the application invokes another ProcessDialog function. Remember the UI Engine only keeps track of one screen at a time.

**UIEngine::ProcessMenu**
Instructs the UI Engine to display a menu

void ProcessMenu (Menu &newMenu)

**Parameters**
- newMenu
  The application instantiated screen

**Remarks**
This function displays the application instantiated menu on the display. This menu remains displayed until either the application calls ClearMenu or the application invokes another ProcessMenu function. Remember the UI Engine only keeps track of one menu at a time.

---

<table>
<thead>
<tr>
<th>Return value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| UNHANDLED    | This implies the UI Engine did not know what to do with this input. Typical examples of this are:  
  - Receiving input from a non-keyboard device  
  - Typing a key when a field that has the focus does not accept keys (e.g., list fields) |
UIEngine::ProcessScreen
Instructs the UI Engine to paint or repaint the screen

```cpp
void ProcessScreen (Screen &newScreen)
```

**Parameters**
- `newScreen`
  
  The application instantiated screen

**Remarks**
This function displays the application-instantiated screen on the display. This screen remains intact until either the application calls `ClearScreen` or the application invokes another `ProcessScreen` function.

The UI Engine keeps track of only one screen at a time.

UIEngine::RestoreDisplayContext
Shows the display context of the current UI Engine object on the screen

```cpp
void RestoreDisplayContext ()
```

**Remarks**
This function restores the display context associated with the application when the application first instantiates the UI Engine. This allows one application to call another application (one thread to call a function in another thread) and use the latter display context.

UIEngine::SetFont
Porting Note: this was formerly `ReplaceFont`. Changes the font used by the UI Engine

- **Form 1:**
  ```cpp
  int SetFont ( int iFontIndex, 
              Font const *pNewFont );
  ```

- **Form 2:**
  ```cpp
  int SetFont ( int iFontIndex, 
              Font const *pNewLargeFont, 
              Font const *pNewSmallFont );
  ```

**Parameters**

- `iFontIndex`
  The index of the system font that should be replaced

- `pNewFont`
  A pointer to the font definition structure to use. It can be created with the LCDFonts.exe program. See the SDK User's Guide for more information on this program.

- `pNewLargeFont`
  A pointer to the large font definition to be associated with `iFontIndex`
pNewSmallFont

A pointer to the small font definition to be associated with the iFontIndex

Remarks  This function sets the font used by all UI Engine objects in the system, not just the current one. It is a global function.
API descriptions

UIEngine class
Index of APIs

BeepChoice
BeepChoice() 13
GetBeep() 13
GetFieldType() 14
SetBeep() 14

BusyStatus
Disable() 16
Draw() 16
Reset() 16
ResetBitmap() 17

Choice
ChangeChoice() 19
ChangeChoiceEx() 20
ChangeChoiceList() 21
Choice() 18
GetFieldType() 21
GetLabel() 22
GetNumEntries() 22
GetSelectedIndex() 22
GetSelectedValue() 22
GetValueAtIndex() 23
SetChoices() 23
SetLabel() 23
SetNumEntries() 24
SetNumericString() 24
SetSelectedIndex() 24
UpdatePtr() 25

ClickDialog
ClickDialog() 26
Go() 27
SetAnchorPoint() 27
SetBitmap() 27
SetDisplayString() 28

DecimalEdit
DecimalEdit() 29
GetNumber() 29
SetNumber() 29

Dialog
ClearDialog() 30
Dialog() 30
DisplayDialog() 31
SetAnchorPoint() 31
SetBitmap() 32
SetDisplayString() 32
SetText() 32
UpdatePtr() 33

Edit
AddProperties() 36
Append() 36
ClearProperties() 36
Delete() 37
EditText() 37
GetBookmark() 37
GetBuffer() 38
GetBufferLength() 38
GetBufferSize() 38
GetCursorOffset() 38
GetFieldLength() 38
GetLabel() 38
GetProperties() 39
GetStringLength() 39
Insert() 39
operator= 39
PutColumn() 39
PutSelection() 40
SetBookmark() 40
SetBuffer() 41
SetCursorOffset() 41
SetLabel() 42
SetProperties() 42
SetSelection() 44

Field
GetFieldLength() 45
IsDirty() 46
IsFocusVisible() 46
MarkAsClean() 46
MarkAsDirty() 46
Redraw() 46
SetJustification() 46

FieldManager
AddField() 48
BottomOfDisplay() 49
GetHeight() 50
GetLastField() 50
GetWidth() 50
Invalidated() 50
OnFieldUpdate() 50
PutFieldAtBottomOfDisplay() 51
PutFieldAtTopOfDisplay() 51
RemoveAllFields() 51
RemoveField() 51
ResetDimensionDependentData() 51
SetFieldWithFocus() 52
TopOfDisplay() 52

Label
SetText() 53

List
Delete() 53
GetColumnFont() 55
GetFieldLength() 55
GetFieldLength() 55
GetNumberOfEntries() 55
GetSelectedIndex() 56
GetTopLineEntry() 56
Insert() 56
List() 57
NowDisplaying() 56
PutColumn() 57
Redraw() 57
SetColumnFont() 58
SetEnableMultipleSelection() 58
SetNumEntries() 58
SetSelectedIndex() 58
SetTopLineEntry() 58

Menu
GetSelectedIndex() 61
GetTopIndex() 61
Menu() 60
SetMenuItems() 62
SetTopIndex() 63

OKDialog
operator = 65
Go() 66
OKDialog() 65
SetQuestion() 66

PropertyName()
49, 50, 61, 62, 63, 64, 70, 72, 77

RichText
GetAttributes() 67
GetBuffer() 67
GetBufferLength() 67
GetCursorOffset() 68
SetAttributes() 68
SetBuffer() 68
SetCursorOffset() 68
SetText() 69

Screen
ClearScreen() 71
Close() 71
DidSave() 71
GetUIEngine() 71
IsDisabled() 71
IsFieldDirty() 71
OnChangeOptions() 72
OnClose() 72
OnKey() 73
OnMenu() 73
OnMessage() 73
OnRoll() 74
OnSave() 74
Process() 74
RemoveLabel() 74
ResetScreen() 75
Save() 75
Screen() 70
SetMenu() 75
SetScreenDisabled() 76

Separator
GetFieldLength() 78
Separator() 77
## Index of APIs

<table>
<thead>
<tr>
<th>Status</th>
<th>ClearModalStatus() 81</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DisplayModalStatus() 81</td>
</tr>
<tr>
<td></td>
<td>GetDisplayTime() 81</td>
</tr>
<tr>
<td></td>
<td>SetBitmap() 81</td>
</tr>
<tr>
<td></td>
<td>SetDisplayTime() 82</td>
</tr>
<tr>
<td></td>
<td>SetText() 82</td>
</tr>
<tr>
<td></td>
<td>Status() 80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StringList</th>
<th>GetFieldType() 83</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SetStringList() 84</td>
</tr>
<tr>
<td></td>
<td>StringList() 83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>structures</th>
<th>TextRangeAttribute 69</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>FindNextCell() 88</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetCaption() 88</td>
</tr>
<tr>
<td></td>
<td>GetNumCells() 89</td>
</tr>
<tr>
<td></td>
<td>GetNumColumns() 89</td>
</tr>
<tr>
<td></td>
<td>GetNumRows() 89</td>
</tr>
<tr>
<td></td>
<td>GetSelectedColumn() 90</td>
</tr>
<tr>
<td></td>
<td>GetTopColumn() 90</td>
</tr>
<tr>
<td></td>
<td>GetTopRow() 90</td>
</tr>
<tr>
<td></td>
<td>IsCellSelectable() 92</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title</th>
<th>SetLocation() 101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SetText() 101</td>
</tr>
<tr>
<td></td>
<td>Title() 100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TopBottomChoice</th>
<th>GetFieldType() 103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetFlag() 103</td>
</tr>
<tr>
<td></td>
<td>operator= 102</td>
</tr>
<tr>
<td></td>
<td>SetFlag() 103</td>
</tr>
<tr>
<td></td>
<td>TopBottomChoice() 102</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UIEngine</th>
<th>ClearDialog() 105</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ClearMenu() 105</td>
</tr>
<tr>
<td></td>
<td>ClearScreen() 105</td>
</tr>
<tr>
<td></td>
<td>DisableScreen() 105</td>
</tr>
<tr>
<td></td>
<td>GetUIVersion() 105</td>
</tr>
<tr>
<td></td>
<td>HandleInput() 106</td>
</tr>
<tr>
<td></td>
<td>Initialize() 107</td>
</tr>
<tr>
<td></td>
<td>ProcessDialog() 107</td>
</tr>
<tr>
<td></td>
<td>ProcessMenu() 107</td>
</tr>
<tr>
<td></td>
<td>ProcessScreen() 108</td>
</tr>
<tr>
<td></td>
<td>RestoreDisplayContext() 108</td>
</tr>
<tr>
<td></td>
<td>SetFont() 108</td>
</tr>
<tr>
<td></td>
<td>UIEngine() 104</td>
</tr>
</tbody>
</table>
Index

A
about this guide 5
activating
  clickdialog boxes 27
adding
  edit properties 36
  list column 57
  screen field object 48
  screen label 70
allowing for table captions 99
appending edit text 36
B
beepchoice boxes
  functions 13
  getting the beep 13
  getting the field type 14
  setting the beep 14
bold text, typographical meaning of 5
buffer
  getting the edit 38
  getting the edit length 38
  setting the edit 41
busy status boxes
  disabling 16
  drawing 16
  functions 15
  resetting 16
  resetting the bitmap 17
C
changing the choice box 19, 20,
  21
choice boxes
  changing 19, 20, 21
  constructing 18
  entering numbers 24
  functions 18
  getting string associated
  with index 23
  getting the field type 21
  getting the label 22
  getting the number of entries 22
  getting the selected index 22
  getting the selected value 22
  setting the choices 23
  setting the label 23
  setting the number of entries 24
  setting the selected index 24
  updating pointer 25
choosing table captions 88
clearing
  edit properties 36
  screens 71
  UI dialog boxes 31, 105
  UI menu 105
  UI screen 75, 105
clickdialog boxes
  activating 27
  functions 26
  setting the anchor point 27
  setting the bitmap 27
  setting the display string 28
code
  typographical representation of 5
constructing
  choice boxes 18
  decimal edit fields 29
  edit fields 35
  menu items 60
  status boxes 80
  title bars 100
  topbottomchoice boxes 102
user interface 104
conventions, typographical 5
cursor offset
  getting the edit 38
  setting the edit 41
D
decimal edit fields
  constructing 29
  functions 29
  getting the number 29
  indicating content changes 77
  setting the number 29
deleting
  columns in tables 87
  edit fields 37
  fields 51
list items 55
rows in tables 87
screen labels 74
dialog boxes
  clearing 31, 105
  displaying 31
  functions 30
  setting fields 32
  setting the anchor point 31
  setting the bitmap 32
  setting the display string 32
  updating pointers 33
disabling
  busy status box 16
  UI screen 105
display context
  restoring 108
displaying
  busy status box 16
  current list 56
  dialog boxes 31
  menu items 64
  modal status box 81
  OK dialog boxes 66
  status box text 82

E
edit fields
  adding properties 36
  appending text 36
  clearing properties 36
  constructing 35
  deleting text 37
  getting bookmarks 37
  getting buffer size 38
  getting properties 39
  getting the buffer 38
  getting the buffer length 38
  getting the cursor offset 38
  getting the field type 38
  getting the label 39
  getting the string length 39
  inserting strings 39
  replacing the selection 40
  setting bookmarks 40
  setting properties 42
  setting the buffer 41
  setting the cursor offset 41
  setting the display string 32
  setting the label 42
  setting the selection 44
  exceptions, watchPuppy 15

F
field manager
  adding a field object 48
  functions 48
  getting field after pField 50
  getting field before pField 50
  getting field with focus 49
  getting height 50
  getting last field 50
  getting the first 49
  getting width 50
  invalidating 50
  monitoring focus changes 49
  notification of changed data 50
  putting fields at bottom of display 51
  putting fields at top of display 51
  removing all fields 51
  removing fields 51
  setting fields with focus 52
  setting focus at bottom 49
  setting focus at top 52
  determining changes 46
  functions 45
  getting type 45
  marking clean 46
  marking dirty 46
  setting justification 46

FieldType
  constants 45
  finding in tables
    next cell 88
  selectable cells 92
  fonts, setting 108
  functions
    beep choice field type 14
    beep choice beep 13
bookmarks 37
buffer size for edit fields 38
choice box label 22
choice field type 21
decimal edit field number 29
edit field buffer 38
edit field cursor offset 38
edit field label 39
edit field properties 39
edit field string length 39
edit field type 38
field with focus 49
first column in a table 90
first row in a table 90
first screen 49
focus changes 49
height 50
last field 50
length of edit field buffer 38
list field type 55
number of cells in a table 89
number of columns in a table 89
number of entries 22
number of list entries 55
number of rows in a table 89
screen field after pField 50
screen field before pField 50
selected cell in a table 90
selected choice box index 22
selected choice box value 22
selected column in a table 90
selected list index 56
selected menu index 61
selected row in a table 90
separator field type 78
status display time 81
string for index 23
string list field type 83
table field type 89
top list index entry 56
top menu index 61
topbottomchoice box flag 103
topbottomchoice field type 103
UI version 105
width 50
H
handling input 106
header files 7
I
initializing the user interface 107
input, handling 106
inserting
edit fields 39
list index entry 56
inserting columns in tables 91
inserting rows in tables 91
italic text, typographical meaning of 5
L
lists
adding a column 57
deleting items 55
displaying current 56
enabling multiple selection 58
functions 54
getting font for the column 55
getting the field type 55
getting the number of entries 55
getting the selected index 56
getting the top index entry 56
inserting an item 56
redrawing lines 57
setting the font for the column 58
setting the number of entries 58
setting the selected index 58
setting the top position 59
M
marking
field as clean 72
fields as dirty 72

menu items
clearing 105
creating 60
functions 60
getting selected index 61
getting top 61
hiding 61, 62
setting 62
setting the selected index 63
setting top 63
showing 64
O
OKdialog boxes
displaying 66
functions 65
Index

setting the text 66

P
processing
dialog boxes 107
menu items 107
screens 108

properties
getting for edit fields 39
setting for edit fields 42

R
redrawing
list lines 57
tables 94
removing
all fields 51
edit fields 37
fields 51
list items 55
screen labels 74

replacing the selection 40
resetting
bitmap in a busy status box 17
screen 75

resetting the busy status box 16
restoring the display context 108

richtext
functions 67
get attributes 67
get buffer 67
get buffer length 68
get cursor offset 68
set attributes 68
set buffer 68
set cursor offset 68
set label 69

S
screens
adding a label 70
clearing 71, 105
close 71, 72
disabling 71, 76, 105
dialog box display strings 32
description 15
descriptions 15
edit 34
functions 70
gets the associated UI 71
handling key input 73
implementing change options 72
is field dirty 71
marking fields as clean 72
marking fields as dirty 72
processing 74
receiving messages 73
removing labels 74
resetting 75
saving 71, 74, 75
selecting menu items 73
setting menus 75
title 100
trackwheel movement 74

SDK
components 5
selecting (in tables)
a cell 97
cells 95
information to be displayed 92

network 66

set anchor point for clickdialog boxes

S
sep
settings 66

screen

setting the text 66

P
processing
dialog boxes 107
menu items 107
screens 108

properties
getting for edit fields 39
setting for edit fields 42

R
redrawing
list lines 57
tables 94
removing
all fields 51
edit fields 37
fields 51
list items 55
screen labels 74

replacing the selection 40
resetting
bitmap in a busy status box 17
screen 75

resetting the busy status box 16
restoring the display context 108

richtext
functions 67
get attributes 67
get buffer 67
get buffer length 68
get cursor offset 68
set attributes 68
set buffer 68
set cursor offset 68
set label 69

S
screens
adding a label 70
clearing 71, 105
close 71, 72
disabling 71, 76, 105
dialog box display strings 32
description 15
descriptions 15
edit 34
functions 70
gets the associated UI 71
handling key input 73
implementing change options 72
is field dirty 71
marking fields as clean 72
marking fields as dirty 72
processing 74
receiving messages 73
removing labels 74
resetting 75
saving 71, 74, 75
selecting menu items 73
setting menus 75
title 100
trackwheel movement 74

SDK
components 5
selecting (in tables)
a cell 97
cells 95
information to be displayed 92

sep
settings 66
table borders 86, 95
table properties 97
table size 98
table structure 98
the text for OKdialog boxes 66
title location 101
title text 101
top cell in a table 99
top index 63
top line 59
topbottomchoice flag 103
showing
current list 56
dialog boxes 31
menu items 64
modal status box 81
status box text 82
text in table cells 94
status boxes
clearing modal 81
constructing 80
displaying modal 81
functions 79
getting the display time 81
purpose 79
setting bitmaps 81
setting text 82
setting the display time 82
stringlist items
functions 83
getting the field type 83
setting the stringlist 84
strings
getting length for the edit field 39
inserting 39

tables
allowing for captions 99
choosing captions 88
deleting columns 87
deleting rows 87
finding selectable cells 92
finding the next cell to select 88
functions 85
getting field type 89
getting first row 90
getting number of cells 89
getting number of columns 89
getting number of rows 89
getting selected cells 90
getting selected column 90
getting selected row 90
getting the first column 90
inserting columns 91
inserting rows 91
redrawing 94
selecting a cell 97
selecting cells 95
selecting information to be displayed 92
setting borders 86, 95
setting column width 95
setting highlight mode 96
setting justification 96
setting properties 97
setting row height 97
setting table size 98
setting table structure 98
setting the top cell 99
showing cell text 94
updating the caption 99

title bars
constructing 100
setting text 101
setting the location 101
topbottomchoice boxes
constructing 102
functions 102
getting the field type 103
getting the flag 103
setting the flag 103
typographical conventions 5

U
user interface
clearing dialog boxes 105
clearing menu items 105
clearing screens 105
constructing 104
disabling screens 105
functions 104
getting the version 105
handling input 106
initializing 107
processing dialog boxes 107
processing menu items 107
processing screens 108
restoring the display context 108
setting fonts 108

V
version, getting UI 105

W
watchPuppy 15
Index