

BlackBerry Software Development Kit

Version 2.5

UI Engine API Reference Guide

BlackBerry Software Development Kit Version 2.5 UI Engine API Reference Guide
Last modified: 6 June 2002

Part number: PDF-04641-001

At the time of publication, this documentation complies with RIM Wireless Handheld version 2.5.

© 2002 Research In Motion Limited. All Rights Reserved. The BlackBerry and RIM families of related marks, images and symbols are the exclusive properties of Research In Motion Limited. RIM, Research In Motion, 'Always On, Always Connected', the "envelope in motion" symbol and the BlackBerry logo are registered with the U.S. Patent and Trademark Office and may be pending or registered in other countries. All other brands, product names, company names, trademarks and service marks are the properties of their respective owners.

The handheld and/or associated software are protected by copyright, international treaties and various patents, including one or more of the following U.S. patents: 6,278,442; 6,271,605; 6,219,694; 6,075,470; 6,073,318; D445,428; D433,460; D416,256. Other patents are registered or pending in various countries around the world. Visit www.rim.net/patents.shtml for a current listing of applicable patents.

While every effort has been made to ensure technical accuracy, information in this document is subject to change without notice and does not represent a commitment on the part of Research In Motion Limited, or any of its subsidiaries, affiliates, agents, licensors, or resellers.

Research In Motion Limited
295 Phillip Street
Waterloo, ON N2L 3W8
Canada

Published in Canada

Contents

	About this guide.....	5
	Related documentation.....	5
CHAPTER 1	Application software design principles	7
	Design guidelines	7
	Style guide	9
CHAPTER 2	UI Engine API Reference	13
	BeepChoice	14
	Bitmask	18
	BusyStatus.....	21
	Choice	24
	ClickDialog	32
	DateTime	35
	Structures	36
	Functions.....	37
	DecimalEdit	42
	Dialog	44
	Edit.....	48
	EditDate.....	60
	EditTime.....	65
	Field	71
	FieldManager.....	83
	Label.....	92
	List.....	95
	Menu	101
	OKDialog	108
	RichText.....	110
	Structures	110
	Functions.....	111
	Screen.....	116

Separator	125
Status	128
StringList	133
Table.....	135
Title	152
TopBottomChoice	155
UIEngine	157
YesNoChoice	166
YesNoDialog.....	168
 Index of functions	 171
 Index	 175

About this guide

The User Interface Engine (UI Engine) application programming interface (API) is a set of classes that provides common aesthetics for all RIM Wireless Handheld applications. The UI Engine features menu and screen handling, user fields and dialog boxes. UI Engine 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.

Related documentation

Before you use this guide, you should be familiar with the following documentation. These other resources can help you develop C++ applications for the RIM Wireless Handheld.

- *BlackBerry SDK 2.5 Developer Guide*

The *BlackBerry SDK Developer Guide* explains how to use the BlackBerry SDK, with tutorials and sample code to demonstrate how to write handheld applications. For additional information, visit the BlackBerry Developer Zone at <http://www.blackberry.net/developers>.

- *OS API Reference Guide*

Several OS API classes provide base functionality of the UI Engine, including Lcd and Keypad.

About this guide

Chapter 1

Application software design principles

Design guidelines

To provide a consistent interface between RIM Wireless Handheld applications, consider the following guidelines when using UI Engine components. These guidelines are very general, and are meant to get you started with designing effective applications for the BlackBerry Wireless Handheld.

User control

The user should always be in control of the application.

Responsiveness

The user must be able to navigate and perform actions quickly. If there are long delays between the time the user performs an action and display of the action's results, users will not feel as if they have control.

Customization

Because preferences differ, users should have the ability to customize the interface (for example, what information to display in a list and the order in which information is displayed). However, the application should provide defaults that are helpful for the majority of users.

Perform tasks at hand

A good interface makes the users' tasks easier, rather than drawing attention to itself. The best interfaces are the ones that are

hardly noticed. You should make an effort for your application's interface to blend in with other BlackBerry application.

Intuitiveness

An interface should be as intuitive and straightforward as possible.

One of the most effective ways to create an intuitive interface is to use the object-action metaphor. Users select objects and perform actions on them (for example, user selects a mail message and reads it).

Consistency

All tasks between applications should be consistent. You must be aware of other applications being written for the device in order to keep your application's interface a consistent part of the whole user experience. This way, your users will know what to expect from your application's interface, even before they use it.

Additionally, consistency between applications enables you to reuse common interface elements.

Clarity

The interface should be clear, both visually and conceptually. Clarity is an especially important design aspect because of the handheld's small interface (132 by 65 pixels on the RIM 850/950, 160 by 160 pixels on the RIM 857/957).

- If the interface is jumpy (that is, causes distractions by breaking the visual flow), users have trouble maintaining control.
- When using icons, keep in mind that they will be small and if they are cluttered, it will not be clear to the user how the icon is related to the interface at hand.
- Do not fill the interface with excessive controls. Too many controls only emphasizes the small size of the display.
- Also, consider that RIM Wireless Handheld applications are not Windows applications.

Dialog and status boxes should contain brief and concise text. This text should not merely indicate to users what they have done incorrectly; dialogs should also indicate what action is necessary to correct the problem.

Aesthetics

The handheld is a two-way wireless device; it is not a desktop solution. Three-dimensional graphics or flashing cursors on the device display are inefficient and distracting. A simple design offers the best way to maximize the aesthetic appeal of the RIM Wireless Handheld application icons.

Restrict the changing information on the display to one item, and keep the users' focus on one place at a time. When presenting a lot of information to users, display the most important information first (for example, when presenting each address book entry, display the name fields first, followed by the email address, phone, fax and pager numbers fields, and leave the Notes field to the end).

Another way the UI Engine keeps the handheld aesthetically pleasing is by taking advantage of the vertical nature of the trackwheel. Menu items and lists are all presented in a vertical fashion.

Double-clicking (as opposed to 2 quick single clicks) is not a desirable input, based on the size of the device and the size of the trackwheel itself.

Feedback

Feedback is the application's response to user input (for example, the appearance of a menu when users click the trackwheel or display of characters when users add information to a field). Immediate feedback is essential and the application should respond quickly to commands. For instance, one of the learning curves for new users of the handheld is typing on the keyboard. Since keyboard typing may take a while to become accustomed to, new users quickly become frustrated if feedback from typing or invoking screens and dialogs is slow.

Forgiveness

The interface must allow users to change their minds and undo commands; essentially, it must be forgiving.

Users cannot be allowed to do anything destructive without being warned (for example, an action such as deleting a message or Address Book entry must be confirmed before the action occurs).

Common menu items are not placed in proximity to commands that undo the task at hand. For example, the **Cancel** menu item is not close to the **Hide menu** menu item).

Style guide

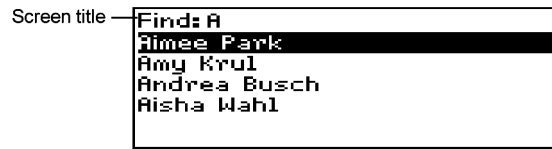
Certain UI Engine components are designed to have a specific use. This section outlines specific components and typical applications.

Screen title

Every screen will have a title on the first line of the display. The title line should not only inform the users where they are, but also display brief information to help users perform a particular task. For example, when users perform a search in the Address Book, the first line of the screen displays the title as well as the text that invoked the search.

Chapter 1: Application software design principles

For example, you can use the screen title to also display user input, such as “A” in the following screenshot, where the screen title displays “Find: A”:



Address book screen title

When users subsequently type ‘M’, the screen will display all of the records that start with ‘AM’ and the title will display Find: AM.

Menus

Invoking a menu

For consistency, a click of the trackwheel should always invoke a menu. Once a menu item is highlighted (by scrolling the trackwheel), the item can be selected (by clicking the trackwheel). Subsequently, a screen is displayed. Depending on the functionality of the item selected, this screen may be the same as the previous one.

Menu items

The first item in the menu is **Hide Menu**. The UI Engine automatically places this item at the top of the menu. When users select **Hide Menu**, the menu disappears and the previous screen remains on the display.

Another common menu item is **Cancel**. Choosing **Cancel** enables users to escape from the current task and return to a familiar screen. For example, suppose users select the Compose menu item from the menu associated with the Message List screen. They then select a recipient to whom to send the message, and start composing a message body. If users then decide not to send the message, they can invoke the menu and select **Cancel**. This returns the user to the message list screen.



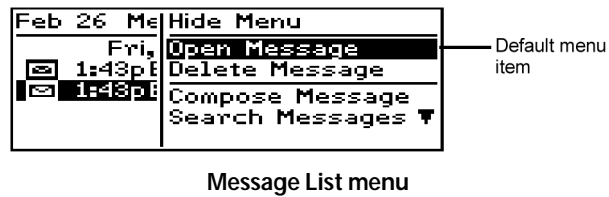
Note: **Cancel** may not be applicable in all cases. For example, **Cancel** is not applicable in the first screen of an application.

Menu format

The default item will be highlighted and should appear in the middle of the list of items when the menu first appears. The menu items that are selected more often by users should immediately surround the default item while the items that are less often selected should be placed further away from the default item.

Placing common menu options close to the default menu item minimizes excessive trackwheel scrolling by the user in order to locate the desired menu item.

For example, if the Message List screen is displayed on the LCD display and the user clicks the trackwheel, the menu looks like this:



Note: The default menu item should never be destructive. For example, the user should not be able to delete an entry of a message by clicking on the default menu item.

Controls

Controls are tools such as icons and buttons that the user selects to issue commands. Keep the number of controls on the display at any time as small as possible. Too many controls on the display emphasizes the small size of the display.

Keyboard and trackwheel

As much as possible, keep the user from having to move back and forth between the keyboard and the trackwheel. For example, if a user is entering data, he or she should not need to repeatedly switch between the track-wheel and the keyboard to perform the task.

Chapter 1: Application software design principles

Chapter 2

UI Engine API Reference

The UI Engine API contains the following classes. Constructors, where applicable, appear directly following the class description.

Classes	Page	Header file
BeepChoice	14	Choice.h
Bitmask	18	Bitmask.h
BusyStatus	21	BusyStatus.h
Choice	24	Choice.h
ClickDialog	32	ClickDialog.h Dialog.h
DateTime	35	DateTime.h
DecimalEdit	42	Edit.h
Dialog	44	Dialog.h
Edit	48	Edit.h
EditDate	60	DateTime.h
EditTime	65	DateTime.h
Field	71	Field.h

Classes	Page	Header file
FieldManager	83	Manager.h
Label	92	Label.h
List	95	List.h
Menu	101	Menu.h
OKDialog	108	OKDialog.h
RichText	110	RichText.h
Screen	116	Screen.h
Separator	125	Separator.h
Status	128	Status.h
StringList	133	List.h
Table	135	Table.h
Title	152	Title.h
TopBottomChoice	155	Choice.h
UIEngine	157	UIEngine.h
YesNoChoice	166	Choice.h
YesNoDialog	168	YesNoDialog.h

BeepChoice

BeepChoice is a class derived from Choice. It is related to the OS function `RimTestAlert`. A BeepChoice object is a one-line scrollable selection box that has a label and is accompanied by notification when it appears on the screen.

To include BeepChoice functions in your application, you must include `<Choice.h>` in your code.

The BeepChoice constructor has two forms:

```
Form 1: BeepChoice(
    char * Label,
    int CurrentChoice,
    bool NoOff = false)
```

Form 2: BeepChoice(const BeepChoice & src)

Parameters	Label	The label associated with the choice box.
	CurrentChoice	The same as the first parameter in RimTestAlert. One of: NO_NOTIFY, KILL_NOTIFY, TONE_SEQUENCE_1 through TONE_SEQUENCE_6.
	NoOff	True if notification is turned off; false otherwise.
	src	A reference to an initialized BeepChoice object.

Form 2 creates a BeepChoice object that is a duplicate of src.

Functions

The following functions are listed alphabetically.

BeepChoice::~BeepChoice	15
BeepChoice::GetBeep	15
BeepChoice::GetFieldType	16
BeepChoice::NowDisplaying	16
BeepChoice::Operator =	16
BeepChoice::SetBeep	16

BeepChoice::~~BeepChoice

Destroys an instance of a BeepChoice object.

```
virtual ~BeepChoice()
```

BeepChoice::GetBeep

Retrieves the value displayed in the BeepChoice field.

```
int GetBeep()
```

Returns The same as the first parameter in RimTestAlert. One of:

```
NO_NOTIFY
KILL_NOTIFY,
TONE_SEQUENCE_1
TONE_SEQUENCE_2
TONE_SEQUENCE_3
TONE_SEQUENCE_4
TONE_SEQUENCE_5
TONE_SEQUENCE_6
```

BeepChoice::GetFieldType

Retrieves 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. |
| Returns | | Returns the type of the base class of the current field (that is, the type of the first class derived from the Field object). |
| Description | | For a BeepChoice object, the function returns CHOICE_FIELD, and pDerived is filled in with BEEP_CHOICE_FIELD. |

BeepChoice::NowDisplaying

A virtual function called by the UI Engine when a particular entry needs to be drawn.

```
void NowDisplaying(int const index)
```

- | | | |
|-------------------|--------------|---|
| Parameters | index | The index of the BeepChoice object being updated. |
|-------------------|--------------|---|

BeepChoice::Operator =

Sets one BeepChoice object equal to another BeepChoice object.

```
BeepChoice & Operator = (const BeepChoice & src)
```

- | | | |
|--------------------|------------|---|
| Parameters | src | An initialized BeepChoice object. |
| Returns | | A BeepChoice object that is a duplicate of src. |
| Description | | BeepChoice::Operator = sets the left side parameter to be a duplicate of src. |

BeepChoice::SetBeep

Sets the value displayed in the BeepChoice field.

```
void SetBeep(int newBeepNumber)
```


BeepChoice

Parameters	newBeepNumber	The same as the first parameter in RimTestAlert. One of: NO_NOTIFY KILL_NOTIFY, TONE_SEQUENCE_1 TONE_SEQUENCE_2 TONE_SEQUENCE_3 TONE_SEQUENCE_4 TONE_SEQUENCE_5 TONE_SEQUENCE_6
-------------------	----------------------	---

Bitmask

The `Bitmask` class provides functions for manipulating bitmasks. A bitmask is a dynamically-sized construct.

To include `Bitmask` functions in your application, you must include `<Bitmask.h>` in your code.

The `Bitmask` constructor has four forms:

Form 1: `Bitmask()`

Form 2: `Bitmask(
 int * bitmaskPtr,
 unsigned int numberOfItems = 1,
 unsigned int size = 0)`

Form 3: `Bitmask(int bitmask)`

Form 4: `Bitmask(const Bitmask & src)`

Parameters	<code>bitmaskPtr</code>	A pointer to an array of integers used to initialize the bitmask: <ul style="list-style-type: none"> • The first element in the array represents the lowest order bit. • The last element in the array represents the highest order bit. • The 0th bit in the first element is the 0th bit.
	<code>numberOfItems</code>	The number of items in the array.
	<code>size</code>	The number of bytes in the bitmask. If this is 0, then the bitmask will be instantiated with the minimum number of bytes needed to fit the integer array.
	<code>bitmask</code>	A singular integer. This would be the equivalent of sending a 1 dimensional array with <code>numberOfItems</code> equal to 1.
	<code>src</code>	A reference to an initialized <code>Bitmask</code> object.

Form 4 creates a `Bitmask` object that is a duplicate of `src`.

Functions

The following interfaces are listed alphabetically.

<code>Bitmask::~~Bitmask</code>	19
<code>Bitmask::BlankBitmask</code>	19
<code>Bitmask::IsBitSet</code>	19
<code>Bitmask::LeftShift</code>	19
<code>Bitmask::SetBit</code>	19

Bitmask::SetBitmask	20
Bitmask::SetSize	20

Bitmask::~~Bitmask

Destroys an instance of a Bitmask object.

`~Bitmask()`

Bitmask::BlankBitmask

Erases the current bitmask.

`virtual bool BlankBitmask()`

Returns True if the bitmask was successfully erased; false otherwise.

Description All bits in the bitmask are set to 0.

Bitmask::IsBitSet

Determines if the requested bit is 1.

`bool IsBitSet(int bitPosition)`

Parameters `bitPosition` The bit to check.

Returns True if the requested bit is 1; false otherwise.

Bitmask::LeftShift

Shifts all the bits in the bitmask one position to the left.

`bool LeftShift()`

Returns True if the shift is successful; false otherwise.

Bitmask::SetBit

Sets the bit value to 1.

`bool SetBit(
 int bitPosition,
 bool set = true)`

Parameters `bitPosition` The bit to check.

`set` True to set the bit to 1; false otherwise.

Returns True if the operation is successful; false otherwise.

Bitmask::SetBitmask

Sets a specified number of bits in the bitmask to 1.

Form 1: `bool SetBitmask(
 const int * bitmask,
 unsigned int numberOfItems)`

Form 2: `bool SetBitmask(const Bitmask & src)`

Parameters	<code>bitmask</code>	A pointer to an array of integers to use to set the bit values.
	<code>numberOfItems</code>	The number of items in the bitmask.
	<code>src</code>	The number of items in the array.

Bitmask::SetSize

Sets the number of bytes in the bitmask.

`bool SetSize(unsigned int size)`

Parameters	<code>size</code>	The number of bytes.
-------------------	-------------------	----------------------

Returns True if the operation is successful; false otherwise.

BusyStatus

A BusyStatus object can be used during long operations to inform the user that an operation is still in progress. It resembles a regular status box, but 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...", 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 declare only one of these and draw it once, it doesn't show up, because there is no delay between calls.

To include BusyStatus functions in your application, you must include <BusyStatus.h> in your code.

Use the following constructor to create a BusyStatus object:

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

Parameters	InitString	The string to be displayed in the busy status box. If this parameter is NULL, no string will be displayed.
	InitTime	The length of time (in 1/100 second intervals) that the busy status box remains on the screen after a call to Draw, before being removed by the UI Engine.

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 will remain displayed throughout.

Functions

The following functions are listed alphabetically.

BusyStatus::~~BusyStatus	22
BusyStatus::Disable	22
BusyStatus::Draw	22
BusyStatus::NextBitmap	22
BusyStatus::Operator =	22
BusyStatus::Reset	23
BusyStatus::ResetBitmap	23

BusyStatus::~~BusyStatus

Destroys an instance of a BusyStatus object.

`~BusyStatus()`

BusyStatus::Disable

Disables the BusyStatus box.

`void Disable()`

Description The busy status box remains 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.

`void Draw()`

Description The BusyStatus object is 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::NextBitmap

Advances the bitmap to the next bitmap in the animation sequence.

`void NextBitmap()`

BusyStatus::Operator =

Sets one BusyStatus object equal to another BusyStatus object.

`BusyStatus & Operator = (const BusyStatus & src)`

Parameters `src` An initialized BusyStatus object.

Returns A BusyStatus object that is a duplicate of src.

Description BusyStatus::Operator = sets the left side parameter to be a duplicate of src.

BusyStatus::Reset

Resets the busy status box with a new string and a new display time.

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

Parameters	InitString	The string to be displayed in the busy status box. If this parameter is NULL, no string is displayed.
	InitTime	The length of time (in 1/100 second intervals) that the busy status box remains on the screen after a call to Draw, before being removed by the UI Engine.

Description The BusyStatus object is displayed when Draw is called, and remains 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.

```
void ResetBitmap()
```

Description The BusyStatus hourglass animation bitmap consists of seven individual bitmaps; this functions resets the animation sequence to the first bitmap.

Choice

A Choice object is a one-line scrollable selection box that has a label. Choice box labels are always left justified; options in a choice box is always right justified.

To include Choice functions in your application, you must include <Choice.h> in your code.

The Choice constructor has three forms:

Form 1: Choice()

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

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

Parameters	<code>pnewLabel</code>	A pointer to the choice box label. Labels are left justified.
	<code>newStringArray</code>	A pointer to a string array. This string is displayed right justified.
	<code>newStartValue</code>	The integer value to start a range.
	<code>newEndValue</code>	The integer value to end a range.
	<code>newIndex</code>	The starting value within the about range.
	<code>Increment</code>	The increment value in the above range.
	<code>justify</code>	The display justification of the data portion of the choice box.

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 two 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 is displayed (rolling down sets the next value, rolling up displays the previous value).

Pass an integer range and initial value. When the user performs ALT+ROLL, the next value in the range is displayed (rolling down sets the next value, rolling up displays the previous value).

Functions

The following functions are listed alphabetically.

Choice::~Choice	25
Choice::ChangeChoice	25
Choice::ChangeChoiceEx	26
Choice::ChangeChoiceList	27
Choice::GetFieldType	28
Choice::GetLabel	28
Choice::GetNewIndex	29
Choice::GetNumEntries	29
Choice::GetSelectedIndex	29
Choice::GetSelectedValue	29
Choice::GetValueAtIndex	29
Choice::PutData	30
Choice::SetChoices	30
Choice::SetLabel	30
Choice::SetNumEntries	30
Choice::SetNumericString	31
Choice::SetSelectedIndex	31
Choice::UpdatePtr	31

Choice::~Choice

Destroys an instance of a Choice object.

```
~Choice()
```

Choice::ChangeChoice

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

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

Parameters	<code>newCallingUI</code>	The UIEngine object associated with this choice box.
	<code>(*CallbackFunction)(int Value, MESSAGE & msg)</code>	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).
	<code>(*func)(int Value, void * tag)</code>	A callback function called when the choice value changes. The Value parameter is passed in as the tag value (and can accordingly be used for context).
Returns	One of UNHANDLED (implies that the BACKSPACE key was pressed) or CLICKED (the user either pressed the ENTER key or clicked the trackwheel).	
Description	The user can change the Choice Box choice by ALT+ROLL. ChangeChoice is an alternative which, when invoked, places a Dialog on the screen that uses another Choice Box (with 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::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	<code>newCallingUI</code>	The UIEngine object associated with this choice box.
-------------------	---------------------------	--

<code>(*CallbackFunction)(int Value)</code>	If the value is not NULL, this function is called each time the associated <code>ChoiceBox</code> value is changed. (The <code>Value</code> parameter of the callback function is the index of the currently selected choice item).
<code>(*func)(int Value, void* tag)</code>	A callback function called when the choice value changes. The <code>Value</code> parameter is passed in as the tag value (and hence can be used for context).
<code>TagValue</code>	Passed to the callback function as the tag value.

Returns One of UNHANDLED (implies the BACKSPACE key was pressed) or CLICKED (implies the user hit either the ENTER key or clicked the trackwheel).

Description One way the user changes the Choice Box choice is by pressing 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 `CallBackFunction`.

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	<code>newCallingUI</code>	The <code>UIEngine</code> object associated with this choice box.
-------------------	---------------------------	---

<code>(*CallbackFunction)(int Value)</code>	If the value is not NULL, this function is called each time the associated <code>ChoiceBox</code> value is changed. (The <code>Value</code> parameter of the callback function is the index of the currently selected choice item).
<code>(*func)(int Value, void* tag)</code>	A callback function called when the choice value changes. The <code>Value</code> parameter is passed in as the tag value (and hence can be used for context).
<code>TagValue</code>	Passed to the callback function as the tag value.

Returns One of `UNHANDLED` (implies the `BACKSPACE` key was pressed) or `CLICKED` (implies the user either pressed the `ENTER` key or clicked the trackwheel).

Description The user can change the Choice Box choice by pressing `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 `CallbackFunction`.

Choice::GetFieldType

Retrieves the type of the current field.

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

Parameters `pDerived` A pointer to a `FIELDTYPE` variable that receives 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.

Returns The type of the base class of the current field (that is, the type of the first class derived from the `Field` object).

Description 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

Retrieves a pointer to the text in the label currently displayed beside the `Choice` field.

```
char const * const GetLabel()
```

Returns The label associated with choice box.

Choice::GetNewIndex

Retrieves the index for a new Choice field item.

```
virtual bool GetNewIndex(char newChar, int & newIndex)
```

Parameters	newChar	The Choice item.
	newIndex	The index of either the string array or the index of the value, depending on what constructor is used.

Choice::GetNumEntries

Retrieves the number of entries in the choice field.

```
int GetNumEntries() const
```

Returns The number of entries associated with the Choice field.

Choice::GetSelectedIndex

Retrieves the index of the currently selected item in the Choice field.

```
int GetSelectedIndex() const
```

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

Choice::GetSelectedValue

Retrieves the numerical value or a pointer to the string currently selected in the Choice box.

```
Form 1: int GetSelectedValue()
```

```
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.
-------------------	---------------	---

Returns Form 1 returns the value of the currently-selected item. Form 2 has no return value.

Choice::GetValueAtIndex

Retrieves the string associated with the index.

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

Parameters	IndexValue	The index of the choice to query for its value.
-------------------	-------------------	---

Returns Returns the character string associated with the index.

Choice::PutData

Assigns data to a Choice field item.

```
void PutData(const char * const newData)
```

Parameters `newData` A pointer to the data to display.

Choice::SetChoices

Sets the list of choices to 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(  
    int newStartValue,  
    int newEndValue,  
    int newIndex = 0,  
    int Increment = 1)
```

Parameters	<code>newStringArray</code>	The string array containing choice box entries.
	<code>newIndex</code>	The index into the currently selected entry associated with <code>newIndex</code> .
	<code>newStartValue</code>	The start value.
	<code>newEndValue</code>	The end value.
	<code>Increment</code>	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.

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

Parameters `NumEntries` The number of entries available to the user.

Description Calling this function will not change the values of the entries in the choice box. `NumEntries` specifies the number of entries to be made 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

Enables the user to enter a number without having to press and hold the ALT or NUM key.

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

Parameters `Numeric` If true, the ALT or NUM key is not required to enter a number.

Description If `Numeric` is set to true, then a user may select a choice by typing the associated number without first activating the ALT or NUM key.

Choice::SetSelectedIndex

Instructs the UI Engine to display the item with the specified index in the Choice field.

```
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.

Description `SetSelectedIndex` sets the index of the choice box.

Choice::UpdatePtr

Updates RAM pointers associated with flash data.

```
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

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 a key.

To include ClickDialog functions in your application, you must include `<ClickDialog.h>` and `<Dialog.h>` in your code.

The ClickDialog constructor has four forms:

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)`

Form 4: `ClickDialog(const ClickDialog & src)`

Parameters	<code>newpDisplayString</code>	Pointer to the text string displayed in the dialog box.
	<code>newpBitmap</code>	Pointer to a bitmap structure that is to be displayed in the dialog box.
	<code>PredefinedBitmap</code>	One of the predefined bitmap shapes specified in <code>PredefinedBitmaps.h</code> : <ul style="list-style-type: none"> • <code>Bitmaps::INFORMATION</code> • <code>Bitmaps::QUESTION</code> • <code>Bitmaps::EXCLAMATION</code> • <code>Bitmaps::HOURLASS</code>
	<code>src</code>	A reference to an initialized ClickDialog object.

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 is cleared automatically when the user clicks the trackwheel or presses a key.

Form 4 creates a ClickDialog object that is a duplicate of `src`.

Functions

The following functions are listed alphabetically.

<code>ClickDialog::~ClickDialog</code>	33
<code>ClickDialog::Go</code>	33
<code>ClickDialog::Operator =</code>	33
<code>ClickDialog::SetAnchorPoint</code>	33

ClickDialog::SetBitmap	34
ClickDialog::SetDisplayString	34

ClickDialog::~ClickDialog

Destroys an instance of a ClickDialog object.

```
virtual ~ClickDialog()
```

ClickDialog::Go

Activates the dialog.

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

Parameters UIEngineToUse The UIEngine to use for rendering the component.

Returns The return value is always zero.

Description 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 a key.

ClickDialog::Operator =

Sets one ClickDialog object equal to another ClickDialog object.

```
ClickDialog & Operator = (const ClickDialog & src)
```

Parameters src An initialized ClickDialog object.

Returns A ClickDialog object that is a duplicate of src.

Description ClickDialog::Operator = sets the left side parameter to be a duplicate of src.

ClickDialog::SetAnchorPoint

Sets the number of pixels from the top of the screen at which the dialog box is to be displayed.

```
void SetAnchorPoint(int AnchorPoint)
```

Parameters AnchorPoint The number of pixels from the top of the screen at which the dialog box is to be displayed.

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

ClickDialog::SetBitmap

Sets the bitmap to be displayed in the click dialog box.

Form 1: `void SetBitmap(Bitmap const * const newpBitmap)`

Form 2: `void SetBitmap(
 Bitmaps::PREDEFINED_BITMAP
 const PredefinedBitmap)`

Parameters	<code>newpBitmap</code>	A pointer to a bitmap (custom or predefined).
	<code>PredefinedBitmap</code>	One of the predefined bitmap shapes specified in <code>PredefinedBitmaps.h</code> : <ul style="list-style-type: none">• <code>Bitmaps::INFORMATION</code>• <code>Bitmaps::QUESTION</code>• <code>Bitmaps::EXCLAMATION</code>• <code>Bitmaps::HOURLASS</code>

Description The bitmap is placed in a dialog box on the display.

ClickDialog::SetDisplayString

Sets the text that to be displayed in the click dialog box.

`void SetDisplayString(char const * const newpDisplayString)`

Parameters	<code>newpDisplayString</code>	A pointer to the string associated with the status box.
-------------------	--------------------------------	---

DateTime

The `DateTime` class handles the format of date and time representations. It makes use of two enumerated types, `DateFormat` and `TimeFormat`. Because `DateTime` makes use of a particular format string, the field characters are described before the APIs.

The `DateTime` class works with `TIME` structures, declared in `rim.h`.

To include `DateTime` functions in your application, you must include `<DateTime.h>` in your code.

Use the following constructor to create a `DateTime` object:

```
DateTime()
```

The format string

`DateTime` uses a character string similar to that used by `sprintf` to format the `TIME` object into the format desired by the user.

A `%` character precedes a formatting instruction. There may be field characters or special instructions after the `%` character.

Field	Meaning
a	Shortened version of the AM/PM field. AM is represented as "a" and PM is represented as "p".
A	An AM/PM field. Possible values are AM and PM and are determined from the <code>TIME</code> object.
d	Date field. (The numerical day of the month.)
H	24 hour field. 1PM will be displayed as 13.
h	12 hour field. 1PM will be displayed as 1.
M	Month string (January, February, etc.)
m	The minute value.
n	The numerical value of the month. (January = 1, etc.)
o	Shortened, 3 letter version of the month string. (January = Jan, etc.)
s	The seconds value.

Field	Meaning
w	The weekday. (Sunday, Monday, etc.)
y	The year.
z	The timezone.

Special characters may also follow the %:

Special	Meaning
–	Forces all the text for the following field to lower case.
^	Forces all the text for the following field to upper case.
0	Forces numbers to be zero filled in front if they are not as large as the specified field width.
digit	An optional digit string specifying the field width. Entries that are too large for this field width are concatenated.

Example For the TIME object as follows:

```
TIME myTime;
myTime.second = 5;
myTime.minute = 10;
myTime.hour = 7;
myTime.date = 17;
myTime.month = 7;
myTime.day = 1;
myTime.year = 2000;
myTime.TimeZone = 0;
```

The default format strings and the output are as follows:

```
DATE_LONG - "%3w, %3M %d, %y" - "Mon, Jul 17, 2000"
DATE_SHORT - "%y/%02n/%02d" - "2000/07/17"
TIME_LONG - "%2h:%02m %^aM" - "7:10 AM"
TIME_SHORT - "%2h:%02m%_a" - "7:10a"
```

Structures

DateTime::Time 36

DateTime::Time

The TIME structure is declared in Rim.h.

```
typedef struct
{
```

```

        BYTE    second;
        BYTE    minute;
        BYTE    hour;
        BYTE    date;
        BYTE    month;
        BYTE    day;
        WORD    year;
        BYTE    TimeZone;
    } TIME;

```

Field	Description
second	The number of seconds.
minute	The number of minutes.
hour	The number of hours.
date	The day number (as in 7 for the 7th of a month).
month	The numerical value of the month (Jan = 1, Feb = 2, ..., Dec = 12).
day	The day of the week (Sun = 0, Mon = 1, ..., Sat = 6).
year	The year.
TimeZone	Time zone relative to Greenwich Mean Time.

Functions

The following functions are listed alphabetically.

DateTime::FieldType	37
DateTime::FieldTypePosition	38
DateTime::Format	38
DateTime::NextFieldType	40
DateTime::NumberOfFields	40
DateTime::Operator =	40
DateTime::PrevFieldType	40
DateTime::SetFormat	41

DateTime::FieldType

Retrieves the field type based on the format string.

```

static EditDateTimeFieldType FieldType(
    char const * const format,
    int const Position)

```

Parameters `format` The character string used to format the date and time.

`Position` The field position to consider.

Returns One of:

DATE_FIELD_MONTH
DATE_FIELD_DAY
DATE_FIELD_YEAR
TIME_FIELD_HOUR
TIME_FIELD_MINUTE
TIME_FIELD_AMPM
UNKNOWN_FIELD

Description `Position` references the number of fields, not the character location. If `Position` is 3 then the field type of the third field is returned.

DateTime::FieldTypePosition

Retrieves the position of the requested field type.

```
static int FieldTypePosition(  
    char const * const format,  
    EditDateTimeFieldType CurrentType)
```

Parameters `format` The character string used to format the date and time.

`CurrentType` The type of field to retrieve. One of: DATE_FIELD_MONTH, DATE_FIELD_DAY, DATE_FIELD_YEAR, TIME_FIELD_HOUR, TIME_FIELD_MINUTE, TIME_FIELD_AMPM, UNKNOWN_FIELD.

Returns The position of the field in the string relative to the other fields or -1 if it does not exist.

Description The return value is not a reference to a specific character in the string, but is the number of fields prior to the requested one. So if the required field is the second field the return value will be 1.

DateTime::Format

Formats the TIME object in the manner specified by `format`.

```
Form 1: static char * Format(  
    TIME const datetime,  
    char const * const format,  
    char * const buffer,  
    int const length)
```

```
Form 2: static char * Format(  
    TIME const datetime,
```

```
DateUserFormat const format,
char * const buffer,
int const length)
```

```
Form 3: static char * Format(
    TIME const datetime,
    TimeUserFormat const format,
    char * const buffer,
    int const length)
```

```
Form 4: static char * Format(
    TIME const datetime,
    char const * const format,
    char * const buffer,
    int const length,
    int const index,
    bool before)
```

```
Form 5: static char * Format(
    TIME const datetime,
    DateFormat const format,
    char * const buffer,
    int const length)
```

```
Form 6: static char * Format(
    TIME const datetime,
    TimeFormat const format,
    char * const buffer,
    int const length)
```

Parameters	datetime	A TIME object with the required date and time information.
	format	The string (or a predefined string) which defines how to format the TIME object The values to pass for predefined strings are TIME_LONG, TIME_SHORT, DATE_LONG, DATE_SHORT.
	buffer	A pointer to the memory location in which to store the formatted data.
	length	The length of the buffer.
	index	A reference to a field index.
	before	A boolean value used to determine when to stop formatting.

Returns A pointer to the formatted data.

Description If format is one of DATE_LONG, DATE_SHORT, TIME_LONG, TIME_SHORT then one of the predefined formatting stirngs will be used.

The data will be formatted according to the supplied string or until the buffer runs out of room.

In Form 4, the data will be formatted until it has formatted a number of fields equal to index if before is true or 1 + index if before is false. These formatted fields do not include Weekday fields (Monday, etc) or TimeZone fields.

DateTime::NextFieldType

Retrieves the type of the field following CurrentType.

```
static EditDateTimeFieldType NextFieldType(  
    char const * const format,  
    EditDateTimeFieldType CurrentType)
```

Parameters	format	The character string used to format the date and time.
	CurrentType	The field type of the current field.

Returns The field type of the field occurring after CurrentType in the character string.

DateTime::NumberOfFields

Retrieves the number of fields in the character string.

```
static int NumberOfFields(char const * const format)
```

Parameters	format	The character string used to format the date and time.
-------------------	---------------	--

Returns The number of fields.

DateTime::Operator =

Sets one DateTime object equal to another DateTime object.

```
DateTime & Operator = (const DateTime & src)
```

Parameters	src	A reference to an initialized DateTime object.
-------------------	------------	--

Returns A DateTime object that is a duplicate of src.

Description DateTime::Operator = sets the left side parameter to be a duplicate of src.

DateTime::PrevFieldType

Retrieves the type of the field preceeding CurrentType.

```
static EditDateTimeFieldType PrevFieldType(  
    char const * const format,  
    EditDateTimeFieldType CurrentType)
```


- Parameters**
- `format` The character string used to format the date and time.
 - `CurrentType` The field type of the current field.
- Returns** The field type of the field occurring before `CurrentType` in the character string.

DateTime::SetFormat

Sets the predefined formatting strings to a user-determined value.

Form 1: `static void SetFormat(
 DateUserFormat format,
 char * fstring)`

Form 2: `static void SetFormat(
 TimeUserFormat format,
 char * fstring)`

- Parameters**
- `format` A value which determines which predefined string to change.
 - `fstring` The new value for the string.
- Description** The predefined strings can be accessed using the following values: `TIME_LONG`, `TIME_SHORT`, `DATE_LONG`, `DATE_SHORT`.

DecimalEdit

DecimalEdit is a class derived from Edit.

The DecimalEdit constructor has three forms:

Form 1: DecimalEdit()

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

Form 3: DecimalEdit(const DecimalEdit & src)

Parameters	pLabel	The label associated with the edit buffer.
	Current	The initial number associated with the buffer.
	justify	The justification.
	src	A reference to an initialized DecimalEdit object.

Form 3 creates a DecimalEdit object that is a duplicate of src.



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).

To include DecimalEdit functions in your application, you must include <Edit.h> in your code.

Functions

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

DecimalEdit::~DecimalEdit	42
DecimalEdit::GetNumber	42
DecimalEdit::Operator =	43
DecimalEdit::SetNumber	43

DecimalEdit::~DecimalEdit

Destroys an instance of a DecimalEdit object.

```
virtual ~DecimalEdit()
```

DecimalEdit::GetNumber

Retrieves the numeric value currently displayed in the DecimalEdit field.

```
int GetNumber()
```

Returns The current number in the DecimalEdit field.

DecimalEdit::Operator =

Sets one DecimalEdit object equal to another DecimalEdit object.

```
DecimalEdit & Operator = (const DecimalEdit & src)
```

Parameters `src` An initialized DecimalEdit object.

Returns A DecimalEdit object that is a duplicate of `src`.

Description `DecimalEdit::Operator =` sets the left side parameter to be a duplicate of `src`.

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

The Dialog class inherits from the FieldManager class.

To include Dialog functions in your application, you must include <Dialog.h> in your code.

The Dialog constructor has four forms:

Form 1: Dialog()

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

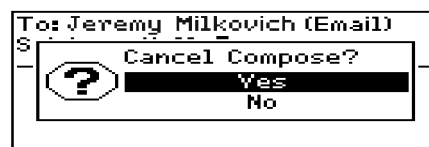
Form 3: Dialog(
 char const * const newpDisplayString,
 Bitmaps::PREDEFINED_BITMAP const PredefinedBitmap,
 Field * const pField=NULL)

Form 4: Dialog(const Dialog & src)

Parameters	newpDisplayString	A pointer to the string associated with the dialog box.
	newpBitmap	A 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: <ul style="list-style-type: none"> • Bitmaps::INFORMATION • Bitmaps::QUESTION • Bitmaps::EXCLAMATION • Bitmaps::HOURLASS
	src	A reference to an initialized Dialog object.

Form 4 creates a Dialog object that is a duplicate of src.

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.



A dialog box

Functions

The following functions are listed alphabetically.

Dialog::~Dialog	45
Dialog::ClearDialog	45
Dialog::DisplayDialog	45
Dialog::SetAnchorPoint	45
Dialog::SetBitmap	46
Dialog::SetDisplayString	46
Dialog::SetField	46
Dialog::SetFieldHeight	47
Dialog::UpdatePtr	47

Dialog::~Dialog

Destroys an instance of a Dialog object.

```
virtual ~Dialog()
```

Dialog::ClearDialog

Removes the dialog from the display.

```
void ClearDialog()
```

Description This function clears the dialog box from the display.

Dialog::DisplayDialog

Instructs the UI Engine to draw the dialog on the display.

```
virtual void DisplayDialog()
```

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

```
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 will be displayed.

```
void SetAnchorPoint(int AnchorPoint)
```

Parameters	AnchorPoint	The number of pixels from the top of the screen at which the dialog box will appear.
-------------------	-------------	--

Description If there is not enough room on the screen to show the entire dialog box starting at `AnchorPoint`, nothing will be displayed.

Dialog::SetBitmap

Sets the bitmap to display in the dialog box.

Form 1: `void SetBitmap(Bitmap const * const newpBitmap)`

Form 2: `void SetBitmap(
 Bitmaps::PREDEFINED_BITMAP
 const PredefinedBitmap)`

Parameters	<code>newpBitmap</code>	A pointer to a bitmap (custom or predefined).
	<code>PredefinedBitmap</code>	One of the predefined bitmap shapes specified in <code>PredefinedBitmaps.h</code> : <ul style="list-style-type: none">• <code>Bitmaps::INFORMATION</code>• <code>Bitmaps::QUESTION</code>• <code>Bitmaps::EXCLAMATION</code>• <code>Bitmaps::HOURLASS</code>

Description The bitmap is placed in a dialog box on the display.

Dialog::SetDisplayString

Sets the text to display in the dialog box.

`void SetDisplayString(char const * const newpDisplayString)`

Parameters	<code>newpDisplayString</code>	A pointer to the string associated with the status box.
-------------------	--------------------------------	---

Dialog::SetField

Sets the field to display in the dialog box.

```
void SetField(  
    Field & newpField,  
    int const newFieldHeight = -1,  
    int const newFieldWidth = -1)
```

Parameters	<code>newpField</code>	The field (typically Edit or List) that is part of the dialog.
	<code>newFieldHeight</code>	The height of the field.
	<code>newFieldWidth</code>	The width of the field.

Dialog::SetFieldHeight

Sets the height of the fields that appear in the dialog box.

```
void SetFieldHeight(int numOfFields)
```

Parameters	<code>numOfFields</code>	The number of fields in the dialog box.
-------------------	--------------------------	---

Description	<code>SetFieldHeight</code> determines the height of fields in order to fit them in a dialog box. The height of each actual field is dependant on the system font size; the height of the dialog box is determined by multiplying the field height by the number of fields.
--------------------	---

Dialog::UpdatePtr

Updates the RAM pointers associated with flash data.

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

Parameters	<code>newpDisplayString</code>	The new display string.
	<code>newpBitmap</code>	The bitmap to display. If NULL, any existing bitmap data is used.

Edit

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

To include `Edit` functions in your application, you must include `<Edit.h>` in your code.

The `Edit` constructor has four forms:

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	<code>pnewLabel</code>	The label associated with the field.
	<code>pnewBuffer</code>	A pointer to the application-defined buffer.
	<code>newLengthofBuffer</code>	The length of the buffer.
	<code>newCharacterOffset</code>	The cursor offset.
	<code>InitialSize</code>	The initial size of UI Engine-allocated buffer.
	<code>MaximumSize</code>	The maximum size of UI Engine-allocated buffer.
	<code>justify</code>	The justification of the edit buffer with respect to the screen.
	<code>TruncateIfFull</code>	The desired behaviour when the <code>Edit</code> field reaches the specified maximum size. If 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 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.
	<code>src</code>	A reference to an initialized <code>Edit</code> object.

Form 4 creates an `Edit` object that is a duplicate of `src`.

Two ways to use the `Edit` class are:

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

Functions

The following functions are listed alphabetically.

<code>Edit::~Edit</code>	49
<code>Edit::AddProperties</code>	50
<code>Edit::Append</code>	50
<code>Edit::ClearProperties</code>	50
<code>Edit::CopySelection</code>	50
<code>Edit::CutSelection</code>	51
<code>Edit::Delete</code>	51
<code>Edit::FocusAtEndOfData</code>	51
<code>Edit::GetBookmark</code>	51
<code>Edit::GetBuffer</code>	52
<code>Edit::GetBufferLength</code>	52
<code>Edit::GetBufferSize</code>	52
<code>Edit::GetCursorOffset</code>	52
<code>Edit::GetFieldType</code>	52
<code>Edit::GetLabel</code>	53
<code>Edit::GetProperties</code>	53
<code>Edit::GetStringLength</code>	53
<code>Edit::Insert</code>	53
<code>Edit::IsCopyAvailable</code>	53
<code>Edit::IsCutAvailable</code>	54
<code>Edit::IsPasteAvailable</code>	54
<code>Edit::Operator =</code>	54
<code>Edit::PasteSelection</code>	54
<code>Edit::ReplaceSelection</code>	54
<code>Edit::SetBookmark</code>	55
<code>Edit::SetBuffer</code>	55
<code>Edit::SetCursorOffset</code>	56
<code>Edit::SetLabel</code>	56
<code>Edit::SetProperties</code>	56
<code>Edit::SetSelection</code>	58
<code>Edit::UpdatePtr</code>	58

`Edit::~Edit`

Destroys an instance of an `Edit` object.

```
virtual ~Edit()
```

Edit::AddProperties

Adds the specified properties to the `Edit` field without clearing the properties that have already been set.

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

Parameters `FlagstoSet` The properties to add.

Description This function leaves the existing properties intact.
See `Edit::SetProperties` for the property definitions.

Edit::Append

Appends text to the end of the buffer.

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

Parameters `pInsertionString` A pointer of the insertion string to append.

 `Length` The length of the string. The default is the length of the insertion string.

Returns True if the text was appended to the buffer; false otherwise.

Edit::ClearProperties

Removes all special properties from the `Edit` field.

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

Parameters `FlagstoClear` The properties to clear.

Description See `Edit::SetProperties` for the definitions. This leaves all other properties intact.

Edit::CopySelection

Copies the currently selected text into memory.

```
bool CopySelection()
```

Returns True if the text was copied successfully; false otherwise.

Description The selected text remains available in memory until the handheld is reset, or the user copies (or cuts) a subsequent selection of text. The copied text can be pasted repeatedly, until the text is removed from memory.

Edit::CutSelection

Moves the currently selected text into memory.

```
bool CutSelection()
```

Returns True if the text was cut successfully; false otherwise.

Description The selected text is removed from its source. It remains available in memory until the handheld is reset, or the user cuts (or copies) a subsequent selection of text. The cut text can be pasted repeatedly, until the text is removed from memory.

Edit::Delete

Removes text from the buffer.

```
void Delete(
    int const Offset,
    int const Length)
```

Parameters

Offset	The label associated with the field.
Length	The length of the string in the edit buffer to delete.

Edit::FocusAtEndOfData

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

```
virtual bool FocusAtEndOfData()
```

Returns True if the cursor is at the end of the entered text in the buffer; 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.
----------------	---

Returns The current position of the character on which the specified bookmark was set.

Description 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

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

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

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

Edit::GetBufferLength

Retrieves the size of the buffer that stores the characters typed into the Edit field.

```
int GetBufferLength() const
```

Returns The length of the buffer.

Edit::GetBufferSize

Retrieves the buffer size.

```
int GetBufferSize() const
```

Returns The number of bytes available to the edit buffer.

Edit::GetCursorOffset

Retrieves the position of the cursor in the edit buffer.

```
int GetCursorOffset() const
```

Returns The current offset in the edit buffer of the cursor.

Edit::GetFieldType

Retrieves 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.

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

Description For an Edit object, the function returns EDIT_FIELD, and pDerived is filled in with NULL_FIELD (since Edit is derived directly from Field).

Edit::GetLabel

Retrieves the label currently displayed beside the edit field.

```
char const * const GetLabel()
```

Returns A pointer to the text of the label associated with the current edit field.

Edit::GetProperties

Retrieves the properties that the edit field is currently using.

```
unsigned long GetProperties()
```

Returns The edit buffer properties.

Description See `Edit::SetProperties` for property definitions.

Edit::GetStringLength

Retrieves the length of the string currently stored in the buffer.

```
int GetStringLength() const
```

Returns The length of the current string in the buffer.

Edit::Insert

Inserts text into the buffer.

```
bool Insert(
    char const * const pInsertionString,
    int const Position = -1,
    int const Length = -1)
```

Parameters	<code>pInsertionString</code>	A pointer to the insertion string.
	<code>Position</code>	The position of insertion.
	<code>Length</code>	The length of string. The default is the length of the insertion string.

Returns True if the text is inserted successfully; false otherwise.

Edit::IsCopyAvailable

Determines if the copy command can be performed on the handheld.

```
bool IsCopyAvailable()
```

Returns True if copy is available; false otherwise.

Edit::IsCutAvailable

Determines if the cut command can be performed on the handheld.

```
bool IsCutAvailable()
```

Returns True if cut is available; false otherwise.

Edit::IsPasteAvailable

Determines if the paste command can be performed on the handheld.

```
bool IsPasteAvailable()
```

Returns True if paste is available; false otherwise.

Edit::Operator =

Sets one Edit object equal to another Edit object.

```
Edit & Operator = (const Edit & src)
```

Parameters `src` An initialized Edit object.

Returns An Edit object that is a duplicate of `src`.

Description `Edit::Operator =` sets the left side parameter to be a duplicate of `src`.

Edit::PasteSelection

Copies the text currently in memory into the insertion point.

```
bool PasteSelection()
```

Returns True if the text was pasted successfully; false otherwise.

Description To paste text, the user must first copy or cut text into memory. The insertion point is the current position of the cursor in the field. Text cannot be pasted if an editable field is not in focus.

Edit::ReplaceSelection

Replaces the current selection with the specified string.

```
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 replacements string. If Length is -1, the entire string will be used up to a terminating NULL character.
Description	This function should be used together with <code>SetSelection</code> to replace one large block of text with another. First, set the selection using <code>SetSelection</code> and then replace it using <code>ReplaceSelection</code> .	

Edit::SetBookmark

Sets a bookmark on a specific character in the edit buffer.

```
void 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 <code>Buffer::NUMBER_OF_BOOKMARKS</code> 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 <code>USER_DEFINED_BOOKMARK_NUMBER</code> . You should only call this function with that value.

Description	A bookmark may be used to keep track of a character in the buffer. It remains with the character on which it is 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: void SetBuffer(
    char * const pnewBuffer,
    int const newLengthofBuffer)

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

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

Parameters	<code>pnewBuffer</code>	A pointer to the application buffer used in the <code>Edit</code> field.
	<code>newLengthofBuffer</code>	The length of above buffer.
	<code>InitialSize</code>	The initial size of the UI Engine allocated buffer.
	<code>MaximumSize</code>	The maximum size of the UI Engine allocated buffer.
	<code>TruncateIfFull</code>	The desired behaviour when the <code>Edit</code> 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.
Returns	True if memory could be allocated for the buffer; false otherwise.	
Description	Use form 1 when the buffer is allocated by the application and is read/write. Use form 2 when the buffer is allocated by the application and is read-only. Use form 3 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.

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

Parameters	<code>newCharacterOffset</code>	The offset in the <code>Edit</code> buffer where the character cursor is set.
Description	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.

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

Parameters	<code>pnewLabel</code>	A pointer to the label.
-------------------	------------------------	-------------------------

Edit::SetProperties

Sets the properties of the `Edit` field, including what types of input the field will accept.


```
void SetProperty(
    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.

Description 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.

Field Property	Description
AUTO_SPACE_SUBSTITUTION	Internal to the UI Engine.
CR_TO_ROLL_DOWN	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.
DISABLE_AUTO_FORMATTING	Bitwise OR of DISABLE_POSITION_AUTO_CAP and DISABLE_POSITION_AUTO_PUNC.
DISABLE_POSITION_AUTO_CAP	Disables the property of capitalizing the word after a punctuation mark followed by a space.
DISABLE_POSITION_AUTO_PUNC	Disables the property of substituting 2 spaces with period then space.
DISABLE_PRESS_HOLD_AUTO_CAP	Disables the property of pressing and holding an alpha key down to automatically capitalize it.
EMAIL_FIELD	Bitwise OR of AUTO_SPACE_SUBSTITUTION, CTYPE : : EMAIL_ADDRESS, and DISABLE_AUTO_FORMATTING.
HANGING_INDENT	Indents the edit buffer.
PASSWORD_FIELD	Substitutes every character entered with an asterisk.
READ_ONLY	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.

Character property	Description
ALPHABETIC	A-Z, a-z
ALPHANUMERIC	ALPHABETIC + DECIMAL_NUMERIC
DECIMAL_NUMERIC	0-9
EMAIL_ADDRESS	All characters except specials, CTRLs, and SPACE (except '@' and '.').
EMAIL_WORD	All characters except specials, CTRLs, and SPACE.
HEXIDECIMAL_NUMERIC	0-9,a-f
PHONE	0-9, (,), -,+,.,X,X
URL	URL characters (a-z,0-9,@,.,#,,?,,=

Edit::SetSelection

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

```
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.

Description SetSelection 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.

Edit::UpdatePtr

Updates the RAM pointers associated with flash data.

```
void UpdatePtr(
    char const * pNewBufferLoc,
    char const * pNewLabelLoc = 0)
```

Parameters	<code>pNewBufferLoc</code>	The location of the data to update in flash; if set to NULL, the data is assumed to be the same as in a previous call and is updated accordingly.
	<code>pNewLabelLoc</code>	The location of the label associated with the edit buffer; if set to NULL, data is assumed to be the same as in a previous call and is updated accordingly.

EditDate

The `EditDate` class handles an editable `Date` field.

To include `EditDate` functions in your application, you must include `<EditDate.h>` in your code.

The `EditDate` constructor has three forms:

Form 1: `EditDate()`

Form 2: `EditDate(`
 `char const * const pnewLabel,`
 `TIME const & datetime,`
 `DateFormat const format = DEF_DATE_FORMAT,`
 `int const newStartYear = 1998,`
 `int const newEndYear = 2029)`

Form 3: `EditDate(const EditDate & src)`

Parameters	<code>pnewLabel</code>	A pointer to the text to use as a label.
	<code>datetime</code>	A <code>TIME</code> object with the date information.
	<code>format</code>	One of: <ul style="list-style-type: none"> • <code>DEF_DATE_FORMAT</code> (Default date format) • <code>MMM_DD_YYYY</code> (Fri, January 31, 2001) • <code>YYYY_MM_DD</code> (2001/01/31) • <code>FORMAT_SHORT_MMM_DD_YYYY</code> (Fri, Jan 31, 2001)
	<code>newStartYear</code>	The minimum year to consider.
	<code>newEndYear</code>	The maximum year to consider.
	<code>src</code>	A reference to an initialized <code>EditDate</code> object.

Form 3 creates an `EditDate` object that is a duplicate of `src`.

Functions

The following functions are listed alphabetically.

<code>EditDate::~EditDate</code>	61
<code>EditDate::ChangeChoice</code>	61
<code>EditDate::FormatDate</code>	61
<code>EditDate::GetDate</code>	62
<code>EditDate::GetString</code>	62
<code>EditDate::GetFieldType</code>	62
<code>EditDate::IsValid</code>	63

```

EditDate::Operator = ..... 63
EditDate::SetDate ..... 63
EditDate::SetDateFormat ..... 63
EditDate::SetLabel ..... 64
EditDate::UpdatePtr ..... 64

```

EditDate::~EditDate

Destroys an instance of an EditDate object.

```
~EditDate()
```

EditDate::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 (*func)(int Value, MESSAGE & msg))

```

Parameters	newCallingUI	The UIEngine object associated with this choice box.
	(*func)(int Value, MESSAGE &msg)	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).

Returns One of:

- UNHANDLED
This implies the BACKSPACE key was pressed.
- CLICKED
The user either pressed the ENTER key or clicked the trackwheel.

Description The user can change the EditDate date by 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.

EditDate::FormatDate

Sets the format of the date used by the handheld.

```

Form 1: static char * FormatDate(
    TIME const datetime,
    char * const buffer = NULL,
    int const length = 0,
    DateFormat const format = DEF_DATE_FORMAT)

```

Chapter 2: UI Engine API Reference

```
Form 2: static char * FormatDate(  
    TIME const datetime,  
    DateUserFormat const format,  
    char * const buffer = NULL,  
    int const length = 0)
```

Parameters	datetime	A TIME object with the date information.
	buffer	The user provided storage location. You must provide a value for buffer; using the default value will throw an exception.
	length	The length of the buffer. You must provide a default value for length; using the default value will throw an exception.
	format	The method in which to format the data. One of: <ul style="list-style-type: none">• DEF_DATE_FORMAT (Default date format)• MMM_DD_YYYY (Fri, January 31, 2001)• YYYY_MM_DD (2001/01/31)• FORMAT_SHORT_MMM_DD_YYYY (Fri, Jan 31, 2001)

Returns A pointer to the formatted date.

EditDate::GetDate

Retrieves the date.

```
TIME GetDate()
```

Returns A TIME object which holds the last valid date held by the EditDate field.

EditDate::GetString

Retrieves the formatted data string associated with the EditDate field.

```
char * GetString(  
    char * const buffer = NULL,  
    int const length = 0)
```

Parameters	buffer	The user provided storage location.
	length	The length of the buffer.

Returns A pointer to the formatted date.

EditDate::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. If this optional parameter is NULL, the function returns no information about the specific derived type.
- Returns** Returns the type of the base class of the current field (i.e. the type of the first class derived from the `Field` object).
- Description** For an `EditDate` object, the function will return `DATE_FIELD`, and `pDerived` will be filled in with `NULL_FIELD` (since `EditDate` is derived directly from `Field`).

EditDate::IsValid

Determines if the date currently entered in the `EditDate` field is a valid date.

```
bool IsValid() const
```

- Returns** True if the date is valid; false otherwise.

EditDate::Operator =

Sets one `EditDate` object equal to another `EditDate` object.

```
EditDate & Operator = (const EditDate & src)
```

- Parameters** `src` An initialized `EditDate` object.
- Returns** An `EditDate` object that is a duplicate of `src`.
- Description** `EditDate::Operator =` sets the left side parameter to be a duplicate of `src`.

EditDate::SetDate

Sets the date associated with the `EditDate` field.

```
void SetDate(TIME const & datetime)
```

- Parameters** `datetime` A `TIME` object containing the appropriate date information.

EditDate::SetDateFormat

Sets the formatting associated with the `EditDate` field.

```
void SetDateFormat(DateFormat const format)
```

Parameters	<code>format</code>	One of: <ul style="list-style-type: none">• <code>DEF_DATE_FORMAT</code> (Default date format)• <code>MMM_DD_YYYY</code> (Fri, January 31, 2001)• <code>YYYY_MM_DD</code> (2001/01/31)• <code>FORMAT_SHORT_MMM_DD_YYYY</code> (Fri, Jan 31, 2001)
-------------------	---------------------	--

Description For more information, refer to “The format string” on page 35.

EditDate::SetLabel

Sets the label associated with the field.

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

Parameters	<code>pNewLabel</code>	A pointer to the text to be used as a label.
-------------------	------------------------	--

EditDate::UpdatePtr

Updates the RAM pointers associated with flash data.

```
void UpdatePtr(char const * const pLabel)
```

Parameters	<code>pLabel</code>	The data associated with the edit buffer; if set to <code>NULL</code> , data is assumed to be the same as in a previous call and is updated accordingly.
-------------------	---------------------	--

EditTime

The `EditTime` class is derived from the `DateTime` class and represents an editable time field.

To include `EditTime` functions in your application, you must include `<EditTime.h>` in your code.

The `EditTime` constructor has three forms:

Form 1: `EditTime()`

Form 2: `EditTime(
 char const * const pnewLabel,
 TIME & datetime,
 TimeFormat const format,
 int const justify = LCD_LEFT_JUSTIFIED,
 TimeZone const zone = GMT)`

Form 3: `EditTime(const EditTime & src)`

Parameters	<code>pnewLabel</code>	A pointer to the text to be used as a label.
	<code>datetime</code>	A <code>TIME</code> object containing the time information.
	<code>format</code>	The format for the time. One of: <ul style="list-style-type: none"> • <code>DEF_TIME_FORMAT</code> (Default time format) • <code>HOURL_12</code> (12 hour format / 2:34 PM) • <code>HOURL_24</code> (24 hour format / 14:34) • <code>HOURL_12_SH</code> (12 hour short format / 2:34p)
	<code>justify</code>	One of <code>LCD_CENTERED</code> , <code>LCD_LEFT_JUSTIFIED</code> , <code>LCD_RIGHT_JUSTIFIED</code> .
	<code>zone</code>	The time zone relative to Greenwich Mean Time.
	<code>src</code>	A reference to an initialized <code>EditTime</code> object.

For more information, refer to “The format string” on page 35.

Functions

The following functions are listed alphabetically.

<code>EditTime::~~EditTime</code>	66
<code>EditTime::ChangeChoice</code>	66
<code>EditTime::FormatTime</code>	66
<code>EditTime::GetDay</code>	67
<code>EditTime::GetFieldType</code>	67
<code>EditTime::GetTime</code>	68

EditTime::GetTimeFormat	68
EditTime::GetTimeString	68
EditTime::Operator =	68
EditTime::SetDay	68
EditTime::SetRound	69
EditTime::SetLabel	69
EditTime::SetTime	69
EditTime::SetTimeFormat	69
EditTime::SetTimeZone	69
EditTime::UpdatePtr	70

EditTime::~~EditTime

Destroys an instance of an EditTime object.

```
~EditTime()
```

EditTime::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 (*func)(int Value, MESSAGE &msg))
```

Parameters	NewCallingUI	The UIEngine object associated with this choice box.
	(*func)(int Value, MESSAGE &msg)	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).

Returns One of:

- UNHANDLED
This implies the BACKSPACE key was pressed.
- CLICKED
The user pressed either the ENTER key or clicked the trackwheel.

Description The user can change the EditTime date by 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.

EditTime::FormatTime

Sets the format of the time used by the handheld.

```
static char * FormatTime(TIME const datetime,
    char * const buffer,
    int const length = 0,
    TimeFormat const format = DEF_TIME_FORMAT)
```

Parameters	datetime	The TIME object with the date information.
	buffer	A user provided storage location.
	length	The length of the buffer.
	format	The format for the time. One of: <ul style="list-style-type: none"> • DEF_TIME_FORMAT (Default time format) • HOUR_12 (12 hour format / 2:34 PM) • HOUR_24 (24 hour format / 14:34) • HOUR_12_SH (12 hour short format / 2:34p)

Returns A pointer to the formatted time.

EditTime::GetDay

Get the stored day.

```
int GetDay()
```

Returns The stored day.

Description When editing an EditTime field, the time—the stored value—starts at 0. As the user moves the thumbwheel or increments/decrements the time, the number of hours may accumulate to positive or negative days. This accumulated value (the “days overflow”) is referred to as the stored day, and is returned by the GetDay() function.

EditTime::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.
-------------------	-----------------	--

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

Description For an `EditTime` object, the function will return `TIME_FIELD`, and `pDerived` will be filled in with `NULL_FIELD` (since `EditTime` is derived directly from `Field`).

EditTime::GetTime

Returns a `TIME` object with the displayed time.

```
TIME GetTime()
```

Returns A `TIME` object containing the displayed time.

EditTime::GetTimeFormat

Returns the current format associate with the `TimeEdit` field.

```
TimeFormat GetTimeFormat() const
```

Returns One of `DEF_TIME_FORMAT`, `HOURL_12`, `HOURL_24`, `HOURL_12_SH`.

Description For more information, refer to `EditTime()`.

EditTime::GetTimeString

Retrieves a pointer to the formatted time string.

```
char * GetTimeString(  
    char * const buffer = NULL,  
    int const length = 0)
```

Parameters `buffer` A user provided storage location.

`length` The length of the buffer.

Returns A pointer to the formatted time.

EditTime::Operator =

Sets one `EditTime` object equal to another `EditTime` object.

```
EditTime & Operator = (const EditTime & src)
```

Parameters `src` An initialized `EditTime` object.

Returns An `EditTime` object that is a duplicate of `src`.

Description `EditTime::Operator =` sets the left side parameter to be a duplicate of `src`.

EditTime::SetDay

Sets the stored day value.

```
void SetDay(int day)
```

Parameters `day` The new value of the stored day.

Description A call to `EditTime::GetDay` immediately after this call returns `day`. For more information, refer to `EditTime::GetDay`.

EditTime::SetRound

Sets the increment value for each time the user changes the minutes.

```
void SetRound(int round)
```

Parameters `round` The number of minutes to increment each time the user increments them.

EditTime::SetLabel

Sets the label associated with the `EditTime` field.

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

Parameters `pnewLabel` The text to appear in the label.

EditTime::SetTime

Sets the time associated with the `EditTime` field.

```
void SetTime(TIME const & datetime)
```

Parameters `datetime` The `TIME` object from which to retrieve the data.

EditTime::SetTimeFormat

Sets the method in which to format the time.

```
void SetTimeFormat(TimeFormat const format)
```

Parameters `format` One of `DEF_TIME_FORMAT`, `HOURL_12`, `HOURL_24`, `HOURL_12_SH`.

EditTime::SetTimeZone

Sets the timezone, relative to GMT.

```
void SetTimeZone(TimeZone const zone)
```

Parameters `zone` The timezone relative to Greenwich Mean Time.

EditTime::UpdatePtr

Updates the RAM pointers associated with flash data.

```
void UpdatePtr(char const * const pLabel)
```

Parameters	pLabel	The data associated with the edit buffer; if set to NULL, data is assumed to be the same as in a previous call and is updated accordingly.
-------------------	---------------	--

Field

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

To include `Field` functions in your application, you must include `<Field.h>` in your code.

The `Field` constructor is called by the `FieldManager`:

```
Field()
```

Functions

The following functions are listed alphabetically.

<code>Field::~Field</code>	72
<code>Field::GetArrows</code>	72
<code>Field::GetFieldType</code>	72
<code>Field::GetFocusRect</code>	73
<code>Field::GetFont</code>	73
<code>Field::GetHeight</code>	73
<code>Field::GetHeightMax</code>	73
<code>Field::GetHeightMin</code>	74
<code>Field::GetJustification</code>	74
<code>Field::GetManager</code>	74
<code>Field::GetRect</code>	74
<code>Field::GetStyles</code>	74
<code>Field::GetTag</code>	74
<code>Field::GetWidth</code>	75
<code>Field::GetWidthMax</code>	75
<code>Field::GetWidthMin</code>	75
<code>Field::GiveFocus</code>	75
<code>Field::HasFocus</code>	75
<code>Field::Invalidate</code>	76
<code>Field::IsDirty</code>	76
<code>Field::IsFocusDirty</code>	76
<code>Field::IsFocusVisible</code>	76
<code>Field::IsHidden</code>	76
<code>Field::IsIntegralHeight</code>	76
<code>Field::IsOnLCD</code>	77
<code>Field::IsReadOnly</code>	77
<code>Field::IsVisible</code>	77
<code>Field::MarkAsClean</code>	77
<code>Field::MarkAsDirty</code>	77
<code>Field::OnKey</code>	77
<code>Field::OnScroll</code>	78
<code>Field::PutFocusAtBottom</code>	78
<code>Field::QueryLineHeight (Deprecated)</code>	78
<code>Field::QueryPixelHeight (Deprecated)</code>	79
<code>Field::Redraw</code>	79

Chapter 2: UI Engine API Reference

Field::ResetDimensionDependentData	79
Field::SetArrows	79
Field::SetBounds	80
Field::SetFont	80
Field::SetIntegralHeight	80
Field::SetJustification	80
Field::SetManager	81
Field::SetTag	81
Field::SetVisible	81
Field::TakeFocus	82

Field::~~Field

Destroys an instance of a Field object.

`~Field()`

Field::GetArrows

Decides whether the scrolling arrows should be displayed.

`virtual void GetArrows(bool & UpDown, bool & LeftRight)`

Parameters	UpDown	Set to true if the up and down arrows should be displayed; false otherwise.
	LeftRight	Set to true if the left and right arrows should be displayed; false otherwise.

Description GetArrows is used by the Field Manager to determine when to show the scrolling arrows. This function can be overridden.

Field::GetFieldType

Retrieves the type of field.

`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.
-------------------	----------	--

Returns 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

```

GetFieldType is a pure virtual function, and must be overridden.

Field::GetFocusRect

Retrieves the XYRect in which the focus cursor should be drawn; this function can be overridden.

```
virtual XYRect GetFocusRect() const
```

Returns The XYRect object in which the cursor should be drawn.

Field::GetFont

Retrieves the font associated with the field.

```
Font const * GetFont() const
```

Returns A pointer to the font that will be used to display the characters in the field.

Field::GetHeight

Retrieves the height of the field.

```
Form 1: virtual int GetHeight() const
```

```
Form 2: virtual int GetHeight(int width) const
```

Parameters width The width of the field.

Returns The height of the field. The default implementation uses the value set in SetBounds. This function can be overridden.

Field::GetHeightMax

Retrieves the maximum height this field can have.

```
virtual int getHeightMax() const
```

Returns The maximum height allowable for this field.

Field::GetHeightMin

Retrieves the minimum height this field can have.

```
virtual int getHeightMin() const
```

Returns The minimum height allowable for this field.

Field::GetJustification

Retrieves the justification associated with the field.

```
JUSTIFICATION GetJustification() const
```

Returns One of: JUST_CENTER, JUST_LEFT, JUST_RIGHT, JUST_FULL

Description Use GetJustification to determine how text should be displayed in the field.

Field::GetManager

Retrieves the FieldManager associated with this field.

```
FieldManager * GetManager() const
```

Returns A pointer to the FieldManager of which this field is a member.

Description Retrieving the field manager is necessary when creating a field; the field should be drawn through the field manager using Draw.

Field::GetRect

Retrieves the XYRect where the field is drawn.

```
XYRect GetRect() const
```

Returns The XYRect object that defines the area in which the field is drawn.

Field::GetStyles

Retrieves the style bitmask.

```
int GetStyles() const
```

Returns The integer bitmask representing the style types used by the field.

Field::GetTag

Retrieves the tag associated with the field.

```
int GetTag()
```

Returns The tag of the field.

Description This value is set using SetTag. The default value of the field is -1.

Field::GetWidth

Retrieves the width of the field.

form 1: `virtual int GetWidth() const`

form 2: `virtual int GetWidth(int height) const`

Parameters `height` The height of the field.

Returns The width of the field. The default implementation uses the value set in `SetBounds`. This function be overridden.

Field::GetWidthMax

Retrieves the maximum width this field can have.

`virtual int GetWidthMax() const`

Returns The maximum width allowable for this field.

Field::GetWidthMin

Retrieves the minimum width this field can have.

`virtual int GetWidthMin() const`

Returns The minimum width allowable for this field.

Field::GiveFocus

Gives the field the focus.

`virtual bool GiveFocus(int const DirectionMagnitude)`

Parameters `DirectionMagnitude` The number of shifts of the trackwheel. A negative number implies an upward scroll. A positive number is a downward scroll.

Returns True if the field has successfully gained focus, false otherwise. This function can be overridden.

Field::HasFocus

Determines if the field has the focus.

`bool HasFocus()`

Returns True if the field currently has the focus; false otherwise.

Field::Invalidate

Forces the UIEngine to redraw the field.

```
void Invalidate()
```

Field::IsDirty

Determines if the component has changed since the last redraw.

```
virtual bool IsDirty() const
```

Returns True if the component has changed since the last redraw and therefore requires a redraw to display new information; false otherwise.

Field::IsFocusDirty

Determines if the focus has changed since the last redraw.

```
bool IsFocusDirty() const
```

Returns True if the focus has changed since the last redraw; false otherwise.

Field::IsFocusVisible

Determines if the cursor is visible; this function can be overridden.

```
virtual bool IsFocusVisible()
```

Returns True if the cursor is visible; false otherwise

Field::IsHidden

Determines if the field is hidden.

```
bool IsHidden() const
```

Returns True if the field should not be shown on the display; false otherwise.

Description This is used by the UI Engine and Field Manager to determine if the field should be drawn.

This function is the opposite of `IsVisible()`.

Field::IsIntegralHeight

Determines if the field is an integral number of text lines in height.

```
bool IsIntegralHeight() const
```

Returns True if the field is an integral number of text lines high; false otherwise.

Description `IsIntegralHeight` could be used to determine which text lines to display on the screen.

Field::IsOnLCD

Determines if the field is currently displayed on the LCD.

```
bool IsOnLCD() const
```

Returns True if the field is displayed on the LCD; false otherwise.

Description If this is true then the field is visible to the user.

Field::IsReadOnly

Determines if the field is read-only.

```
bool IsReadOnly() const
```

Returns True if the field is read-only; false otherwise.

Description A read-only field does not accept user input.

Field::IsVisible

Determines if the field is to be displayed.

```
bool IsVisible() const
```

Returns Returns true if the field should be shown on the display (if its portion of the screen is visible); false otherwise.

Description This is used by the UI Engine and Field Manager to determine if the field should be drawn; if a field is not visible on the screen, it should not be drawn.

This function is the opposite of `IsHidden()`.

Field::MarkAsClean

Marks the field clean.

```
virtual void MarkAsClean()
```

Description `IsDirty()` returns false after a call to `MarkAsClean`.

Field::MarkAsDirty

Marks the field dirty.

```
virtual void MarkAsDirty()
```

Description `IsDirty()` returns true after a call to `MarkAsDirty`.

Field::OnKey

Handles user keyboard input for this field.

```
virtual bool 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.
Returns	True if the field has handled the input; false if the manager must handle it.	
Description	The default implementation ignores all input and returns false. OnKey can be overridden.	

Field::OnScroll

Handles user input with the trackwheel.

```
virtual int OnScroll(int axis, int directionMagnitude)
```

Parameters	axis	Either AXIS_X or AXIS_Y.
	directionMagnitude	The number of shifts of the trackwheel. A negative number implies an upward scroll. A positive number is a downward scroll.
Returns	The number of track wheel shifts not handled by the field.	
Description	The default value of this function returns directionMagnitude and does nothing else. OnScroll can be overridden.	

Field::PutFocusAtBottom

Places the focus at the bottom of the field or the screen, whichever comes first; this function can be overridden.

```
virtual void PutFocusAtBottom()
```

Description	If your code deals with focus in a special way, override this function.
--------------------	---

Field::QueryLineHeight (Deprecated)

Determines the height of a line - deprecated.

```
virtual int QueryLineHeight(int const MaximumRelevant) = 0
```

Parameters	MaximumRelevant	The maximum relevant return value.
Returns	The height of a line.	



Note: This is a deprecated function included only for backwards compatibility. You should give it a dummy implementation.

Field::QueryPixelHeight (Deprecated)

Determines the height of a field - deprecated.

```
virtual int QueryPixelHeight(
    FieldLocation const StartFrom,
    int const MaximumRelevant)
```

Parameters	StartFrom	The location from which to begin counting height.
	MaximumRelevant	The maximum relevant return value.

Returns The height of the field.



Note: This is a deprecated function included only for backwards compatibility. You should give it a dummy implementation.

Field::Redraw

Draws the component.

```
virtual void Redraw()
```

Description Redraw is responsible for displaying the component. It calls the Draw and Paint functions to actually render the component.

Field::ResetDimensionDependentData

Changes the dimensions of the field based on some general size change.

```
virtual void ResetDimensionDependentData() = 0
```

Description ResetDimensionDependantData is called by the FieldManager whenever there is a system wide change that affects the size of field. A common cause of such a change is a modification to the size of the system font.

ResetDimensionDependantData must be overridden.

Field::SetArrows

Determines if scrolling arrows should be shown.

```
virtual void SetArrows(bool UpDown = true, bool LeftRight = false)
```

Parameters UpDown If true then the up and down arrows will be displayed.
 LeftRight If true then the left and right arrows will be displayed.

Description A subsequent call to `GetArrows` returns the values passed in by this function.

Field::SetBounds

Sets the area in which to draw the field.

```
void SetBounds(XYRect const & rect)
```

Parameters XYRect The boundaries of the field.

Description The default implementation of `GetHeight` and `GetWidth` use this function.

Field::SetFont

Sets the font used to display the field.

```
void SetFont(Font const * font)
```

Parameters font The font to use.

Description This font overrides the system font in the default implementation of `Field::GetFont`.

Field::SetIntegralHeight

Changes whether the field is an integral number of text lines in height.

```
void SetIntegralHeight(bool isIntegralHeight = true)
```

Parameters isIntegralHeight Set to true if the field is to be an integral number of text lines in height; false otherwise.

Field::SetJustification

Sets the justification mode for the text in the field.

Form 1: `void SetJustification(int const justification)`

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

Parameters	<code>newJustify</code>	One of: <ul style="list-style-type: none"> • LCD_LEFT_JUSTIFIED • LCD_RIGHT_JUSTIFIED • LCD_CENTERED
	<code>justification</code>	One of: <ul style="list-style-type: none"> • JUST_CENTER • JUST_LEFT • JUST_RIGHT • JUST_FULL

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

Field::SetManager

Sets the `FieldManager` associated with the field object; this function can be overridden.

```
virtual void SetManager(FieldManager * manager)
```

Parameters `manager` A pointer to the manager object containing the field.

Field::SetTag

Sets a tag associated with this field.

```
void SetTag(int newTag)
```

Parameters `newTag` Tag to associate with this field.

Description A tag set with this function can be accessed later using `Field::GetTag`. This would allow you to recognize a field even if you accessed it in a different way (such as `GetFieldWithFocus`)

Field::SetVisible

Switches the field between visible and invisible.

```
void SetVisible(bool showing)
```

Parameters `showing` Set to true if the field is to be visible; false otherwise.

Description An invisible field will not be displayed on the screen and cannot receive focus.

Field::TakeFocus

Removes the focus from the field; this function can be overridden.

```
virtual void TakeFocus()
```

FieldManager

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`.

To include `FieldManager` functions in your application, you must include `<Manager.h>` in your code.

Use the following constructor to create a `FieldManager` object:

```
FieldManager(const FieldManager & src)
```

Parameters `src` An instance of the `FieldManager` class.

Sets the `FieldManager` object to be a duplicate of `src`.

Functions

The following functions are listed alphabetically.

<code>FieldManager::~FieldManager</code>	84
<code>FieldManager::AddField</code>	84
<code>FieldManager::BottomOfDisplay</code>	84
<code>FieldManager::ChangeFocusRedraw</code>	84
<code>FieldManager::ClearFocus</code>	85
<code>FieldManager::Draw</code>	85
<code>FieldManager::GetBounds</code>	85
<code>FieldManager::GetBounds</code>	85
<code>FieldManager::GetFieldWithFocus</code>	85
<code>FieldManager::GetFieldwithFocus (Deprecated)</code>	85
<code>FieldManager::GetFirstField</code>	86
<code>FieldManager::GetHeight</code>	86
<code>FieldManager::GetLastField</code>	86
<code>FieldManager::GetNextField</code>	86
<code>FieldManager::GetPrevField</code>	87
<code>FieldManager::GetUIEngine</code>	87
<code>FieldManager::GetWidth</code>	87
<code>FieldManager::Invalidate</code>	88
<code>FieldManager::IsDisabled</code>	88
<code>FieldManager::MoveFocusRedraw</code>	88
<code>FieldManager::OnFieldUpdate</code>	88
<code>FieldManager::OnFocusEvent</code>	88
<code>FieldManager::Operator =</code>	89
<code>FieldManager::PageDownDisplay</code>	89
<code>FieldManager::PageUpDisplay</code>	89
<code>FieldManager::PutFieldAtBottomOfDisplay</code>	89
<code>FieldManager::PutFieldAtTopOfDisplay</code>	89
<code>FieldManager::RemoveAllFields</code>	90

FieldManager::RemoveField	90
FieldManager::ResetDimensionDependentData	90
FieldManager::SetFieldWithFocus	90
FieldManager::SetFieldwithFocus (Deprecated)	90
FieldManager::TopOfDisplay	91
FieldManager::UpdateNotify	91

FieldManager::~~FieldManager

Destroys an instance of a FieldManager object.

```
~FieldManager()
```

FieldManager::AddField

Adds a field to the field manager.

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

Parameters	newField	A field object (list, choice box, edit box, or table).
	pFieldAfterNewField	A pointer to the existing field you want to follow your field, or NULL (the default value for this parameter).
Description	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.

```
void BottomOfDisplay()
```

FieldManager::ChangeFocusRedraw

Redraws a field.

```
void ChangeFocusRedraw(
    Field & newFieldwithFocus,
    int const Magnitude,
    bool const ForwardDirection)
```

Parameters	<code>newFieldwithFocus</code>	The field to receive focus.
	<code>Magnitude</code>	The number of shifts of the trackwheel. A negative number implies an upward scroll. A positive number is a downward scroll.
	<code>ForwardDirection</code>	A boolean specifying if the field is moving forward (into focus) or backward (out of focus).

FieldManager::ClearFocus

Removes focus from a field.

```
void ClearFocus()
```

FieldManager::Draw

Called by the UI Engine when the FieldManager needs to be redrawn.

```
Form 1: virtual void Draw()
```

```
Form 2: virtual void Draw(Field * field)
```

Parameters	<code>field</code>	The field to be drawn.
-------------------	--------------------	------------------------

Description This function draws the graphical display. It can be overridden to provide your own custom look to a field manager.

FieldManager::GetBounds

Retrieves the coordinates of an area.

```
XYRect GetBounds() const
```

Returns The coordinates of an area (that is, two XYPoints).

FieldManager::GetFieldWithFocus

Returns a pointer to the field that currently has focus.

```
Field * GetFieldWithFocus()
```

Returns A pointer to the field on the screen that currently has the focus.

FieldManager::GetFieldwithFocus (Deprecated)

Returns a pointer to the field that currently has focus - deprecated.

```
Field * GetFieldwithFocus()
```

Returns A pointer to the field on the screen that currently has the focus.

Description This version is included only for backward compatibility. You should call `GetFieldWithFocus` instead.

FieldManager::GetFirstField

Returns a pointer to the field at the top of the field manager.

```
Field * GetFirstField()
```

Returns A 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
```

Returns The height of the field manager in pixels.

Description For more information, refer to `FieldManager::GetWidth`.

FieldManager::GetLastField

Get the last field in the field manager's field list.

```
Field * GetLastField()
```

Returns A pointer to the last field in the field manager.

FieldManager::GetNextField

Returns a pointer to the next field on the field manager

```
Form 1: Field * GetNextField()
```

```
Form 2: Field * GetNextField(Field * pField)
```

```
Form 3: Field * GetNextField(  
    Field * pSelectedField,  
    int propSet,  
    int propClear)
```

Parameters `pField` A pointer to a field.
`pSelectedField`

`propSet` The property to be set.

`propClear` The property to be cleared from the selected field.

Returns Form 1 returns a pointer to the first field.

Form 2 returns a pointer to the field after `pField` or NULL if no such field exists.

Form 3 returns a pointer to the first appropriate field after `pSelectedField` and it sets and clears the properties (bits) specified by `propSet` and `propClear`; if no such field exists, it returns NULL.

Returns A pointer to the field after `pField` or NULL if no such field exists.

FieldManager::GetPrevField

Returns a pointer to the previous field on the field manager.

Form 1: `Field * GetPrevField(Field * pField)`

Form 2: `Field * GetPrevField(Field * pSelectedField, int propSet, int propClear)`

Parameters	<code>pField</code>	A pointer to a field.
	<code>pSelectedField</code>	A pointer to the selected field.
	<code>propSet</code>	The property to be set.
	<code>propClear</code>	The property to be cleared from the selected field.

Returns Form 1 returns a pointer to pointer to the field before `pField` or NULL if no such field exists.

Form 2 returns a pointer to the first appropriate field before `pSelectedField` and it sets and clears the properties (bits) specified by `propSet` and `propClear`; if no such field exists, it returns NULL.

FieldManager::GetUIEngine

Returns the UIEngine associated with the screen.

`virtual UIEngine & GetUIEngine()`

Returns A reference to the UI Engine component associated with the screen.

FieldManager::GetWidth

Retrieves the width of the field manager in pixels.

`int GetWidth()const`

Returns The width of the field manager in pixels.

Description For more information, refer to `FieldManager::GetHeight`.

FieldManager::Invalidate

Invalidates the entire field manager.

```
virtual void Invalidate()
```

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

FieldManager::IsDisabled

Determines if the screen is disabled.

```
virtual bool IsDisabled()
```

Returns True if the screen is disabled; false otherwise.

Description If the FieldManager is disabled, then no changes to it are displayed until it is enabled again.

FieldManager::MoveFocusRedraw

Redraws a field.

```
void MoveFocusRedraw(int const DirectionMagnitude)
```

Parameters

DirectionMagnitude	The number of shifts of the trackwheel. A negative number implies an upward scroll. A positive number is a downward scroll.
--------------------	---

FieldManager::OnFieldUpdate

Notification that a field's data has changed; can override this function.

```
virtual void OnFieldUpdate(Field & field)
```

Parameters

field	A reference to the field whose data has changed.
-------	--

FieldManager::OnFocusEvent

Monitors focus changes for a given field.

```
virtual void OnFocusEvent(  
    FOCUS_EVENT event,  
    Field * field)
```

Parameters

event	One of FOCUS_GAINED or FOCUS_LOST.
field	The field to be monitored.

Description 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::Operator =

Sets one FieldManager object equal to another FieldManager object.

```
FieldManager & Operator = (const FieldManager & src)
```

Parameters `src` A reference to an initialized FieldManager object.

Returns A FieldManager object that is a duplicate of `src`.

Description `FieldManager::Operator =` sets the left side parameter to be a duplicate of `src`.

FieldManager::PageDownDisplay

Approximates scrolling down one page.

```
void PageDownDisplay()
```

Description This will work correctly as long as the font is consistent throughout all the fields and each field has only one scrollable item per line (unlike `DateTime`).

FieldManager::PageUpDisplay

Approximates scrolling up one page.

```
void PageUpDisplay()
```

Description This will work correctly as long as the font is consistent throughout all the fields and each field has only one scrollable item per line (unlike `DateTime`).

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 to 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 `newTopField` The field to 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 & newField)
```

Parameters `newField` A field object (menu, list, choice box, edit box, or table) to remove.

Description 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 functions of all child components.

```
virtual void ResetDimensionDependentData()
```

Description Call this function when system display parameters have changed. For instance, when system fonts have been changed from small to large.

FieldManager::SetFieldWithFocus

Instructs the UI Engine to give the focus to the specified field.

```
bool SetFieldwithFocus(Field * field)
```

Parameters `field` A pointer to a field object (menu, list, choice box, edit box, or table).

Returns True if successful or false if the field does not exist (for example, you did not invoke `AddField` member function).

Description 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.

FieldManager::SetFieldwithFocus (Deprecated)

Instructs the UI Engine to give the focus to the specified field - deprecated.

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

Parameters `pField` A pointer to a field object (menu, list, choice box, edit box, or table).

Returns True if successful or false if the field does not exist (i.e. you did not invoke `AddField` member function).

Description 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 a 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.

```
void TopOfDisplay()
```

Description For more information, refer to `FieldManager::BottomOfDisplay`.

FieldManager::UpdateNotify

Notifies the FieldManager that a field has been updated

```
void UpdateNotify(Field & field)
```

Parameters `field` A reference to the field whose data has changed.

Label

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`.

To include `Label` functions in your application, you must include `<Label.h>` in your code.

Form 1: `Label()`

Form 2: `Label(const Label & src)`

Parameters `src` An initialized `Label` object.

When using form 1, you must call `SetText` before the `Label` is displayed. For more information, refer to `Label::SetText`. Form 2 creates a `FieldManager` object that is a duplicate of `src`.

Functions

The following functions are listed in alphabetical order.

<code>Label::~~Label</code>	92
<code>Label::GetFieldType</code>	92
<code>Label::GetText</code>	93
<code>Label::GiveFocus</code>	93
<code>Label::Operator =</code>	93
<code>Label::SetText</code>	93
<code>Label::UpdatePtr</code>	93

Label::~~Label

Destroys an instance of a `Label` object.

`virtual ~Label()`

Label::GetFieldType

Retrieves the type of the current field.

`FIELDTYPE GetFieldType (FIELDTYPE * pDerived)`

Parameters `pDerived` The container to return the specific derived type of the field (in this case, `NULL_FIELD`).

Returns Type of the base class of the current field. In this case, `LABEL_FIELD`.

Description This function can be used to determine if a generic `Field` pointer (for example, one returned from `GetFieldWithFocus` in `screen.h`) is actually a `Label` field.

Label::GetText

Retrieves the text for this label.

```
char const * GetText() const
```

Label::GiveFocus

Gives the label the focus; this function can be overridden.

```
virtual bool GiveFocus(int const DirectionMagnitude)
```

Parameters `DirectionMagnitude` The number of shifts of the trackwheel. A negative number implies an upward scroll. A positive number is a downward scroll.

Returns True if the label has successfully gained focus; false otherwise.

Label::Operator =

Sets one Label object equal to another Label object.

```
Label & Operator = (const Label & src)
```

Parameters `src` An initialized Label object.

Returns A Label object that is a duplicate of `src`.

Description `Label::Operator =` sets the left side parameter to be a duplicate of `src`.

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` A pointer to the text that appears in the Label.

`length` The length of the text.

Description Form 2 ignores all `\0` characters in the string.

Label::UpdatePtr

Updates the RAM pointers associated with flash data.

Chapter 2: UI Engine API Reference

```
void UpdatePtr(char const * const pLabel)
```

Parameters	pLabel	The data associated with the edit buffer; if set to NULL, data is assumed to be the same as in a previous call and is updated accordingly.
-------------------	---------------	--

List

To include `List` functions in your application, you must include `<List.h>` in your code.

The `List` constructor has three forms:

Form 1: `List()`

Form 2: `List(int const NumEntries)`

Form 3: `List(const List & src)`

Parameters	NumEntries	The number of entries in the list.
	src	An initialized <code>List</code> object.

Form 3 creates a `List` object that is a duplicate of `src`.

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: The developer must 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.)

Functions

The following functions are listed in alphabetical order.

<code>List::~List</code>	96
<code>List::Delete</code>	96
<code>List::GetColumnFont</code>	96
<code>List::GetDisplayRange</code>	96
<code>List::GetFieldType</code>	97
<code>List::GetNumEntries</code>	97
<code>List::GetSelectedIndex</code>	97
<code>List::GetTopLineEntry</code>	97
<code>List::GetXYPint</code>	97
<code>List::Insert</code>	98
<code>List::IsRangeSelected</code>	98
<code>List::NowDisplaying</code>	98
<code>List::PutColumn</code>	98
<code>List::Redraw</code>	99
<code>List::SetColumnFont</code>	99

Chapter 2: UI Engine API Reference

List::SetEnableMultipleSelection	100
List::SetNumEntries	100
List::SetSelectedIndex	100
List::SetSelectedIndex	100
List::SetTopLineEntry	100

List::~~List

Destroys an instance of a List object.

```
virtual ~List()
```

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, is to 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.
-------------------	--------	---

Returns Returns the font object associated with the column.

List::GetDisplayRange

Sets the values of first and last to the indexes of the first element displayed on the screen and the last element displayed on the screen respectfully.

```
virtual void GetDisplayRange(int & first, int & last)
```

Parameters	first	A holder for the index of the first visible entry.
	last	A holder for the index of the last visible entry.

Description The entries between and including these indexes are visible to the user.

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

Retrieves the number of entries currently in the list.

```
int GetNumEntries() const
```

- Returns** The number of entries in the list.

List::GetSelectedIndex

Retrieves the index of the currently selected list entry.

```
int GetSelectedIndex(int * pFirst = NULL) const
```

- Parameters** `pFirst` A pointer to the list entry.
- Returns** The index of the selected item in the list.

List::GetTopLineEntry

Retrieves the specified list entry currently displayed at the top of the display.

```
int GetTopLineEntry() const
```

- Returns** The index of the first entry being displayed on the display.

List::GetXYPoint

Retrieves the XYPoint coordinate for the current list entry.

```
XYPoint GetXYPoint()
```

- Returns** The starting XYPoint position of the string about to be written.

Description An `XYPoint` is an coordinate comprised of an x-axis and a y-axis position.

List::Insert

Inserts an entry into the list.

```
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, is to be redrawn.

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

List::IsRangeSelected

Determines if multiple list elements are selected.

```
virtual bool IsRangeSelected()
```

Returns True if multiple entries are selected; false otherwise.

List::NowDisplaying

A virtual function called by the UI Engine when a particular entry in the list needs to be drawn.

```
form 1: virtual void NowDisplaying(int const index)
```

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

Description 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	<code>pnewStr</code>	A pointer to the string associated with this index.
	<code>ColumnWidth</code>	The maximum width of the location (in pixels).
	<code>Flags</code>	Reserved for future enhancement.
	<code>StrLength</code>	Reserved for future enhancement.
	<code>pBitmap</code>	A pointer to a Bitmap resource to display.
	<code>font</code>	A pointer to the font to be used for display; if NULL or omitted, the currently-selected font is used.

Description 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	<code>Index</code>	The index in the string array of item being updated.
-------------------	--------------------	--

Description Form 1 is an override of the `Field::Redraw()` function. 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 SetEnableMultipleSelection()
```

List::SetNumEntries

Sets the number of entries to 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 the list that is to be selected.
-------	---

List::SetSelectedRange

Sets a selection of multiple items for the list.

```
void SetSelectedRange(int anchorIndex, int selIndex)
```

Parameters

anchorIndex	The position of the first entry in the selection.
selIndex	The position of the last entry in the selection.

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

The Menu class is a base class for handling menus.

To include Menu functions in your application, you must include `<Menu.h>` in your code.

The Menu constructor has three forms:

Form 1: `Menu()`

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

Form 3: `Menu(const Menu & src)`

Parameters	<code>newMenuStringArray</code>	A pointer to the address of a string array, containing user-defined menu items.
	<code>NumberOfEntries</code>	The number of application-defined menu items in the <code>newMenuStringArray</code> (maximum 32).
	<code>src</code>	A reference to an initialized Menu object.

Form 3 creates a Menu object that is a duplicate of `src`.

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



Note: The user can never select menu separators and `GetSelectedIndex` never returns its offset in the `newMenuStringArray`.

Menus are composed of four 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.

Functions

The following functions are listed in alphabetical order.

<code>Menu::~Menu</code>	102
<code>Menu::ClearMenu</code>	102
<code>Menu::DisplayMenu</code>	102
<code>Menu::GetSelectedIndex</code>	103
<code>Menu::GetStartPixelPosition</code>	103
<code>Menu::GetTopIndex</code>	103
<code>Menu::HideItem</code>	103
<code>Menu::HideItems</code>	104
<code>Menu::IsHidden</code>	104
<code>Menu::IsSeparator</code>	104
<code>Menu::IsTarget</code>	105
<code>Menu::MakeSelectorVisible</code>	105
<code>Menu::MatchCharacter</code>	105
<code>Menu::MoveSelector</code>	105
<code>Menu::SetMenuItems</code>	105
<code>Menu::SetSelectedIndex</code>	106
<code>Menu::SetStartPixelPosition</code>	106
<code>Menu::SetTopIndex</code>	107
<code>Menu::ShowItem</code>	107

Menu::~~Menu

Destroys an instance of a `Menu` object.

`~Menu()`

Menu::ClearMenu

Removes the specified menu from the screen.

`void ClearMenu()`

Menu::DisplayMenu

Displays the specified menu.

`void DisplayMenu()`

Menu::GetSelectedIndex

Retrieves the index of the currently selected menu item.

```
int GetSelectedIndex() const
```

Returns The offset of the user-selected menu item.

Description The value returned is an integer, not a bitmap. For more information, refer to `Menu::SetSelectedIndex`.



Note: Menu separator item strings that begin with two underscore characters ("__") cannot be returned.

Menu::GetStartPixelPosition

Retrieves the starting pixel position for the menu.

```
int GetStartPixelPosition() const
```

Returns The starting pixel position for this menu.

Menu::GetTopIndex

Gets the offset to the item to be displayed at the top of the menu.

```
int GetTopIndex() const
```

Returns 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.

```
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 on.

Description `HideItem` is normally used to hide a single item; for multiple items, use `Menu::HideItem`. The function is provided for backwards compatibility.



Note: This function does not do anything if you already have this menu displayed (`ProcessMenu`).

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)`

Form 2: `void HideItems(const Bitmask 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; form 2 expresses it as a bitmask.
- Description** `HideItems` allows you to hide 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`. Form 2 can address more, since the `Bitmask` class can be arbitrarily large.

Menu::IsHidden

Determines if the menu item is hidden.

`bool IsHidden(int const Item) const`

- Parameters** `Item` The menu item to determine if it is hidden.
- Returns** True if the item should never be shown on the display; false otherwise.
- Description** This is used by the UI Engine to determine if the item should be drawn.

Menu::IsSeparator

Determines if the menu item is a separator.

`bool IsSeparator(int const Item) const`

- Parameters** `Item` The menu item to determine if it is a separator.
- Returns** True if the menu item is a separator; false otherwise.

Menu::IsTarget

Determines if the menu item is a target.

```
bool IsTarget(int const Item) const
```

Parameters `Item` The menu item to determine if it is a target.

Returns True if the menu item is a target; false otherwise.

Menu::MakeSelectorVisible

Displays the menu selection icon to the user.

```
void MakeSelectorVisible()
```

Menu::MatchCharacter

```
void MatchCharacter(char CharacterToMatch)
```

Parameters `CharacterToMatch` The character to match.

Menu::MoveSelector

Moves the menu selection icon.

```
void MoveSelector(int const DirectionMagnitude)
```

Parameters `DirectionMagnitude` The number of shifts of the trackwheel. A negative number implies an upward scroll. A positive number is a downward scroll.

Menu::SetMenuItems

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

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

Parameters `newMenuStringArray` A pointer to the address of a string array, containing menu items.

`NumberOfEntries` The number of menu items in the `newMenuStringArray`. In earlier releases, this was limited to 32, but that restriction is now eased by using the `Bitmask` class. The maximum number is 32.

Description 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 four 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.



Note: These can be hidden or shown individually by the associated member functions.

- 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

Shows the specified menu item as highlighted.

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

Parameters `Item` The offset in the `newMenuStringArray`.

Description For more information, refer to `Menu::GetSelectedIndex`.

Menu::SetStartPixelPosition

Sets the starting pixel position for the menu.

```
void SetStartPixelPosition(int StartPixelFromTop)
```

Parameters `StartPixelFromTop` The pixel position from which to start this menu.

Menu::SetTopIndex

Sets the top menu item.

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

Parameters `Item` The index of the first menu item to display.

Description 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

Displays the specified menu item the next time the menu is displayed.

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

Parameters `Item` The offset in `newMenuStringArray` of the item requested to be made visible.

Description This function is the opposite of the `HideItem` function.

OKDialog

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 trackwheel or presses the ENTER key on the 'OK' field.

To include OKDialog functions in your application, you must include <OKDialog.h> in your code.

The OKDialog constructor has two forms:

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.

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

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 trackwheel or presses the ENTER key on the OK field.

Functions

The following functions are listed alphabetically.

OKDialog::~~OKDialog	108
OKDialog::Go	108
OKDialog::Operator =	109
OKDialog::SetQuestion	109

OKDialog::~~OKDialog

Destroys an instance of an OKDialog object.

```
virtual ~OKDialog()
```

OKDialog::Go

Displays the dialog and clears it when the user clicks the trackwheel or presses the ENTER key.

```
virtual RESULT Go(UIEngine & UIEngineToUse)
```

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

Returns Returns `CLICKED` when the user clicks the trackwheel or presses the `ENTER` key.

OKDialog::Operator =

Sets one `OKDialog` object equal to another `OKDialog` object.

`OKDialog & Operator = (const OKDialog & src)`

Parameters `src` An initialized `OKDialog` object.

Returns A `OKDialog` object that is a duplicate of `src`.

Description `OKDialog::Operator =` sets the left side parameter to be a duplicate of `src`.

OKDialog::SetQuestion

Sets the text to be displayed in the dialog box.

`bool SetQuestion(char const * const pStatement)`

Parameters `pStatement` The statement to be displayed in the dialog box.

Returns True if the statement was set successfully; false otherwise.

Description The statement will be displayed in a dialog box with an information bitmap and a field containing the string 'OK'.

RichText

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.

To include `BusyStatus` functions in your application, you must include `<RichText.h>` in your code.

Use the following constructor to create a `RichText` object:

```
RichText()
```

Structures

The following structures are declared in `RichText.h`.

```
RichText::TextRangeAttribute ..... 110
RichText::WrapPoint ..... 110
```

RichText::TextRangeAttribute

A structure containing text range attribute information.

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

Field	Description
Offset	The offset into the text buffer indicating the start of the text range.
Length	The length of the affected text.
Attribute	A font index indicating the font used for the text range; a number in the range of 0 to 4.

RichText::WrapPoint

A structure containing text wrapping attribute information.

```
struct WrapPoint
{
    int LineNumber;
    int OffsetToLine;
    int LengthOfLine;
```

```
};
```

Field	Description
LineNumber	The line number of the affected text.
OffsetToLine	The wrapping point in the line in which the text should be broken.
LengthOfLine	The length of the affected line.

Functions

The following functions are listed alphabetically.

RichText::~RichText	111
RichText::FindOffset	111
RichText::FindWidth	112
RichText::GetAttributes	112
RichText::GetBuffer	112
RichText::GetBufferLength	112
RichText::GetCursorOffset	113
RichText::GetFieldType	113
RichText::GetWrapPoint	113
RichText::SetAttributes	113
RichText::SetBuffer	113
RichText::SetBufferAppended	114
RichText::SetCursorOffset	114
RichText::SetFonts	114
RichText::SetLabel	114
RichText::UpdatePtr	115

RichText::~RichText

Destroys an instance of a RichText object.

```
virtual ~RichText()
```

RichText::FindOffset

Finds the offset, in characters, from the given location to the end of the line or until the specified width is reached.

```
int FindOffset(
    int offset,
    int width) const
```

Parameters

offset	The offset into the text buffer.
width	The maximum pixel width to consider.

Returns The offset in characters until the end of line or the given width, whichever is smaller.

RichText::FindWidth

Retrieves the width of the specified text.

```
short FindWidth(int offset, int length) const
```

Parameters

offset	The offset into the text buffer.
length	The number of characters to consider.

Returns The width of the text string on the screen.

RichText::GetAttributes

Retrieves attributes of the RichText field.

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

Parameters

AttributeArray	The 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
```

Returns The rich text string as pure text.

RichText::GetBufferLength

Retrieves the buffer length associated with the RichText component buffer.

```
int GetBufferLength() const
```

Returns The length of the RichText buffer.

RichText::GetCursorOffset

Retrieves the current cursor offset.

```
int GetCursorOffset() const
```

Returns The offset of the cursor inside the RichText buffer.

RichText::GetFieldType

Returns the type of the current field.

```
FIELDTYPE GetFieldType (FIELDTYPE * pDerived)
```

Parameters `pDerived` The container to return the specific derived type of the field.

Returns Type of the base class of the current field. In this case, RICHTEXT_FIELD.

Description This function can be used to determine if a generic Field pointer (for example, one returned from GetFieldWithFocus in screen.h) is actually a RichText Field.

RichText::GetWrapPoint

Retrieves the wrap point for a line of text.

```
WrapPoint GetWrapPoint () const
```

Returns The wrap point for the line of RichText.

Description The wrap point is the position in the text where the line is broken and the text is continued over to the next line.

RichText::SetAttributes

Sets attributes for a rich text component.

```
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.

```
void SetBuffer(
    char const * buffer,
    int length)
```

Parameters	buffer	The new buffer to set for the RichText component.
	length	The length of the buffer.

RichText::SetBufferAppended

Sets the buffer for the rich text component.

```
void SetBufferAppended(  
    char const * buffer,  
    int length);
```

Parameters	buffer	A pointer to the character string to set for the buffer.
	length	The length of the string pointed to by the buffer.



Warning: SetBufferAppended() does not currently append text to the end of the buffer; it overwrites the existing buffer.

RichText::SetCursorOffset

Sets the cursor offset into the RichText buffer.

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

Parameters	NewCharacterOffset	The offset into the buffer.
-------------------	---------------------------	-----------------------------

RichText::SetFonts

Sets the font array to be used when setting up text attributes.

```
void SetFonts (  
    Font const * const * fonts,  
    int nFonts)
```

Parameters	fonts	The font array.
	nFonts	The number of fonts in the array.

Description The font index in RichText::TextRangeAttribute is an offset into this font array. For more information, refer to RichText::TextRangeAttribute.

RichText::SetLabel

Sets the label for a RichText field.

Form 1: `void SetLabel(char const * label)`

Form 2: `void SetLabel(Field * label)`

Parameters `label` The label for the RichText field.

RichText::UpdatePtr

Updates RAM pointers associated with flash data.

```
virtual void UpdatePtr (
    char const * pNewBufferLoc,
    char const * pNewLabelLoc = 0,
    TextRangeAttribute const * AttributeArray = 0)
```

Parameters

<code>pNewBufferLoc</code>	The new location of the text buffer.
<code>pNewLabelLoc</code>	The new location of the label.
<code>AttributeArray</code>	The new location of the text attributes.

Description If any of the parameters are set to NULL, they are assumed to be unchanged.

Screen

The screen class inherits from the `FieldManager` class.

To include Screen functions in your application, you must include `<Screen.h>` in your code.

The Screen constructor has three forms:

Form 1: `Screen()`

Form 2: `Screen(UIEngine & uiEngine)`

Parameters	<code>uiEngine</code>	A reference to a <code>UIEngine</code> component to use for rendering the screen.
-------------------	-----------------------	---

This constructor instantiates a `Screen` object. A `Screen` object is the container of the field objects and a `title` object.

Functions

The following functions are listed alphabetically.

<code>Screen::~Screen</code>	117
<code>Screen::AddField</code>	117
<code>Screen::AddLabel</code>	117
<code>Screen::ClearScreen</code>	117
<code>Screen::Close</code>	117
<code>Screen::DidSave</code>	118
<code>Screen::Draw</code>	118
<code>Screen::GetUIEngine</code>	118
<code>Screen::Invalidate</code>	118
<code>Screen::IsDisabled</code>	118
<code>Screen::IsEmptyOfLabels</code>	119
<code>Screen::IsFieldDirty</code>	119
<code>Screen::MarkFieldAsClean</code>	119
<code>Screen::MarkFieldAsDirty</code>	119
<code>Screen::OnChangeOptions</code>	119
<code>Screen::OnClose</code>	120
<code>Screen::OnFieldUpdate</code>	120
<code>Screen::OnKey</code>	120
<code>Screen::OnMenu</code>	120
<code>Screen::OnMenuItem</code>	121
<code>Screen::OnMessage</code>	121
<code>Screen::OnSave</code>	121
<code>Screen::OnRoll</code>	122
<code>Screen::Process</code>	122
<code>Screen::PutFocusAtBottom</code>	122
<code>Screen::RemoveAllLabels</code>	122

Screen::RemoveLabel	123
Screen::ResetDimensionDependentData	123
Screen::ResetScreen	123
Screen::Save	123
Screen::SetIntegralHeight	123
Screen::SetMenu	124
Screen::SetScreenDisabled	124

Screen::~Screen

Destroys an instance of a Screen object.

```
virtual ~Screen()
```

Screen::AddField

Adds a Field object to the screen.

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

Parameters	newField	A Field object (list, choice box, edit box, or table).
	pFieldAfterNewField	A pointer to the existing field that will follow the new field, or NULL (default value); if NULL, the new field is added to the end of the associated screen's list of fields.

Description This function is inherited from and overridden by FieldManager.

Screen::AddLabel

Adds a title to the top of the screen.

```
void AddLabel(Label & newTitle)
```

Parameters newTitle A Title object.

Description For more information on Title objects, refer to "Title" on page 152.

Screen::ClearScreen

Removes all fields from the screen component.

```
void ClearScreen()
```

Screen::Close

Exits the Process() message loop.

```
void Close()
```

Screen::DidSave

Determines if a user has requested a save.

```
bool DidSave()
```

Returns True if the user has requested a save for the screen component; false otherwise.

Description If Save is not overridden, then this function can be used to determine if users have requested a save on the screen component.

Screen::Draw

Draws the screen.

```
Form 1: virtual void Draw()
```

```
Form 2: virtual void Draw(Field * field)
```

Parameters `field` The field to draw.

Description Form 1 draws the entire screen. Form 2 draws a specific field.

Screen::GetUIEngine

Gets the UIEngine associated with the screen.

```
UIEngine & GetUIEngine()
```

Returns Returns a reference to the UIEngine component associated with the screen.

Screen::Invalidate

Invalidates the entire screen (causes repaint).

```
void Invalidate()
```

Description If you're using Process as the message loop, calling Invalidate causes a repaint on the screen component.

Screen::IsDisabled

Determines if the screen is disabled.

```
bool IsDisabled()
```

Returns Returns true if the screen is disabled; false otherwise.

Screen::IsEmptyOfLabels

Determines if there are no label objects associated with the screen.

```
bool IsEmptyOfLabels()
```

Returns True if no labels are associated with this screen; false otherwise.

Description Returns true if no fields have been added using `AddField` or if they have all been subsequently removed using `RemoveField`.

Screen::IsFieldDirty

Determines if any field on the screen is marked as dirty.

```
bool IsFieldDirty()
```

Returns True if any of the fields associated with the screen have changed; false otherwise.

Description 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()
```

Description 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()
```

Description 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.

```
virtual bool OnChangeOptions()
```

Returns True if the Change Option was successful; false otherwise.

Description This function is called when the field in focus is a choice field and the user selects the "change options" menu item. This function can be overridden.

Screen::OnClose

Called when the user exits the screen.

```
virtual void OnClose()
```

Description The default implementation of this function exits the screen if `IsFieldDirty()` returns false. Otherwise, a default dialog is displayed, indicating that screen component data has changed. Override this function to display a custom dialog.

Screen::OnFieldUpdate

Sends notification that a field's data has changed.

```
virtual void OnFieldUpdate(Field & field)
```

Parameters `field` A reference to the field for which data has changed.

Description If you're using `Process()` for the message loop, your code should override this function to provide functionality. This function can be overridden.

Screen::OnKey

Handles key input for the screen.

```
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.

Description Override this function to provide input handling at the screen level. This function is only called if the field in focus does not accept the input. This function can be overridden.

Screen::OnMenu

Called when the user requests a menu; can override first form.

```
Form 1: virtual int OnMenu(  
    int selection,  
    int fDisable)
```

```
Form 2: int OnMenu(  
    int selection,
```


Bitmask fDisable)

Parameters	selection	The default selection as set through a call to <code>SetMenu(...)</code> .
	fDisable	A bitfield indicating items to hide. Also set through a call to <code>SetMenu(...)</code> . An 'on' bit in the bitfield represents a menu item to hide. Up to 32 items can be disabled in this way. In the integer version (form 1), up to 32 items can be disabled. In the Bitmask version (form 2), any number of items can be disabled.

Returns The index of the item selected; -1 if the menu is hidden (that is, on `HIDE_MENU`).

Description Developers can override this function and do preprocessing (hiding items, adding items) prior to calling the default implementation of either form.

Screen::OnMenuItem

Called when a menu item has been selected.

```
virtual void OnMenuItem(int item)
```

Parameters **item** The menu item selected.

Description 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. This function can be overridden.

Screen::OnMessage

Called when a message has been received.

```
virtual void OnMessage(const MESSAGE & message)
```

Parameters **message** The RIM message structure populated with newly received message information.

Description This function 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. This function can be overridden.

Screen::OnSave

Checks if the screen has changed and calls `Save()`.

```
virtual void OnSave()
```

Description This function 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()` function is subsequently called. This function can be overridden.

Screen::OnRoll

Called when the trackwheel has moved.

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

Parameters	<code>directionMagnitude</code>	A value indicating the magnitude of the roll. Positive values indicate a roll down. A value of 1 causes the screen to display one new field or one new line of text.
	<code>flags</code>	Key modifiers such as <code>ALT_STATUS</code> or <code>SHIFT_STATUS</code> , etc.

Description Normally this function 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. This function can be overridden.

Screen::Process

Activates the screen and provides a message loop.

```
int Process()
```

Returns If the menu is activated during the process call, the return value is the item selected, otherwise the return is -1.

Screen::PutFocusAtBottom

Sets the focus to the bottom of the current field

```
void PutFocusAtBottom()
```

Description Sets the focus to the bottom of the currently selected field or does nothing if no field is currently selected.

Screen::RemoveAllLabels

Removes all titles associated with the screen

```
void RemoveAllLabels()
```

Description Removes every label object associated with the screen, using `AddLabel()`. (The `RemoveLabel()` call removes only the title.) For more information, refer to `Screen::AddLabel`, and `Screen::RemoveLabel`.

Screen::RemoveLabel

Removes a title from the screen.

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

Parameters TitleToRemove A title object.

Description This function removes the Title object from the list of titles in the associated screen. For more information, refer to Screen::AddLabel, and Screen::RemoveAllLabels.

Screen::ResetDimensionDependentData

Calls the recalculation methods of all fields associated with this screen.

```
virtual void ResetDimensionDependentData()
```

Description Call this function when system display parameters have changed—for instance, when system fonts have been changed from small to large.

Screen::ResetScreen

Removes all titles and fields from the screen.

```
void ResetScreen()
```

Description 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.

```
virtual void Save()
```

Description You should override this function and save to flash any data that is to persist. This function is called by OnSave() if fields have changed and need to be saved. This function can be overridden.

This function is inherited and overridden from FieldManager.

Screen::SetIntegralHeight

Sets whether the screen is an integral number of font heights high.

```
void SetIntegralHeight(bool isIntegralHeight = true)
```

Parameters isIntegralHeight If set to true, tells engine to make the screen an integral number of font heights high.

Screen::SetMenu

Sets the menu for use with the default menu handling.

```
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 <i>verb object</i> , 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(...) function as the selection parameter.
	fMenuMask	A 32 bit bitmask indicating which items are to be disabled. An ‘on’ bit indicated that the corresponding item at that index should be disabled.

Screen::SetScreenDisabled

Disables the screen.

```
void SetScreenDisabled()
```

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

Separator

A separator field is used to separate two fields for readability purposes. If no parameters are passed to it, the string that is 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.

To include Separator functions in your application, you must include `<Separator.h>` in your code.

The Separator constructor has three forms:

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)`

Form 3: `Separator (const Separator & src)`

Parameters	<code>pnewSeparatorString</code>	A pointer to the label associated with the separator; if this is <code>NULL</code> , the Separator object will be displayed as a black line across the width of the screen.
	<code>newSeparatorStringLength</code>	The length associated with the separator.
	<code>newJustification</code>	The justification associated with the separator.
	<code>pnewSeparatorBitMap</code>	A pointer to bitmap associated with the separator.
	<code>newXOffset</code>	The horizontal offset of the bitmap.
	<code>FontHeight</code>	If set to <code>true</code> , the Separator object will have the same height as a single line of text. If set to <code>false</code> , the Separator object will be shorter.
	<code>src</code>	A reference to an initialized Separator object.

Form 3 creates a Separator object that is a duplicate of `src`.

Functions

The following functions are listed alphabetically.

Separator::~Separator	126
Separator::GetFieldType	126
Separator::Operator =	126
Separator::SetText	126
Separator::UseFontHeight	127

Separator::~Separator

Destroys an instance of a Separator object.

`~Separator()`

Separator::GetFieldType

Returns the type of the current field.

`FIELDTYPE GetFieldType(FIELDTYPE * pDerived = NULL)`

Parameters	<code>pDerived</code>	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.
Returns		Returns the type of the base class of the current field (that is, the type of the first class derived from the Field object)
Description		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::Operator =

Sets one Separator object equal to another Separator object.

`Separator & Operator = (const Separator & src)`

Parameters	<code>src</code>	An initialized Separator object.
Returns		A Separator object that is a duplicate of src.
Description		<code>Separator::Operator =</code> sets the left side parameter to be a duplicate of src.

Separator::SetText

Sets the text to be displayed by the Separator object.

```
void SetText(char const * const pnewText)
```

Parameters pnewText The text to be displayed.

Description If pnewText is NULL, the Separator object will be displayed as a black line across the width of the screen.

Separator::UseFontHeight

Bases separator dimensions on the current font size.

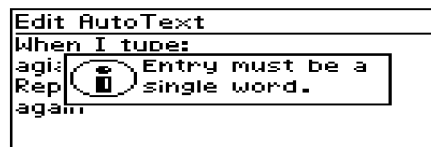
```
virtual void UseFontHeight(bool FontHeightFlag)
```

Parameters FontHeightFlag The current font height.

Status

The status box displays system modal informational messages for a predefined 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

To include Status functions in your application, you must include <Status.h> in your code.

The Status constructor has six forms:

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	A pointer to a bitmap.
	pdb	One of the predefined bitmap shapes specified in <code>PredefinedBitmaps.h</code> : <ul style="list-style-type: none"> • <code>Bitmaps::INFORMATION</code> • <code>Bitmaps::QUESTION</code> • <code>Bitmaps::EXCLAMATION</code> • <code>Bitmaps::HOURLASS</code>
	DisplayTime	The amount of time (in 1/100 second intervals) the status is displayed.

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.

Functions

The following functions are listed alphabetically.

<code>Status::~~Status</code>	129
<code>Status::ClearModalStatus</code>	130
<code>Status::DisplayModalStatus</code>	130
<code>Status::DrawStatus</code>	130
<code>Status::GetDisplayTime</code>	130
<code>Status::Operator =</code>	130
<code>Status::PostAckStatus</code>	131
<code>Status::SetBitmap</code>	131
<code>Status::SetDisplayTime</code>	131
<code>Status::SetText</code>	132
<code>Status::ShowStatus</code>	132

Status::~~Status

Destroys an instance of a `Status` object.

`~Status()`

Status::ClearModalStatus

Removes the current status box from the display.

```
void ClearModalStatus()
```

Description 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.

```
void DisplayModalStatus()
```

Description 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::DrawStatus

Draws the status box.

```
void DrawStatus()
```

Description Draws the status box centered at the top of the current screen with the proper dimensions and with line breaks inserted into the text.

The status box will not be automatically cleared from the screen.

Status::GetDisplayTime

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

```
int GetDisplayTime() const
```

Returns The amount of time (in 1/100 second intervals) that the status will be displayed.

Description 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::Operator =

Sets one Status object equal to another Status object.

```
Status & Operator = (const Status & src)
```

Parameters `src` An initialized Status object.

Returns A Status object that is a duplicate of `src`.

Status::PostAckStatus

Draws a status box that requires user acknowledgement.

```
static void Status::PostAckStatus(const char * pErrStr, int nErrStrLen)
```

Parameters `pErrStr` The string to display in the status box.

`nErrStrLen` The length of the string to display.

Description `PostAckStatus` is useful when displaying a message that you want to ensure the user reads, such as an error message.

Status::SetBitmap

Sets the bitmap that will appear in the status box.

```
Form 1: void SetBitmap(Bitmap const * const pBitmap)
```

```
Form 2: void SetBitmap(Bitmaps::PREDEFINED_BITMAP const pdb)
```

Parameters `pBitmap` A pointer to a bitmap.

`pdb` One of the predefined bitmap shapes specified in `PredefinedBitmaps.h`:

- `Bitmaps::INFORMATION`
- `Bitmaps::QUESTION`
- `Bitmaps::EXCLAMATION`
- `Bitmaps::HOURLASS`

Description `SetBitmap` 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.

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

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

Description SetDisplayTime 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 be automatically removed after the specified period of time.

Status::SetText

Sets the text that will appear in the status box.

```
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.

Description SetText is used if the default constructor was invoked.

Status::ShowStatus

Displays a status box.

```
Form 1: static inline void ShowStatus(  
    Bitmaps::PREDEFINED_BITMAP const bitmap,  
    int time,  
    const char * const pszText)
```

```
Form 2: static inline void ShowStatus(int time,  
    const char * const pszText)
```

```
Form 3: static inline void ShowStatus(const char * const pszText)
```

Parameters

bitmap	One of the predefined bitmap shapes specified in PredefinedBitmaps.h: <ul style="list-style-type: none">• Bitmaps::INFORMATION• Bitmaps::QUESTION• Bitmaps::EXCLAMATION• Bitmaps::HOURLASS
time	The length of time that the status box is to remain on the screen.
pszText	The text to display in the status box.

StringList

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.

To include StringList functions in your application, you must include <List.h> in your code.

The StringList constructor has three forms:

Form 1: StringList()

Form 2: StringList(char const * const * StringArray)

Form 3: StringList(const StringList & src)

Parameters	StringArray	A pointer to an array of strings. The last entry must be 0.
	src	A reference to an initialized StringList object.

Form 3 of the constructor creates a new StringList object that is a duplicate of src.

Functions

The following functions are listed alphabetically.

StringList::~StringList	133
StringList::GetFieldType	133
StringList::Operator =	134
StringList::SetStringList	134

StringList::~StringList

Destroys an instance of a StringList object.

```
virtual ~StringList()
```

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.

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

Description For a `StringList` object, the function will return `LIST_FIELD`, and `pDerived` will be filled in with `STRING_LIST_FIELD`.

StringList::Operator =

Sets one `StringList` object equal to another `StringList` object.

`StringList & Operator = (const StringList & src)`

Parameters `src` An initialized `StringList` object.

Returns A `StringList` object that is a duplicate of `src`.

Description `StringList::Operator =` sets the left side parameter to be a duplicate of `src`.

StringList::SetStringList

Sets the strings that will be shown in the list .

Form 1: `void SetStringList(char const * const * newStringList)`

Form 2: `void SetStringList(
 char const * const * newStringList,
 int nEntries)`

Parameters `newStringList` 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

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

To include `Table` functions in your application, you must include `<Table.h>` in your code.

The `Table` constructor has three forms:

```
Form 1: Table()
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)
Form 4: Table(const Table & src)
```

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 <code>SetDrawFlags()</code> ; alternatively, each call to <code>PutCell()</code> can specify a border style for an individual cell.
	ShowCaptions	If true, the caption for the current cell will be displayed beneath the table.
	src	A reference to an initialized <code>Table</code> object.

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()`.

Functions

The following functions are listed alphabetically.

Table::~Table	136
Table::AllowBorders	137

Chapter 2: UI Engine API Reference

Table::DeleteColumnTable:	137
Table::DeleteRowTable:	137
Table::FindNextCell	138
Table::GetArea	138
Table::GetCaption	139
Table::GetColumnWidth	139
Table::GetFieldType	139
Table::GetLeftRightIndent	139
Table::GetNumCells	140
Table::GetNumColumns	140
Table::GetNumRows	140
Table::GetRowHeight	140
Table::GetSelectedCell	140
Table::GetSelectedColumn	141
Table::GetSelectedRow	141
Table::GetTopCell	141
Table::GetTopColumn	141
Table::GetTopRow	141
Table::InsertColumn	142
Table::InsertRow	142
Table::IsCellSelectable	142
Table::MoveHighlight	143
Table::NowDisplaying	143
Table::OnChangedFocus	144
Table::Operator =	144
Table::PutCell	144
Table::PutCell	145
Table::Redraw	146
Table::SelectingCell	146
Table::SetColumnWidth	147
Table::SetDrawFlags	147
Table::SetHighlight	147
Table::SetJustification	148
Table::SetProperties	148
Table::SetRowHeight	149
Table::SetSelectedCell	149
Table::SetTableSize	149
Table::SetTableStructure	149
Table::SetTopCell	150
Table::ShowCaptions	150
Table::UpdateCaption	150

Table::~Table

Destroys an instance of a Table object.

```
virtual ~Table()
```


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	True if the UI Engine should leave space for a border; false otherwise.
Description	The border style for every cell in the table can be specified by calling <code>SetDrawFlag()</code> ; alternatively, each call to <code>PutCell()</code> can specify a border style for an individual cell.	

Table::DeleteColumnTable:

Removes a column from a table

```
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.
Description	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.

```
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.

Description 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.

```
virtual 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.

Description 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::GetArea

Retrieves an Area covering the cell at Column and Row.

```
Area GetArea(  
    int const Column,  
    int const Row) const
```

Parameters	Column	The column of the cell for which to return the area.
	Row	The row of the cell for which to return the area.

Returns The area covering the requested cell.

Description This could be used to make further graphical modifications to a certain cell. GetArea() throws an exception if you do not first add the appropriate field to the manager.

Table::GetCaption

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

```
virtual 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. |
- Returns** The caption for the specified cell to displayed beneath the table.
- Description** This virtual function will be called each time a cell is selected.

Table::GetColumnWidth

Retrieves the width of a column in the table.

```
int GetColumnWidth()const
```

- Returns** The width of a column.
- Description** All columns in the table are the same width.

Table::GetFieldType

Retrieves 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. |
|----------|--|
- Returns** The type of the base class of the current field (i.e. the type of the first class derived from the Field object)
- Description** 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::GetLeftRightIndent

Retrieves the indentation of text from the left/right sides of the cell.

```
virtual int GetLeftRightIndent()
```

- Returns** The default implementation returns 0.

Description This can be overloaded to force the text to remain a certain distance from the edges of the cells.

Table::GetNumCells

Retrieves the number of cells in the table

```
void GetNumCells(  
    int * oNumColumns,  
    int * oNumRows)
```

Parameters	oNumColumn	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

Retrieves the number of columns in the table.

```
int GetNumColumns() const
```

Returns Number of columns in the table.

Table::GetNumRows

Retrieves the number of rows in the table.

```
int GetNumRows() const
```

Returns Number of rows in the table.

Table::GetRowHeight

Retrieves the height of a row in the table.

```
int GetRowHeight() const;
```

Returns The height of a row in pixels.

Description All rows in the table are the same height.

Table::GetSelectedCell

Retrieves 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

Retrieves the index of the selected column.

```
int GetSelectedColumn() const
```

Returns One-based index of the selected column.

Table::GetSelectedRow

Retrieves the index of the selected row.

```
int GetSelectedRow() const
```

Returns One-based index of the selected row.

Table::GetTopCell

Retrieves the row and column of the cell in the top left corner of the display.

```
void GetTopCell(
    int * Column,
    int * Row)
```

Parameters	Column	A pointer to an integer variable that will receive the column of the top left cell.
	Row	A pointer to an integer variable that will receive the row of the top left cell.

Table::GetTopColumn

Retrieves the index of the column at the left side of the display.

```
int GetTopColumn() const
```

Returns One-based index of the column at the left of the display.

Table::GetTopRow

Retrieves the index of the row at the top of the display.

```
int GetTopRow() const
```

Returns One-based index of the row at the top of the display.

Table::InsertColumn

Inserts a column into a table.

```
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.

Description 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.

```
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.

Description 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 if a cell in a table is selectable (can be highlighted).

```
virtual 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.
Returns	True if the specified cell is selectable; false otherwise.	
Description	By default, every cell in the table is selectable. This is a virtual function that enables you to specify which cells can receive a highlight and which cannot. The default implementation of <code>FindNextCell</code> skips over cells that return false from <code>IsCellSelectable</code> .	

Table::MoveHighlight

Highlights a cell or changes which cell is highlighted.

```
void MoveHighlight(
    const int nNewColOnScreen,
    const int nNewRowOnScreen,
    const int nOldColOnScreen = -1,
    const int nOldRowOnScreen = -1)
```

Parameters	nNewColOnScreen	The column of the cell to highlight.
	nNewRowOnScreen	The row of the cell to highlight.
	nOldColOnScreen	The column of the cell that was previously highlighted. -1 assumes no cell was selected.
	nOldRowOnScreen	The row of the cell that was previously highlighted. -1 assumes no cell was selected.

Table::NowDisplaying

A virtual member function that allows you to dynamically choose the information to be displayed in each cell.

```
form1 : virtual void NowDisplaying(
    int const Column,
    int const Row) = 0
```

Parameters	Column	The column of the cell being displayed.
	Row	The row of the cell being displayed.
Description	This virtual function is called each time a cell is displayed on the screen. In response to this function being called, you should call <code>PutCell</code> to display the correct data.	

Table::OnChangedFocus

Performs an action whenever a cell gains focus.

```
virtual void OnChangedFocus(  
    int const column,  
    int const row)
```

Parameters

column	The column of the cell gaining focus.
row	The row of the cell gaining focus.

Description The default implementation does nothing. This function can be overridden.

Table::Operator =

Sets one Table object equal to another Table object.

```
Table & Operator = (const Table & src)
```

Parameters

src	An initialized Table object.
-----	------------------------------

Returns A Table object that is a duplicate of src.

Description Table::Operator = sets the left side parameter to be a duplicate of src.

Table::PutCell

Called from the NowDisplaying function to show text inside a cell that is being drawn on the screen.

```
void PutCell(  
    char const * const Str,  
    int const ColumnWidth = -1,  
    int const justifyFlags = JF_DEFAULT,  
    int const StrLength = INT_MAX,  
    int const BorderFlags = DF_DEFAULT)
```


Parameters	Str	The text 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, 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 in the cell vertically left, center or right justified, respectively.
	StrLength	The maximum number of characters to be copied from Str. The default value displays 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.
Description	PutCell can be called multiple times for the same cell; the specified text is placed at the end of the last specified column width, and extends 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.

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

Parameters	BitmapPtr	A pointer to a bitmap structure to be displayed in the cell.
-------------------	------------------	--

XPos	The x-coordinate of the position where the bitmap is to be displayed, relative to the left edge of the cell.
YPos	The y-coordinate of the position where the bitmap is to 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.

Description PutCell can be called multiple times for the same cell. The specified bitmap is placed at the specified position each time. If the bitmap does not fit inside the cell, it is 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: virtual void Redraw()

Form 2: virtual 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.

Description 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() is 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.

virtual void SelectingCell(
 int const Column,
 int const Row)

Parameters	Column	The column of the cell that is about the be selected.
	Row	The row of the cell that is about the be selected.
Description	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.

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

Parameters	ColumnWidth	The width of each column.
Description	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.

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

Parameters	DrawFlags	A combination of the flags DF_LEFT, DF_RIGHT, DF_TOP, DF_BOTTOM, and DF_OUTLINE.
Description	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.

```
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.
Description	HF_HIGHLIGHT_NONE does not highlight the selected cell in any way. HF_HIGHLIGHT_OUTLINE highlights the selected cell by drawing an outline around the edge of the cell. HF_HIGHLIGHT_CELL highlights the selected cell by inverting the entire	

area of the cell. `HF_HIGHLIGHT_CORNERS` highlights the selected cell by placing a dot in each corner. `HF_HIGHLIGHT_ELBOW` highlights the selected cell by drawing L-shaped brackets in each corner.

Table::SetJustification

Sets the justification mode for the entire table.

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

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

Parameters	<code>justify</code>	A combination of the flags <code>JF_HOR_LEFT</code> , <code>JF_HOR_CENTERED</code> , <code>JF_HOR_RIGHT</code> , <code>JF_VER_TOP</code> , <code>JF_VER_CENTERED</code> , <code>JF_VER_LEFT</code> , <code>JF_VER_RIGHT</code> and <code>JF_VER_BOTTOM</code> .
	<code>justification</code>	One of: <code>JUST_CENTER</code> , <code>JUST_LEFT</code> , <code>JUST_RIGHT</code> , <code>JUST_FULL</code> .
Description	<p>These flags control the justification of the text in every cell in the table. At most, one of <code>JF_HOR_LEFT</code>, <code>JF_HOR_CENTERED</code>, and <code>JF_HOR_RIGHT</code> should be specified to display the text horizontally left, center, or right justified, respectively. At most, one of <code>JF_VER_LEFT</code>, <code>JF_VER_CENTERED</code>, or <code>JF_VER_RIGHT</code> 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 <code>PutCell()</code>.</p> <p>Form 1 is deprecated; developers should use Form 2.</p>	

Table::SetProperties

Sets the properties for the entire table.

`void SetProperties(int const Properties)`

Parameters	<code>Properties</code>	A combination of the flags <code>PF_TOP_BOTTOM_CHECK</code> , <code>PF_LEFT_RIGHT_CHECK</code> , <code>PF_ALIGN_CENTER</code> , and <code>PF_ALIGN_RIGHT</code> .
Description	<p>If <code>PF_TOP_BOTTOM_CHECK</code> is not specified and the user scrolls down past the bottom of the table, the cursor appears at the top of the next column in the table. If the user scrolls up past the top of the table the cursor appears at the bottom of the previous column. If <code>PF_TOP_BOTTOM_CHECK</code> is specified, the cursor stops at the top and bottom of the table.</p> <p>If <code>PF_LEFT_RIGHT_CHECK</code> is not specified and the user scrolls past the right edge of the table, the cursor appears at the left side of the next row. If the user scrolls left past the left edge of the table, the cursor appears at the right side of the previous row.</p> <p>Only one of <code>PF_ALIGN_CENTER</code> and <code>PF_ALIGN_RIGHT</code> should be specified. They specify that the entire table should be displayed in the center or against the right side of the screen, respectively.</p>	

Table::SetRowHeight

Sets the height of each row in the table.

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

Parameters RowHeight The height of each row.

Description 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: void SetSelectedCell(
        int const Column,
        int const Row)
```

```
Form 2: void SetSelectedCell(int Offset)
```

Parameters Column The column of the cell to select.

Row The row of the cell to select.

Offset The offset of the cell to select.

Description 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.

```
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.

```
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.

Description 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 is to be placed in the top left corner of the display.

```
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.

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

Parameters	ShowCaptions	True if the UI Engine should leave space for a caption; false otherwise.
-------------------	--------------	--

Description The caption can be associated dynamically with each cell. In order to display the caption for a particular cell, the UI Engine calls 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 function called when the user has moved focus or when a paint function is called.

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

Parameters	<code>column</code>	The new column with focus.
	<code>row</code>	The new row with focus.
Description	Use this function to display extra information about the cell that the user has censored over.	

Title

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

To include `Title` functions in your application, you must include `<Label.h>` in your code.

```
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)
```

```
Form 3: Title(const Title & src)
```

Parameters	<code>pnewText</code>	The position where text starts (in pixels).
	<code>XPosition</code> <code>Xposition</code>	The position where text starts (in pixels) from top-left corner.
	<code>Width</code>	The width of the text (in pixels).
	<code>justify</code>	The justification of the text. One of <code>JUST_CENTER</code> , <code>JUST_LEFT</code> , <code>JUST_RIGHT</code> , <code>JUST_FULL</code> .
	<code>src</code>	A reference to an initialized <code>Title</code> object.

Form 1 ignores the justify settings. Form 3 creates a `Title` object that is a duplicate of `src`.

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 average width of system font characters on the RIM Wireless Handheld is 5 pixels.

Functions

The following functions are listed in alphabetical order.

<code>Title::~Title</code>	153
<code>Title::Operator =</code>	153
<code>Title::SetLocation</code>	153
<code>Title::SetText</code>	153

Title::~~Title

Destroys an instance of a Title object.

```
virtual ~Title()
```

Title::Operator =

Sets one Title object equal to another Title object.

```
Title & Operator = (const Title & src)
```

Parameters `src` An initialized Title object.

Returns A Title object that is a duplicate of `src`.

Description `Title::Operator =` sets the left side parameter to be a duplicate of `src`.

Title::SetLocation

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

```
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).

Description 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: `void SetText(char const * const pnewText)`

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

Chapter 2: UI Engine API Reference

Parameters	<code>pnewText</code>	A pointer to the text that appears in the title.
	<code>text</code>	A pointer to the text that appears in the title.
	<code>length</code>	The length of the text.
Description	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 <code>SetLocation</code> member function. Form 2 ignores all <code>\0</code> characters in the string.	

TopBottomChoice

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

To include TopBottomChoice functions in your application, you must include <Choice.h> in your code.

The TopBottomChoice constructor has two forms:

Form 1: TopBottomChoice(char * Label, bool CurrentChoice)

Form 2: TopBottomChoice(const TopBottomChoice & src)

Parameters	Label	A 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	A initialized TopBottomChoice object.

Form 1 is a standard constructor. Form 2 creates a TopBottomChoice object that is a duplicate of src.

Functions

The following functions are listed alphabetically.

TopBottomChoice::~TopBottomChoice	155
TopBottomChoice::GetFieldType	155
TopBottomChoice::GetFlag	156
TopBottomChoice::Operator =	156
TopBottomChoice::SetFlag	156

TopBottomChoice::TopBottomChoice

Constructor for the TopBottomChoice class.

TopBottomChoice::~~TopBottomChoice

Destroys an instance of a TopBottomChoice object.

virtual ~TopBottomChoice()

TopBottomChoice::GetFieldType

Retrieves 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.
Returns	The type of the base class of the current field (that is, the type of the first class derived from the Field object).	
Description	For a TopBottomChoice object, GetFieldType returns CHOICE_FIELD, and pDerived is completed with TOP_BOTTOM_CHOICE_FIELD.	

TopBottomChoice::GetFlag

Retrieves the value displayed in the Top/Bottom Choice field.

```
bool GetFlag()
```

Returns True if Top; false if Bottom.

TopBottomChoice::Operator =

Sets one TopBottomChoice object equal to another TopBottomChoice object.

```
TopBottomChoice & Operator = (const TopBottomChoice & src)
```

Parameters	src	An initialized TopBottomChoice object.
Returns	A TopBottomChoice object that is a duplicate of src.	
Description	TopBottomChoice::Operator = sets the left side parameter to be a duplicate of src.	

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

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.

To include UIEngine functions in your application, you must include `<UIEngine.h>` in your code.

Use the following constructor to create a UIEngine object:

```
UIEngine()
```

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 displays a screen (`ProcessScreen()`) then displays 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.

Functions

The following functions are listed alphabetically.

UIEngine::~UIEngine	158
UIEngine::ClearDialog	158
UIEngine::ClearMenu	158
UIEngine::ClearScreen	158
UIEngine::DisableScreen	159
UIEngine::DoClickDialog	159
UIEngine::DoDialog	159
UIEngine::DoOKDialog	159
UIEngine::DoYesNoDialog	160
UIEngine::DoYesNoCancelDialog	160
UIEngine::GetFont	160

UIEngine::GetUIVersion	160
UIEngine::HandleInput	161
UIEngine::Initialize	162
UIEngine::Operator =	162
UIEngine::ProcessDialog	162
UIEngine::ProcessMenu	163
UIEngine::ProcessScreen	163
UIEngine::RemoveDialog	163
UIEngine::RemoveMenu	163
UIEngine::RemoveScreen	164
UIEngine::RestoreDisplayContext	164
UIEngine::SetDisplayContext	164
UIEngine::SetFont	164

UIEngine::~~UIEngine

Destroys an instance of a UI Engine object.

`~UIEngine()`

UIEngine::ClearDialog

Instructs the UI Engine to remove the current dialog box from the display.

`void ClearDialog()`

Description 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()`

Description This function removes the application-instantiated menu on the display. `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 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()`

Description This function clears the application-instantiated screen from the UI Engine. `ProcessScreen` must be invoked for the UI Engine to be aware of it again.

UIEngine::DisableScreen

Prevents any updates to the screen from appearing on the screen.

```
void DisableScreen()
```

UIEngine::DoClickDialog

Enables a ClickDialog object to be used by means of a single entry point.

```
int DoClickDialog(const char * const pszText = NULL,
    Bitmaps::PREDEFINED_BITMAP const bitmap = Bitmaps::EXCLAMATION)
```

Parameters	<code>pszText</code>	The text to display in the dialog.
	<code>Bitmaps::PREDEFINED_BITMAP const bitmap = Bitmaps::EXCLAMATION</code>	The predefined exclamation point bitmap shapes specified in <code>PredefinedBitmaps.h</code> .

UIEngine::DoDialog

Enables a Dialog object to be used by means of a single entry point.

```
int DoDialog(int AnchorPoint,
    const char * const pszText,
    int flags,
    Bitmaps::PREDEFINED_BITMAP const bitmap = Bitmaps::EXCLAMATION)
```

Parameters	<code>AnchorPoint</code>	The number of pixels from the top of the screen at which the dialog box is to be displayed.
	<code>pszText</code>	The text to display in the dialog.
	<code>flags</code>	A combination of dialog flags.
	<code>Bitmaps::PREDEFINED_BITMAP const bitmap = Bitmaps::EXCLAMATION</code>	The predefined exclamation point bitmap shapes specified in <code>PredefinedBitmaps.h</code> .

UIEngine::DoOKDialog

Enables a OKDialog object to be used by means of a single entry point.

Displays an OKDialog object.

```
Form 1: int DoOKDialog(const char * const pszText = NULL)
Form 2: int DoOKDialog(int anchorPoint,
    const char * const pszText = NULL)
```

Parameters	pszText	The text to display in the dialog.
	anchorPoint	The number of pixels from the top of the screen at which the dialog box is to be displayed.

UIEngine::DoYesNoDialog

Enables a YesNoDialog object to be used by means of a single entry point.

```
int DoYesNoDialog(const char * const pszText = NULL,  
    int flags = DLG_DEFAULT_YES,  
    bool bShowCancel = false)
```

Parameters	pszText	The text to display in the dialog.
	flags	The default selected item.
	bShowCancel	Set to true to display a Cancel option; false otherwise.

UIEngine::DoYesNoCancelDialog

Enables a YesNoCancelDialog object to be used by means of a single entry point.

```
int DoYesNoCancelDialog(const char * pszText = NULL,  
    int flags = DLG_DEFAULT_YES)
```

Parameters	pszText	The text to display in the dialog.
	flags	The default selected item.

UIEngine::GetFont

Retrieves the default font associated with the UI Engine.

```
Form 1: Font const * GetFont()  
Form 2: Font const * GetFont(char FontIndex)
```

Parameters	FontIndex	A pointer to the table containing the fonts used by the UI Engine.
-------------------	------------------	--

Returns A pointer to the default font used to display the characters in the UI Engine.

Description Form 2 retrieves the font from the lookup table.

UIEngine::GetUIVersion

Retrieves 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.

Returns The full version of the UI Engine.

UIEngine::HandleInput

Manages any keypad messages received from the Operating System.

```
virtual RESULT HandleInput(MESSAGE & InputMessage)
```

Parameters

InputMessage	A RIM MESSAGE which contains device events.
--------------	---

Returns This function returns one of five RESULT values:

Return value	Description
CLICKED	This implies the user has clicked the trackwheel. Typically clicking the trackwheel implies the application displays a menu, or if there is a menu or dialog already displayed, it will disappear.
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.

Return value	Description
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” on page 101.). Typically an application would clear the menu and go back waiting for another RIM message, when receiving this.
UNHANDLED	This implies that the UI Engine could not handle 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 (such as, list fields)

UIEngine::Initialize

Sets up the UIEngine object and initializes it for first use.

```
virtual void Initialize()
```

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

UIEngine::Operator =

Sets one UI Engine object equal to another UI Engine object.

```
UIEngine & Operator = (const UIEngine & src)
```

Parameters `src` An initialized UI Engine object.

Returns A UI Engine object that is a duplicate of `src`.

Description `UIEngine::Operator =` sets the left side parameter to be a duplicate of `src`. There should only be one UI Engine object per application.

UIEngine::ProcessDialog

Instructs the UI Engine to display a dialog box on the screen.

```
void ProcessDialog(Dialog & newDialog)
```

Parameters `newDialog` The application-instantiated dialog.

Description 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 menu.

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

UIEngine::ProcessScreen

Instructs the UI Engine to paint or repaint the screen.

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

Parameters `newScreen` The application-instantiated screen.

Description 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::RemoveDialog

Removes a `Dialog` object from the UI Engine.

```
void RemoveDialog(Dialog & DialogToRemove)
```

Parameters `DialogToRemove` The `Dialog` object to remove from the UI Engine.

Description `RemoveDialog` can be called to remove any `Dialog` object from the UI Engine, not only the `Dialog` object currently being displayed.

To remove the current `Dialog` object, call `ClearDialog`.

UIEngine::RemoveMenu

Removes a `Menu` object from the UI Engine.

```
void RemoveMenu(Menu & MenuToRemove)
```

Parameters MenuToRemove The Menu object to remove from the UI Engine.

Description RemoveMenu can be called to remove any Menu object from the UI Engine, not only the Menu object currently being displayed.

To remove the current Menu object, call ClearMenu.

UIEngine::RemoveScreen

Removes a Screen object from the UI Engine.

```
void RemoveScreen(Screen & ScreenToRemove)
```

Parameters ScreenToRemove The Screen object to remove from the UI Engine.

Description RemoveScreen can be called to remove any Screen object from the UI Engine, not only the Screen object currently being displayed.

To remove the current Screen object, call ClearScreen.

UIEngine::RestoreDisplayContext

Shows the display context of the current UI Engine object on the screen.

```
void RestoreDisplayContext()
```

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

UIEngine::SetDisplayContext

Sets the display context of a UI Engine object.

```
int SetDisplayContext()
```

Description This function sets the display context association with the application when the application first instantiates the UI Engine. This enables one application to call another application (one thread to call a function in another thread) and use the display context of the latter.

UIEngine::SetFont

Changes the font used by the UI Engine.

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

```

Form 2: 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 <i>BlackBerry SDK Developer Guide</i> 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 iFontIndex.

Description This function sets the font used by all UIEngine objects in the system, not just the current one. It is a global function.



Note: setFont was originally ReplaceFont.

YesNoChoice

YesNoChoice is a class derived from Choice. It fixes the string array to be either **Yes** or **No**.

To include YesNoChoice functions in your application, you must include `<Choice.h>` in your code.

The YesNoChoice constructor has two forms:

Form 1: YesNoChoice(
 char * Label,
 bool CurrentChoice)

Form 2: YesNoChoice(const YesNoChoice & src)

Parameters	Label	The label associated with the choice box.
	CurrentChoice	If true, the current value in the choice box is Yes ; if false, the current value is No .
	src	An initialized YesNoChoice object.

Form 2 constructs a YesNoChoice object that is a duplicate of src.

Functions

The following functions are listed alphabetically.

YesNoChoice::~YesNoChoice	166
YesNoChoice::GetFieldType	166
YesNoChoice::GetFlag	167
YesNoChoice::GetFieldType	166
YesNoChoice::SetFlag	167

YesNoChoice::~~YesNoChoice

Destroys an instance of a YesNoChoice object.

```
virtual ~YesNoChoice()
```

YesNoChoice::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.
Returns		Returns the type of the base class of the current field (i.e. the type of the first class derived from the Field object)
Description		For a YesNoChoice object, GetFieldType returns CHOICE_FIELD, and pDerived is completed with YES_NO_CHOICE_FIELD.

YesNoChoice::GetFlag

Retrieves the value displayed in the Yes/No Choice field.

```
bool GetFlag()
```

Returns True if the user chooses **Yes**; false otherwise.

YesNoChoice::Operator =

Sets one YesNoChoice object equal to another YesNoChoice object.

```
YesNoChoice & Operator = (const YesNoChoice & src)
```

Parameters	src	An initialized YesNoChoice object.
Returns		A YesNoChoice object that is a duplicate of src.
Description		YesNoChoice::Operator = sets the left side parameter to be a duplicate of src.

YesNoChoice::SetFlag

Sets the value displayed in the Yes/No Choice field.

```
void SetFlag(bool newFlagValue)
```

Parameters	newFlagValue	If true, the value in the choice box is set to Yes ; if false, the value is set to No .
-------------------	---------------------	---

YesNoDialog

YesNoDialog is a class derived from Dialog. It uses a list object as the field, poses an optional question, and puts up the list items **Yes**, **No** and optionally **Cancel**. YN_ITEM is an enumerated type which has the following possible values: ITEM_YES, ITEM_NO, and ITEM_CANCEL.

To include YesNoDialog functions in your application, you must include <YesNoDialog.h> in your code.

The YesNoDialog constructor has two forms:

Form 1: YesNoDialog(
 char const * const pQuestion = NULL,
 YN_ITEM DefaultItem = ITEM_YES,
 bool fShowCancel = false)

Form 2: YesNoDialog(const YesNoDialog & src)

Parameters	pQuestion	A pointer to the string.
	DefaultItem	The default items used in the list field.
	fShowCancel	The option to determine if Cancel is one of the list items.
	src	An initialized YesNoDialog object.

Form 2 creates a YesNoDialog object that is a duplicate of src.

Functions

The following functions are listed alphabetically.

YesNoDialog::~YesNoDialog	168
YesNoDialog::AnswerIsNo	169
YesNoDialog::AnswerIsYes	169
YesNoDialog::GetAnswer	169
YesNoDialog::Go	169
YesNoDialog::Operator =	169
YesNoDialog::SetDefaultItem	170
YesNoDialog::SetQuestion	170
YesNoDialog::ShowCancelItem	170

YesNoDialog::~YesNoDialog

Destroys an instance of a YesNoDialog object.

```
virtual ~YesNoDialog()
```


YesNoDialog::AnswerIsNo

Determines if the user last selected **No**.

```
bool AnswerIsNo()
```

Returns True if the item that was selected last was **No**; false otherwise.

YesNoDialog::AnswerIsYes

Determines if the user last selected **Yes**.

```
bool AnswerIsYes()
```

Returns True if the item that was selected last was **Yes**; false otherwise.

YesNoDialog::GetAnswer

Retrieves the answer last selected by the user.

```
YN_ITEM GetAnswer()
```

Returns The item that was selected last.

YesNoDialog::Go

Draws the dialog on the screen, receives input from the user and returns the result of the user's choice.

```
virtual YN_ITEM Go(UIEngine & UIEngineToUse)
```

Parameters UIEngineToUse The associated UIEngine object to use.

Returns ITEM_YES, ITEM_NO, or optionally ITEM_CANCEL.

Description Go displays the dialog on the screen.

YesNoDialog::Operator =

Sets one YesNoDialog object equal to another YesNoDialog object.

```
YesNoDialog & Operator = (const YesNoDialog & src)
```

Parameters src An initialized YesNoDialog object.

Returns A YesNoDialog object that is a duplicate of src.

Description YesNoDialog::Operator = sets the left side parameter to be a duplicate of src.

YesNoDialog::SetDefaultItem

Sets the item that is initially selected when the dialog box is displayed.

```
bool SetDefaultItem(YN_ITEM DefaultItem)
```

Parameters `DefaultItem` Sets the item that will be the user's default choice.

Returns True if successful; false otherwise.

YesNoDialog::SetQuestion

Sets the question to display in the dialog box.

```
bool SetQuestion(char const * const pQuestion)
```

Parameters `pQuestion` Sets the text string to be used for the question.

Returns True if successful; false otherwise.

YesNoDialog::ShowCancelItem

Determines if a Cancel item is displayed in the list of choices.

```
bool ShowCancelItem(bool fShowCancelItem = true)
```

Parameters `fShowCancelItem` If `fShowCancelItem` is true, **Cancel** is displayed in the list. Otherwise, **Cancel** is not displayed.

Returns True if successful; false otherwise.

Index of functions

B

BeepChoice

GetBeep(), 15
GetFieldType(), 16
SetBeep(), 16

Bitmask

BlankBitmask(), 19
IsBitSet(), 19
LeftShift(), 19
SetBit(), 19
SetBitmask(), 20
SetSize(), 20

BusyStatus

Disable(), 22
Draw(), 22
Reset(), 23
ResetBitmap(), 23

C

Choice

ChangeChoice(), 25
ChangeChoiceEx(), 26
ChangeChoiceList(), 27
GetFieldType(), 28
GetLabel(), 28
GetNumEntries(), 29
GetSelectedIndex(), 29
GetSelectedValue(), 29
GetValueAtIndex(), 29
SetChoices(), 30
SetLabel(), 30
SetNumEntries(), 30
SetNumericString(), 31
SetSelectedIndex(), 31
UpdatePtr(), 31

ClickDialog

Go(), 33
SetAnchorPoint(), 33
SetBitmap(), 34
SetDisplayString(), 34

D

DateTime

FieldType(), 37
FieldTypePosition(), 38
Format(), 38

NextFieldType(), 40
NumberOfFields(), 40
PrevFieldType(), 40
SetFormat(), 41

DecimalEdit

GetNumber(), 42
SetNumber(), 43

Dialog

ClearDialog(), 45
DisplayDialog(), 45
SetAnchorPoint(), 45
SetBitmap(), 46
SetDisplayString(), 46
SetField(), 46
UpdatePtr(), 47

E

Edit

AddProperties(), 50
Append(), 50
ClearProperties(), 50
Delete(), 51
GetBookmark(), 51
GetBuffer(), 52
GetBufferLength(), 52
GetBufferSize(), 52
GetCursorOffset(), 52
GetFieldType(), 52
GetLabel(), 53
GetProperties(), 53
GetStringLength(), 53
Insert(), 53
operator=, 54
ReplaceSelection(), 54
SetBookmark(), 55
SetBuffer(), 55
SetCursorOffset(), 56
SetLabel(), 56
SetProperties(), 56
SetSelection(), 58
UpdatePtr(), 58

EditDate

ChangeChoice(), 61
FormatDate(), 61
GetDate(), 62
GetDateString(), 62
GetFieldType(), 62
IsValid(), 63

Index of functions

- operator =, 63
- SetDate(), 63
- SetDateFormat(), 63
- SetLabel(), 64
- EditTime
 - ChangeChoice(), 66
 - FormatTime(), 66
 - GetDay(), 67
 - GetFieldType(), 67
 - GetTime(), 68
 - GetTimeFormat(), 68
 - GetTimeString(), 68
 - operator =, 68
 - SetDay(), 68
 - SetLabel(), 69
 - SetRound(), 69
 - SetTime(), 69
 - SetTimeFormat(), 69
 - SetTimeZone(), 69

F

- Field
 - GetArrows(), 72
 - GetFieldType(), 72
 - GetFocusRect(), 73
 - GetFont(), 73
 - GetHeight(), 73
 - GetJustification(), 74
 - GetManager(), 74
 - GetRect(), 74
 - GetStyles(), 74
 - GetTag(), 74
 - GetWidth(), 75
 - GiveFocus(), 75
 - HasFocus(), 75
 - Invalidate(), 76
 - IsDirty(), 76
 - IsFocusVisible(), 76
 - IsHidden(), 76
 - IsIntegralHeight(), 76
 - IsOnLCD(), 77
 - IsReadOnly(), 77
 - IsVisible(), 77
 - MarkAsClean(), 77
 - MarkAsDirty(), 77
 - OnKey(), 77
 - OnScroll(), 78
 - PutFocusAtBottom(), 78
 - QueryPixelHeight(), 79
 - Redraw(), 79
 - ResetDimensionDependentData(), 79
 - SetArrows(), 79
 - SetBounds(), 80

- SetFont(), 80
- SetIntegralHeight(), 80
- SetJustification(), 80
- SetManager(), 81
- SetTag(), 81
- SetVisible(), 81
- TakeFocus(), 82
- FieldManager, 83
 - AddField(), 84
 - BottomOfDisplay(), 84
 - Draw(), 85
 - GetArea(), 85
 - GetFieldWithFocus(), 85
 - GetHeight(), 86
 - GetLastField(), 86
 - GetUIEngine(), 87
 - GetWidth(), 87
 - Invalidate(), 88
 - IsDisabled(), 88
 - OnFieldUpdate(), 88
 - OnFocusEvent, 88
 - operator =, 89
 - PageDownDisplay(), 89
 - PageUpDisplay(), 89
 - PutFieldAtBottomOfDisplay(), 89
 - PutFieldAtTopOfDisplay(), 89
 - RemoveAllFields(), 90
 - RemoveField(), 90
 - ResetDimensionDependentData(), 90
 - SetFieldWithFocus(), 90
 - SetFieldwithFocus(), 90
 - TopOfDisplay(), 91
 - UpdateNotify(), 91

L

- Label
 - operator =, 93
 - SetText(), 93
- List
 - Delete(), 96
 - GetColumnFont(), 96
 - GetDisplayRange(), 96
 - GetFieldType(), 97
 - GetNumEntries(), 97
 - GetSelectedIndex(), 97
 - GetTopLineEntry(), 97
 - Insert(), 98
 - IsRangeSelected(), 98
 - NowDisplaying(), 98
 - PutColumn(), 98
 - Redraw(), 99
 - SetColumnFont(), 99
 - SetEnableMultipleSelection(), 100

SetNumEntries(), 100
 SetSelectedIndex(), 100
 SetTopLineEntry(), 100

M

Menu

GetSelectedIndex(), 103
 GetTopIndex(), 103
 HideItems(), 104
 SetMenuItems(), 105
 SetTopIndex(), 107

O

OKDialog

operator =, 109
 Go(), 108
 SetQuestion(), 109

P

PropertyName(), 85, 125

R

RichText, 110

FindOffset(), 111
 FindWidth(), 112
 GetAttributes(), 112
 GetBuffer(), 112
 GetBufferLength(), 112
 GetCursorOffset(), 113
 GetFieldType(), 113
 SetAttributes(), 113
 SetBuffer(), 113
 SetBufferAppended(), 114
 SetCursorOffset(), 114
 SetFonts(), 114
 SetLabel(), 114
 TextRangeAttribute structure, 110
 UpdatePtr(), 115

S

Screen

AddField(), 117
 ClearScreen(), 117
 Close(), 117
 DidSave(), 118
 GetUIEngine(), 118
 Invalidate(), 118
 IsDisabled(), 118
 IsEmptyOfLabels(), 119
 IsFieldDirty(), 119
 OnChangeOption(), 119

OnChangeOptions(), 119
 OnClose(), 120
 OnFieldUpdate(), 120
 OnKey(), 120
 OnMenu(), 120
 OnMenuItem(), 121
 OnMessage(), 121
 OnRoll(), 122
 OnSave(), 121
 Process(), 122
 PutFocusAtBottom(), 122
 RemoveAllLabels(), 122
 RemoveLabel(), 123
 ResetDimensionDependentData(), 123
 ResetScreen(), 123
 Save(), 123
 SetMenu(), 124
 SetScreenDisabled(), 124

Screen::Process, 150

Separator

GetFieldType(), 126
 operator =, 126
 SetText(), 126

Status

ClearModalStatus(), 130
 DisplayModalStatus(), 130
 GetDisplayTime(), 130
 SetBitmap(), 131
 SetDisplayTime(), 131
 SetText(), 132

StringList

GetFieldType(), 133
 operator =, 134
 SetStringList(), 134

structures

TextRangeAttribute, 110
 TIME, 36

T

Table

FindNextCell(), 138
 GetArea(), 138
 GetCaption(), 139
 GetColumnWidth(), 139
 GetLeftRightIndent(), 139
 GetNumCells(), 140
 GetNumColumns(), 140
 GetNumRows(), 140
 GetRowHeight, 140
 GetSelectedColumn(), 141
 GetTopCell(), 141
 GetTopColumn(), 141
 GetTopRow(), 141

Index of functions

- IsCellSelectable(), 142
- MoveHighlight(), 143
- OnChangedFocus(), 144
- operator =, 144

Title

- operator =, 153
- SetLocation(), 153
- SetText(), 153

TopBottomChoice

- GetFieldType(), 155
- GetFlag(), 156
- operator=, 156
- SetFlag(), 156
- TopBottomChoice(), 155

U

UIEngine

- ClearDialog(), 158
- ClearMenu(), 158
- ClearScreen(), 158
- DisableScreen(), 159
- GetUIVersion(), 160
- HandleInput(), 161

- Initialize(), 162
- ProcessDialog(), 162
- ProcessMenu(), 163
- ProcessScreen(), 163
- RestoreDisplayContext(), 164
- SetFont(), 164

Y

YesNoChoice

- GetFieldType(), 166
- GetFlag(), 167
- operator =, 166
- SetFlag(), 167

YesNoDialog

- AnswerIsNo(), 169
- AnswerIsYes(), 169
- GetAnswer(), 169
- Go(), 169
- operator =, 169
- SetDefaultItem(), 170
- SetQuestion(), 170
- ShowCancelItem(), 170

Index

A

- design guidelines
 - See also user control
- activating
 - clickdialog boxes, 33
- adding
 - edit properties, 50
 - list column, 98
 - screen field object, 84
- aesthetics, 8
- allowing for table captions, 150
- appending edit text, 50

B

- beepchoice boxes
 - functions, 14
 - getting the beep, 15
 - getting the field type, 16
 - setting the beep, 16
- buffer
 - getting the edit, 52
 - getting the edit length, 52
 - setting the edit, 55
- busy status boxes
 - disabling, 22
 - drawing, 22
 - functions, 21
 - resetting, 23
 - resetting the bitmap, 23

C

- changing the choice box, 25, 26, 27
- choice boxes
 - changing, 25, 26, 27
 - entering numbers, 31
 - functions, 25
 - getting string associated with index, 29
 - getting the field type, 28
 - getting the label, 28
 - getting the number of entries, 29
 - getting the selected index, 29
 - getting the selected value, 29
 - setting the choices, 30
 - setting the label, 30
 - setting the number of entries, 30
 - setting the selected index, 31

- updating pointer, 31
- choosing table captions, 139
- clarity, 8
- clearing
 - edit properties, 50
 - screens, 117
 - UI dialog boxes, 45, 158
 - UI menu, 158
 - UI screen, 123, 158
- clickdialog boxes
 - activating, 33
 - functions, 32
 - setting the anchor point, 33
 - setting the bitmap, 34
 - setting the display string, 34
- consistency, 8
- constructing
 - topbottomchoice boxes, 155
- controls, 11
- cursor offset
 - getting the edit, 52
 - setting the edit, 56

D

- decimal edit fields
 - functions, 42
 - getting the number, 42
 - indicating content changes, 125
 - setting the number, 43
- deleting
 - columns in tables, 137
 - edit fields, 51
 - fields, 90
 - list items, 96
 - rows in tables, 137
 - screen labels, 123
- design guidelines, 7–9
 - aesthetics, 8
 - clarity, 8
 - consistency, 8
 - controls, 11
 - dialog boxes, 8
 - feedback, 9
 - forgiveness, 9
 - icons, 8
 - intuitiveness, 8
 - keyboard, 11
 - status boxes, 8

Index

- trackwheel, 11
- dialog boxes
 - clearing, 45, 158
 - displaying, 45
 - functions, 44
 - setting fields, 46
 - setting the anchor point, 45
 - setting the bitmap, 46
 - setting the display string, 46
 - updating pointers, 47
- disabling
 - busy status box, 22
 - UI screen, 159
- display context
 - restoring, 164
- displaying
 - busy status box, 22
 - current list, 98
 - dialog boxes, 45
 - modal status box, 130
 - OKdialog boxes, 108
 - status box text, 132
 - yesnodialog box canceled item, 170

E

- edit fields
 - adding properties, 50
 - appending text, 50
 - clearing properties, 50
 - deleting text, 51
 - getting bookmarks, 51
 - getting buffer size, 52
 - getting properties, 53
 - getting the buffer, 52
 - getting the buffer length, 52
 - getting the cursor offset, 52
 - getting the field type, 52
 - getting the label, 53
 - getting the string length, 53
 - inserting strings, 53
 - replacing the selection, 54
 - setting bookmarks, 55
 - setting properties, 56
 - setting the buffer, 55
 - setting the cursor offset, 56
 - setting the label, 56
 - setting the selection, 58
- exceptions, watchPuppy, 21

F

- feedback, 9
- field manager
 - adding a field object, 84

- functions, 83
 - getting field with focus, 85
 - getting height, 86
 - getting last field, 86
 - getting width, 87
 - invalidating, 88
 - notification of changed data, 88
 - putting fields at bottom of display, 89
 - putting fields at top of display, 89
 - removing all fields, 90
 - removing fields, 90
 - setting fields with focus, 90
 - setting focus at bottom, 84
 - setting focus at top, 91

fields

- determining changes, 76, 77
- functions, 71
- getting type, 72
- marking clean, 77
- marking dirty, 76, 77
- setting justification, 80

FieldType

- constants, 72

finding in tables

- next cell, 138
- selectable cells, 142

fonts, setting, 164

forgiveness, 9

functions

- beepchoice boxes, 14
- busy status boxes, 21
- choice boxes, 25
- clickdialog boxes, 32
- decimal edit fields, 42
- dialog boxes, 44
- edit, 48
- field manager, 83
- fields, 71
- lists, 95
- OKdialog boxes, 108
- richtext, 110
- screens, 116
- separators, 125
- status boxes, 128
- stringlist items, 133
- tables, 135
- title, 152
- user interface, 157
- yesnochoice boxes, 166
- yesnodialog boxes, 168

G

- getting

- beep choice field type, 16
- beepchoice beep, 15
- bookmarks, 51
- buffer size for edit fields, 52
- choice box label, 28
- choice field type, 28
- decimal edit field number, 42
- edit field buffer, 52
- edit field cursor offset, 52
- edit field label, 53
- edit field properties, 53
- edit field string length, 53
- edit field type, 52
- field with focus, 85
- first cell in a table, 141
- first column in a table, 141
- first row in a table, 141
- height, 86
- last field, 86
- length of edit field buffer, 52
- list field type, 97
- number of cells in a table, 140
- number of columns in a table, 140
- number of entries, 29
- number of list entries, 97
- number of rows in a table, 140
- selected cell in a table, 140
- selected choice box index, 29
- selected choice box value, 29
- selected column in a table, 141
- selected list index, 97
- selected menu index, 103
- selected row in a table, 141
- separator field type, 126
- status display time, 130
- string for index, 29
- string list field type, 133
- table field type, 139
- top list index entry, 97
- top menu index, 103
- topbottomchoice box flag, 156
- topbottomchoice field type, 155
- UI version, 160
- width, 87
- yes no choice field type, 166
- yesnochoice flag, 167
- yesnodialog box answer, 169
- guidelines
 - design See design guidelines
 - style See style guidelines

H

- handling input, 161

I

- initializing the user interface, 162
- input, handling, 161
- inserting
 - edit fields, 53
 - list index entry, 98
- inserting columns in tables, 142
- inserting rows in tables, 142
- intuitiveness, 8
- invoking a menu, 10

K

- keyboard, 11

L

- lists
 - adding a column, 98
 - deleting items, 96
 - displaying current, 98
 - enabling multiple selection, 100
 - functions, 95
 - getting font for the column, 96
 - getting the field type, 97
 - getting the number of entries, 97
 - getting the selected index, 97
 - getting the top index entry, 97
 - inserting an item, 98
 - redrawing lines, 99
 - setting the font for the column, 99
 - setting the number of entries, 100
 - setting the selected index, 100
 - setting the top position, 100

M

- menu items
 - cancel menu item, 10
 - clearing, 158
 - design, 10
 - designing, 10
 - format, 10
 - getting selected index, 103
 - getting top, 103
 - hide menu, 10
 - invoking a menu, 10
 - setting, 105
 - setting top, 107

O

- object-action paradigm, 8
- OKdialog boxes
 - displaying, 108

Index

- functions, 108
- setting the text, 109

P

- processing
 - dialog boxes, 162
 - menu items, 163
 - screens, 163
- properties
 - getting for edit fields, 53
 - setting for edit fields, 56

R

- redrawing
 - list lines, 99
 - tables, 146
- removing
 - all fields, 90
 - edit fields, 51
 - fields, 90
 - list items, 96
 - screen labels, 123
- replacing the selection, 54
- resetting
 - bitmap in a busy status box, 23
 - screen, 123
- resetting the busy status box, 23
- restoring the display context, 164
- richtext
 - functions, 110
 - get attributes, 112
 - get buffer, 112
 - get buffer length, 112
 - get cursor offset, 113
 - set attributes, 113
 - set buffer, 113
 - set cursor offset, 114
 - set label, 114
- running yesnodialog boxes, 169

S

- screens
 - clearing, 117, 158
 - close, 117, 120
 - disabling, 118, 124, 159
 - edit, 48
 - functions, 116
 - gets the associated UI, 118
 - handling key input, 120
 - implementing change options, 119
 - is field dirty, 119
 - processing, 122

- receiving messages, 121
- removing labels, 123
- requesting menus, 120
- resetting, 123
- saving, 118, 121, 123
- selecting menu items, 121
- setting menus, 124
- title, 152
- titles, 9
- trackwheel movement, 122
- search, 9
- selecting (in tables)
 - a cell, 149
 - cells, 146
 - information to be displayed, 143
- separators
 - functions, 125
 - getting field type, 126
 - setting text, 126
- setting
 - anchor point for clickdialog boxes, 33
 - anchor points in dialog boxes, 45
 - beepchoice box beep, 16
 - bitmaps for clickdialog boxes, 34
 - bitmaps in dialog boxes, 46
 - bookmarks, 55
 - choice box choices, 30
 - choice box labels, 30
 - column width, 147
 - decimal edit field number, 43
 - default yesnodialog item, 170
 - dialog box display strings, 46
 - dialog box fields, 46
 - display string for clickdialog boxes, 34
 - edit field buffer, 55
 - edit field cursor offset, 56
 - edit field label, 56
 - edit field properties, 56
 - field justification, 80
 - field with focus, 90
 - font, 164
 - highlight mode, 147
 - justification for tables, 148
 - menu items, 105
 - number of entries, 30
 - number of list entries, 100
 - row height, 149
 - selected choice box index, 31
 - selected list index, 100
 - status bitmaps, 131
 - status display time, 131
 - status text, 132
 - stringlist items, 134
 - table borders, 137, 147

- table properties, 148
- table size, 149
- table structure, 149
- the text for OKdialog boxes, 109
- title location, 153
- title text, 153
- top cell in a table, 150
- top index, 107
- top line, 100
- topbottomchoice flag, 156
- yesnochoice flag, 167
- yesnodialog box default items, 170
- yesnodialog box questions, 170
- showing
 - current list, 98
 - dialog boxes, 45
 - modal status box, 130
 - status box text, 132
 - text in table cells, 145
 - yesnodialog box canceled item, 170
- status boxes
 - clearing modal, 130
 - displaying modal, 130
 - functions, 128
 - getting the display time, 130
 - purpose, 128
 - setting bitmaps, 131
 - setting text, 132
 - setting the display time, 131
- stringlist items
 - functions, 133
 - getting the field type, 133
 - setting the stringlist, 134
- strings
 - getting length for the edit field, 53
 - inserting, 53
- style guidelines, 9–11
 - menus See menu items
 - screen titles, 9

T

- tables
 - allowing for captions, 150
 - choosing captions, 139
 - deleting columns, 137
 - deleting rows, 137
 - finding selectable cells, 142
 - finding the next cell to select, 138
 - functions, 135
 - getting field type, 139
 - getting first cell, 141
 - getting first row, 141
 - getting number of cells, 140

- getting number of columns, 140
- getting number of rows, 140
- getting selected cells, 140
- getting selected column, 141
- getting selected row, 141
- getting the first column, 141
- inserting columns, 142
- inserting rows, 142
- redrawing, 146
- selecting a cell, 149
- selecting cells, 146
- selecting information to be displayed, 143
- setting borders, 137, 147
- setting column width, 147
- setting highlight mode, 147
- setting justification, 148
- setting properties, 148
- setting row height, 149
- setting table size, 149
- setting table structure, 149
- setting the top cell, 150
- showing cell text, 145
- updating the caption, 150
- title bars
 - setting text, 153
 - setting the location, 153
- topbottomchoice boxes
 - constructing, 155
 - functions, 155
 - getting the field type, 155
 - getting the flag, 156
 - setting the flag, 156
- trackwheel, 11

U

- user control, 7–8
- user interface
 - clearing dialog boxes, 158
 - clearing menu items, 158
 - clearing screens, 158
 - disabling screens, 159
 - functions, 157
 - getting the version, 160
 - handling input, 161
 - initializing, 162
 - processing dialog boxes, 162
 - processing menu items, 163
 - processing screens, 163
 - restoring the display context, 164
 - setting fonts, 164

V

- version, getting UI, 160

Index

W

watchPuppy, 21

Y

yesnochoice boxes
 functions, 166
 getting the flag, 167
 setting the flag, 167
yesnochoice fields

 getting the field type, 166
yesndialog boxes
 answer is no, 169
 answer is yes, 169
 displaying the canceled item, 170
 functions, 168
 getting the answer, 169
 running, 169
 setting the default item, 170
 setting the question, 170



© 2002 Research In Motion Limited
Produced in Canada