LANSA

Newsletter

April 2013

Adopting DirectX in Visual LANSA applications

Visual LANSA version 13 now has a choice of application rendering engine: the old Win32 or the new DirectX.

Using DirectX as a rendering engine is a necessary step forward for the LANSA runtime, and will help to keep Visual LANSA at the forefront of Windows application development as point and click desktop applications make way for touch friendly web and tablet style user interfaces.
With the inevitable rise of Windows 8, hybrid laptop/tablets and mobile computing, application styles are changing rapidly, and version 13 introduces new user designed controls, dynamic styles, mouse events, popups, animations and more.

Technology Insurance

With the change of a single flag, existing applications can start using the DirectX engine and will run exactly as they did in version 12. LANSA has gone to great lengths to make sure that "flicking the switch" and adopting the DirectX runtime, is as trouble free as possible.

However, despite LANSA's best efforts, there are few scenarios where the adoption of a new technology has led to the occasional concession. This newsletter contains a separate document (**AdoptingDirectX.pdf**) that outlines these issues and provides a practical approach to managing the resulting development challenges.

This is a valuable document for all customers looking to incorporate DirectX as part of their LANSA strategy.

In This Issue

Understanding Carousel and User Designed Controls in Visual LANSA V13

Many modern user interfaces have gone far beyond showing simple columnar lists and instead use free format panels of information, images, or both. Ebay item listings and iTunes album view are both great examples of this. Rather than the user seeing a long list of names, controls such as carousel can allow a user to browse through data visually.

Given the requirements, it is no longer possible for a predefined control to match the needs of the developer. Even something as simple as a carousel item is almost endlessly configurable and control has to be handed over to the developer to determine how each of the panels will appear.

User Designed Controls (UDC)

Carousel (prim_caro) is one of a number of new DirectX controls in version 13, which, along with Book (prim_book) , Tree (prim_tree) and Tile (prim_tile), address the growing need for greater interface flexibility. Developers still want the power and simplicity of LANSA's list handling, but, to incorporate the rise of touch enabled devices and Windows 8, also need their screen designs to go far beyond the strictures put in place by tradition controls such as Tree view (prim_trvw).

UDCs act just like lists, supporting Add_Entry, SelectList and the other typical list commands. However, instead of the control determining the appearance of each item and the columns of the list determining the fields, the developer is free to create a design, or many different designs, to be added to the control, using whatever fields are required. When an entry is added to the control, an instance of the design is created and populated with the fields based on the current values.

The reusable part acting as the design for the UDC needs to implement a specific interface.

```
Begin_Com Role(*EXTENDS #Prim_panl *Implements #Prim_caro.ICarouselDesign
*ListFields #ListFields)
Group_By Name(#ListFields) Fields(#Givename #Surname #Empimg #Empno)
```

This allows the UDC to communicate with the design instances at runtime, allowing them to respond to a change of Focus or Selection. Each UDC has its own design interface (prim_tile.iTileDesign, prim_tree.iTreeDesign etc.),
When the Carousel is defined, the design reusable part is specified on the Define_com. This is enough for Visual LANSA to be able to hook up the necessary fields and manage the underlying list.

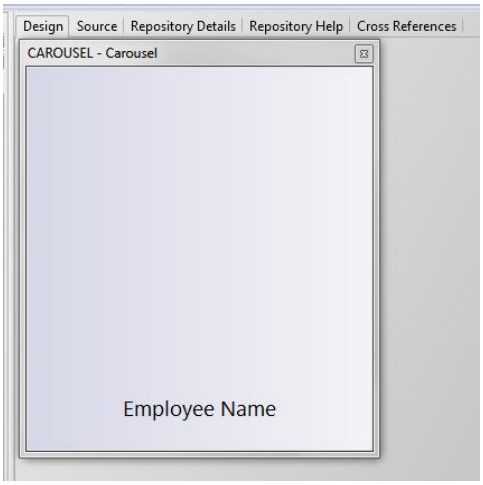
```
Define_Com Class(#prim_caro<#CarouselDesign>) Name(#UDCCarousel)
```

This simple pattern is repeated on all UDCs with a control having a design and many instances of the design being created at runtime as entries are added. However, because of this, there is an overhead to using a UDC. Creating many component instances is relatively expensive, and UDCs are best suited to UIs that are high graphical but show a relatively small amounts of data per screen. For high volume scenarios, other techniques are recommended.

Example

The following example is a simple form that uses a carousel and a track bar for navigation. You will need the demonstration materials and it is assumed the partition has long name support.

Firstly, create a reusable part called CarouselDesign and copy the code below (next page). This will act as the design for each of the items.



```
Function Options(*direct)
Begin_Com Role(*EXTENDS #XDXBasePanel *implements #Prim_caro.ICarouselDesign
*ListFields #ListFields) Height(341) Layoutmanager(#Layout) Width(307)
* Fields mapped in when the entry is added to the Book
Group_By Name(#ListFields) Fields(#Givename #Surname #Empimg #Empno)
Define_Com Class(#Prim_atlm) Name(#Layout)
Define_Com Class(#Prim_atli) Name(#LayoutItem) Attachment(Center) Manage(#Content)
Marginbottom(2) Marginleft(20) Marginright(20) Margintop(12) Parent(#Layout)
Define_Com Class(#Prim_atli) Name(#LayoutItemBG) Attachment(Center)
Manage(#Background) Parent(#Layout)
Define_Com Class(#PRIM_ATLI) Name(#LayoutItemCaption) Attachment(Bottom)
Manage(#Name) Marginbottom(12) Marginleft(20) Marginright(20) Margintop(2)
Parent(#Layout)
Define_Com Class(#PRIM_Panl) Name(#Background) Displayposition(1) Height(341)
Layoutmanager(#Layout) Left(0) Parent(#COM_OWNER) Tabposition(1) Tabstop(False)
Top(0) Width(307)
Define_Com Class(#PRIM_Panl) Name(#Content) Displayposition(1) Height(263)
Left(20) Parent(#Background) Style(#ContentStyle) Tabposition(1) Tabstop(False)
Top(12) Width(267)
Define_Com Class(#prim_labl) Name(#Name) Alignment(Center) Caption('Employee
Name') Displayposition(2) Left(20) Parent(#Background) Style(#LargeText)
Tabposition(2) Tabstop(False) Top(279) Verticalalignment(Center) Width(267)
Define_Com Class(#prim_vs.Style) Name(#ContentStyle)
Backgroundbrush(#ContentBrush)
Define_Com Class(#prim_vs.ImageBrush) Name(#ContentBrush) Sizing(BestFit)
Define_Com Class(#prim_vs.Style) Name(#LargeText) Fontsize(14)
Define_Com Class(#Prim_bmp) Name(#Image) Reference(*Dynamic)

Mthroutine Name(OnAdd) Options(*redefine)
#Com_owner.Opacity := 30
#Com_owner.Cursor <= #sys_appln.Cursors<Hand>
#Name := ("&1 &2 (&3)").Substitute( #GiveName #Surname #Empno )
* A panel with an image brush is a better technique than simply using an image
control.
* DirectX has inbuilt mechanisms that handle images extremely well.
#Image <= #sys_appln.CreateBitmap
#Image.FileName := #Empimg.FileName
#ContentBrush.Image <= #Image
Endroutine

Mthroutine Name(onItemGotFocus) Options(*Redefine)
#Com_owner.Opacity := 100
Endroutine

Mthroutine Name(onItemLostFocus) Options(*Redefine)
#Com_owner.Opacity := 30
Endroutine
End_Com
```

Secondly, create a new form called FormWithCarousel and copy the code below. This defines the carousel and the design that it will use. Compile both and execute the form, ensuring that you execute as DirectX.

Once you've seen it run, execute again in debug to see how the design instances are created and how they react with focus change.

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Caption('Carousel') Clientheight(492)
Clientwidth(890) Height(530) Layoutmanager(#Layout) Left(114) Style(#Background)
Top(208) Width(906)
```

```
* User Designed Control - Carousel
* Individual items are made by adding entries as per typical LANSA list processing
* Fields in the list are defined by the *ListFields parameter of the Design being
made
* Prim_caro defines the carousel
* #CarouselDesign defines the appearance of the items created
Define_Com Class(#prim_caro<#CarouselDesign>) Name(#UDCCarousel)
Displayposition(1) Height(442) Left(0) Navigationstyle(None) Parent(#COM_OWNER)
Tabposition(1) Top(0) Width(890)
* Trackbar used to navigate through the carousel
Define_Com Class(#prim_tkbr) Name(#TrackBar) Displayposition(2) Left(20)
Parent(#COM_OWNER) Tabposition(2) Tickstyle(None) Top(442) Value(1) Width(850)
Define_Com Class(#prim_atlm) Name(#Layout)
Define_Com Class(#PRIM_ATLI) Name(#ATLI_1) Attachment(Center) Manage(#UDCCarousel)
Parent(#Layout)
Define_Com Class(#PRIM_ATLI) Name(#ATLI_2) Attachment(Bottom) Manage(#TrackBar)
Marginleft(20) Marginright(20) Parent(#Layout)
Define_Com Class(#Prim_timr) Name(#TrackBarTimer) Interval(300)
* Simple white background - Defined locally for simplicity
* Styles are best defined as part of a Visual Style.
Define_Com Class(#prim_vs.style) Name(#Background) Normbackcolor(White)

Evtroutine Handling(#Com_owner.CreateInstance)
#TrackBarTimer.Stop
#Com_owner.Load
Endroutine

Mthroutine Name(Load) Help("Create carousel items") Access(*Private)
* Create Carousel Items
Select Fields(#Surname #Givename #empno) From_File(pslmst)
Fetch Fields(#empimg) From_File(pslimg) With_Key(#Empno)
* Adding an entry creates an instance of the design (#CarouselDesign)
* The fields specified by the *Listfields parameter in the design will be
populated in the design instance.
Add_Entry To_List(#UDCCarousel)
Endselect
#UDCCarousel.Items<1>.Focus := True
Endroutine

Evtroutine Handling(#TrackBar.Changed)
* A timer is used so that mulitple changes to the trackbar aren't immediately
transmitted to the carousel, just the last one.
* This provides a smoother user experience.
#TrackBarTimer.Stop
#TrackBarTimer.Start
Endroutine

Evtroutine Handling(#UDCCarousel.ItemGotFocus) Item(#Item)
* Update the trackbar to match the carousel position.
#TrackBar.Value := #Item.Entry
Endroutine

Evtroutine Handling(#TrackBarTimer.Tick)
* When the timer fires, the user has stopped making changes to the trackbar
position.
* Update the carousel position and stop the timer.
#UDCCarousel.Items.Item<#TrackBar.Value>.Focus := True
#TrackBarTimer.Stop
Endroutine
End_Com
```

PNG images in LANSA V13

Version 13 of Visual LANSA introduces full support for PNG format images.

When an application is executed with DirectX rendering, PNG images will be shown correctly including any transparency embedded in to the image. This greatly improves the appearance of applications, allowing backgrounds to appear through an image. PNG images can be enrolled in the repository as Bitmaps.

Making your own images

Whilst making your own images is typically something best left to professional graphic artists, it is nonetheless possible to take existing PNG images and to cut and paste bits from a variety of images to make new ones. Free tools such as Paint.Net greatly simplify this process.

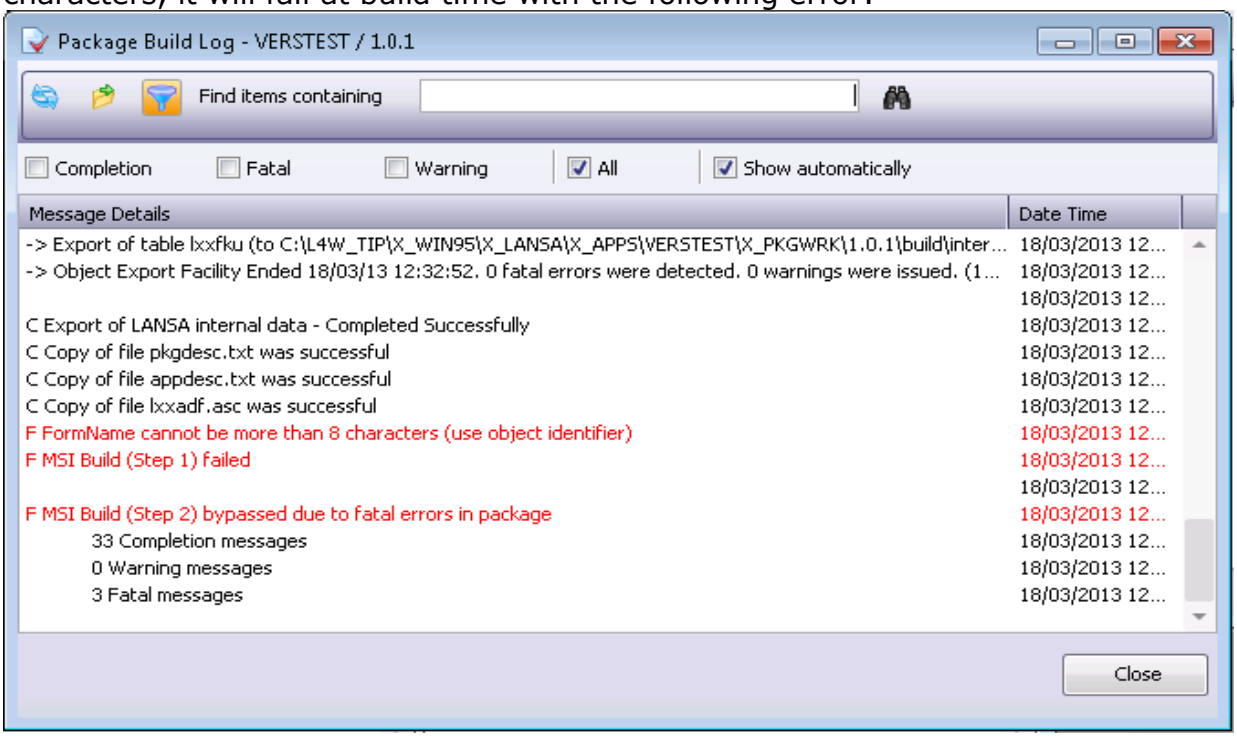
LANSA images

The Visual LANSA IDE no longer uses icons and bitmaps. It now uses PNG format images. These images, and a great many more are available on request. Contact LANSA support to request the images.

Note:
The file contains over 7000 images ranging in size from 16x16 to 512x512. All images are supplied as is.

V13 Deployment tool does not allow you to enter a 9 character Form to Execute

In version 13, an issue has been found in the deployment tool with regards to 9 character Form Names. Although the deployment tool allows you to enter the full 9 characters, it will fail at build time with the following error:

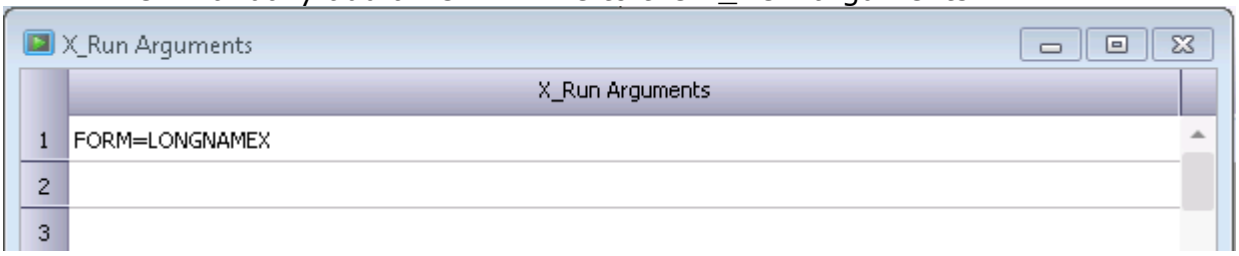


This build error message is not correct, as form names/Object Identifiers can be up to 9 characters long. Note that when creating a long filename object, LANSA will automatically generate an *8* character Identifier, so this will only affect users that manually change the Identifier to something longer, or those with existing 9 character form names from V12.

Solution

The issue is being fixed in the next release for V13, however in the meantime we suggest you use the following workaround:

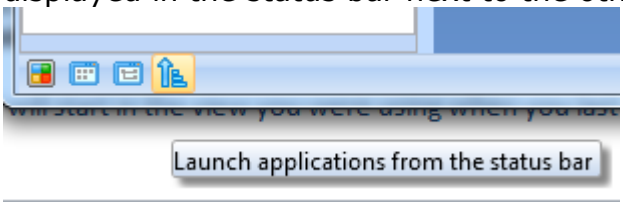
- First, blank out the Form to Execute parameter
- Then manually add a FORM= line to the X_RUN arguments:



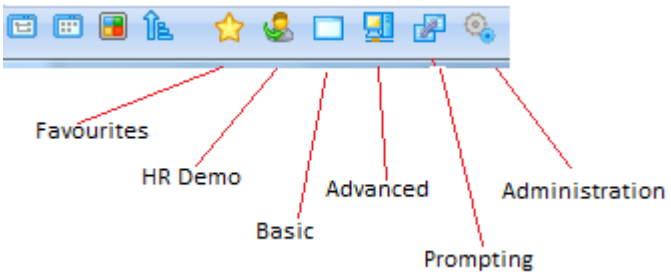
VLF new feature in V13 - Launching Applications from the status bar

Launching Applications from the Status Bar

When the Framework is executed using RenderType M, the launch button  is displayed in the status bar next to the other navigation pane view buttons:



When the launch button is clicked, applications in the Framework are arranged in the status bar:



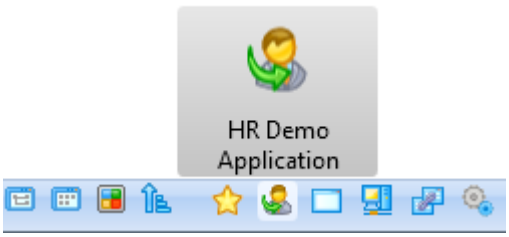
If an application has views, the view is visualised. If an application has no business objects it is not shown.

The applications or views respond to two events:

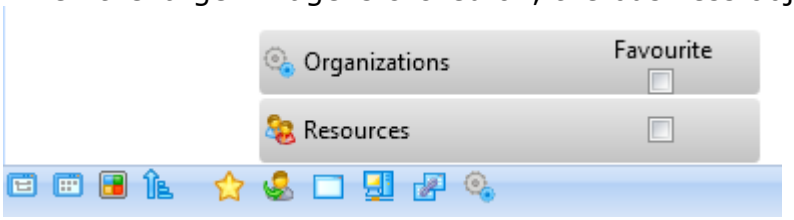
- Mouse hover
- Click

Mouse hover

With a mouse hover a larger icon with the application/view caption appears:



When the larger image is clicked on, the business objects in the application pop up:



If the popup item is clicked, it triggers the default business object's behaviour, as if you clicked on the business object in the navigation pane.

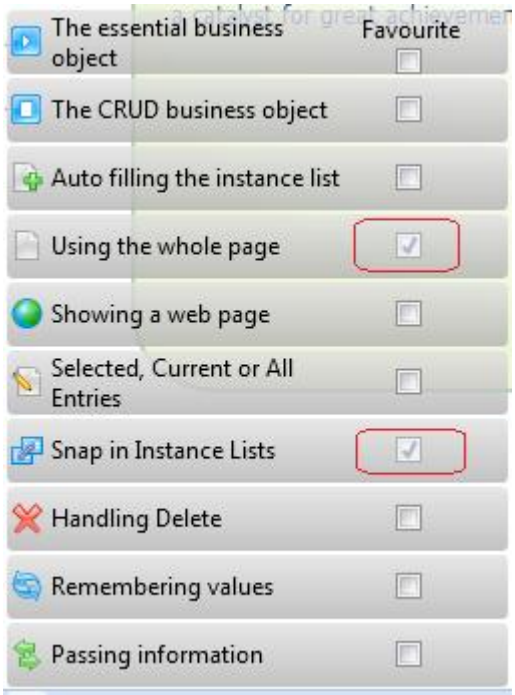
You can also make the business object a favourite by checking the Favourite checkbox. Note: the Business Object will be added to the first application that allows favourites in the sequence they appear. If you need to add it to another application you need to use either the Tree or List Navigation View.

Click

If you click on the application/view, the behaviour is exactly the same as clicking on the larger image.

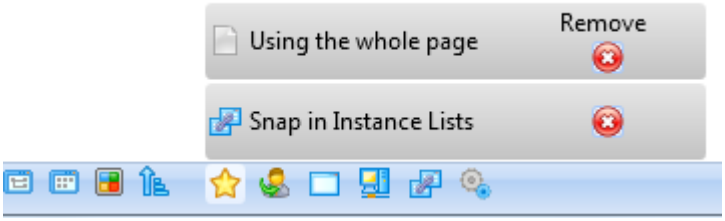
Removing a favourite business object

An application which has business objects that have been made favourites will have the Favorite checkbox is ticked but disabled:



This is because one business object can be a favourite in more than one application.

To remove a business object from a favourite application, hover or click on the favourite application:

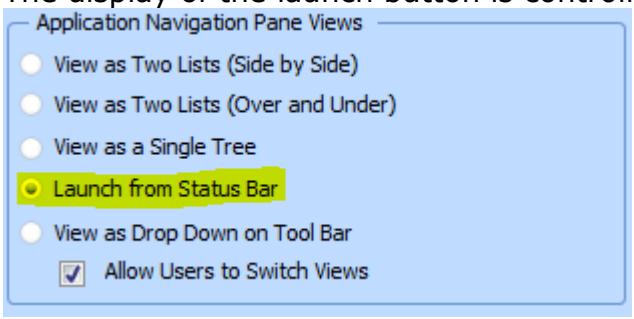


Click the Remove button to remove the business object from the application's Favourites.

Warning:
Due to space constraints, this navigation option may not be suited to Frameworks with a large number of applications and/or applications with large number of business objects. In those cases use any of the other three navigation pane views.

Enabling the Launch button

The display of the launch button is controlled in the Framework properties:



New LongRange version available (Version 11 - RV11)

A new LongRange version is available and it contains lots of new features:

1. **General Appearance, Navigation and Functionality Improvements**
   - Submit device language to the server
   - Submit current device language to server
   - Allow clear and search icons and associated events inside textbox fields.
   - More options to control title bar and command area
   - Allow buttons on the Navigation / Title Bar- one on right-hand side for actions like Save, Done, Edit, etc.
   - Able to show/change this button and its associated action dynamically.
   - Improve and resize the action and command bars to better match Apple's 'recommended' look.
   - Overflow menus
   - New image control properties
   - Next on keyboards for forward tabbing
   - Place tabs on bottom (iOS only)
   - "Dirty (modified) form content notified.
   - Allow enter (return) in input field to submit event to server (e.g.: Search fields)

This newsletter contains a separate document (**LongRange rv11.pdf**) that outlines all new LongRange features.

2. **Complete the LongRange JavaScript API**
   - Complete and release the LongRange JavaScript API for iOS and Android. Include a new example schema and web accessible documentation – for RPG and RDMLX. This feature allows customers and service people with JavaScript skills to get maximum value from LongRange. It also increases the value that LongRange can add to HTML5 based solutions.
   - JavaScript API enhanced to allow server side programs and JavaScript code to 'communicate' by exchanging properties and signalling events (ie: requested actions).

Unicode and DBCS in LANSA V13

LANSA V13 introduced two new field types to support Unicode.

Nchar

An Nchar is a fixed-length character field, with a length between 1 and 65,535 (this is the number of characters, not the byte length).
Nchar fields store alpha data of any code page. For example, in a list, an Nchar field may have Japanese in one row, and French in another row.

Nchars are classified as Unicode strings.

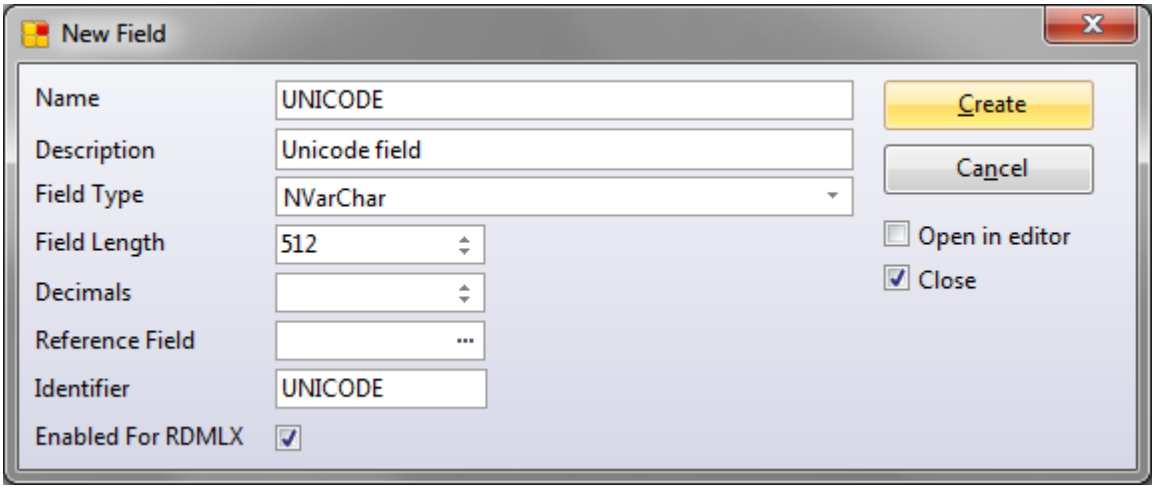
Depending on the database type, Nchar may or may not treat trailing blanks as significant. If trailing blanks are not desired, an Nvarchar field should be used.

Nvarchar

Nvarchar is a variable-length character field, with a maximum length between 1 and 65,535 (this is the number of characters, not the byte length).
Nvarchars store alpha data of any code page. For example, in a list, an Nvarchar field may have Japanese in one row, and French in another row.

Nvarchars are classified as Unicode strings.

An Nvarchar retains any trailing blanks, they are significant. When concatenating an Nvarchar with spaces on the end, those spaces are retained. But the space is NOT SIGNIFICANT for comparisons.



Before LANSA V13, where we did not have Unicode string support, alpha fields in a DBCS (Double Byte Character Set) language like Chinese or Japanese should have a minimum length of 4 characters (Shift out = 1 byte, DBCS character = 2 bytes, Shift in = 1 byte).

Unicode no longer requires Shift bytes. In general, 1 character tales 1 length value (it's not bytes any more), no matter which character it is, UNLESS it's an Uncommon Character. The characters are encoded as UTF-16. That's 16 bits per character for characters in Plane 0. This is almost all characters. For example, the only characters missing from CJK (Chinese, Japanese and Korean) are some of the less common characters in the Hong Kong Supplementary Character Set (HKSCS). For these Uncommon Characters a further 16 bits is required for each of the characters. These are called surrogate pairs.

If you are dealing with a descriptive field, the length is not usually critical as the description can be shortened if necessary. LANSA takes care of limiting data in the Field to the UTF-16 length. If a surrogate character is in the data then that restricts the number of characters to 1 less. If you try and put more data in than specified it will be truncated on a character boundary, just like an Alpha truncates on a DBCS boundary.

So the minimum length Unicode field that can contain any single Japanese character is 1.

If we use UNICODE in V13, should we still define a language as Japanese for example as a DBCS language, or is that irrelevant when we use UNICODE?

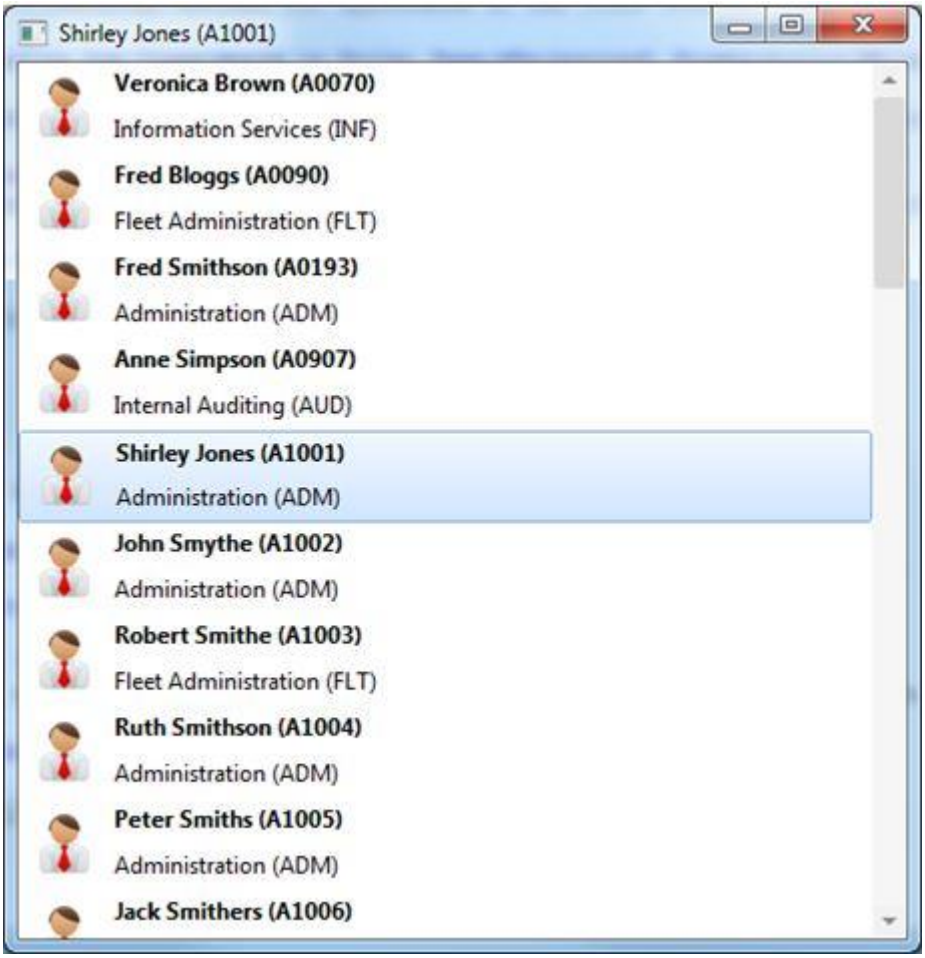
Answer

You still need a Japanese language for Repository multi-lingual data. Firstly so that the data is correctly checked in to IBM i – its stored natively on IBM i, it's not Unicode. Secondly, so that you can translate your app to another language.

Understanding the Tree User Design Control in Visual LANSA V13

The Tree User Design Control (Prim_tree) allows the developer to utilize the power of LANSA lists while producing a user interface that can go far beyond the traditional columnar list appearance.

Like all user defined controls, Tree allows the developer to create a reusable part to act as the design for each of the items. This means there are no real restrictions as to what data is used, or how it can be displayed.



Example

Pre-requisite - This example makes use of repository objects that are part of the Visual LANSA DirectX demonstration material.

For the purposes of this example, tree is being used as though it were a standard list, with no child items being created. The items created show the data on two rows, rather than side by side in two columns. This is a common pattern seen in modern user interfaces, particularly those designed with a touch screen in mind. Rather than a wide thin strip of data, a formatted tile is used to show the data in a more organised, efficient and aesthetically pleasing way.

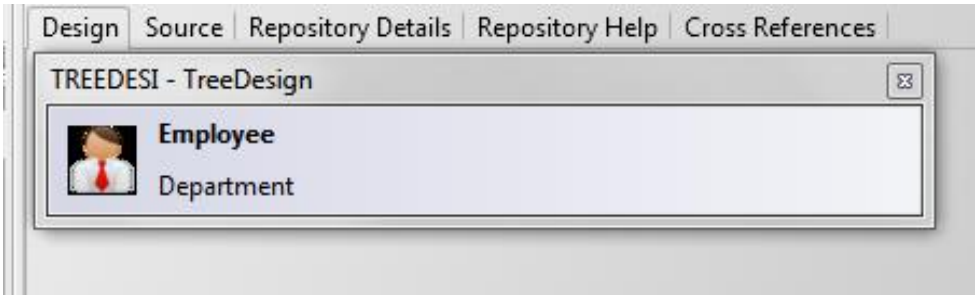
The first step is to create a design panel. This is really little more than a simple reusable part. However, to enable the tree to communicate effectively with the design instances, it is necessary for the design to implement the TreeDesign interface.

```
Begin_Com Role(*EXTENDS #Prim_panl *Implements #Prim_tree.iTreeDesign *ListFields
#ListFields)
Group_By Name(#ListFields) Fields(#Givename #Surname #Empimg #Empno)
```

This provides a series of methods that can be redefined to allow the reusable part to perform some processing when the entry is added, selected, gets focus and so on.

To make data handling easier, the *ListFields parameter of the Begin_com can be used to specify the fields that will be automatically mapped in to the design when the list entry is added or updated.

Firstly, create a reusable part called TreeDesign and copy the code below. This will act as the design for each of the items.




```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_PANL *implements #Prim_Tree.iTreeDesign *ListFields
#ListFields) Displayposition(1) Height(48) Layoutmanager(#Table) Left(0)
Tabposition(1) Top(0) Width(403)

* Fields mapped in when the entry is added to the tree
Group_By Name(#ListFields) Fields(#Givename #Surname #Empno #Deptment #DeptDesc)
Define_Com Class(#prim_vs.style) Name(#Bold) Bold(True)
Define_Com Class(#PRIM_PANL) Name(#ImagePanel) Displayposition(1) Height(48)
Image(#XDXImageEmployee32) Left(0) Parent(#COM_OWNER) Tabposition(1)
Tabstop(False) Top(0) Width(48)
Define_Com Class(#PRIM_LABL) Name(#Employee) Caption('Employee')
Displayposition(2) Ellipses(Word) Height(24) Left(48) Marginleft(2)
Parent(#COM_OWNER) Style(#Bold) Tabposition(2) Tabstop(False) Top(0)
Verticalalignment(Center) Width(355)
Define_Com Class(#PRIM_LABL) Name(#Department) Caption('Department')
Displayposition(3) Ellipses(Word) Height(24) Left(48) Marginleft(2)
Parent(#COM_OWNER) Tabposition(3) Tabstop(False) Top(24) Verticalalignment(Center)
Width(355)
Define_Com Class(#Prim_tblo) Name(#Table)
Define_Com Class(#Prim_tblo.Column) Name(#Column1) Parent(#Table) Units(Pixels)
Width(48)
```

```
Define_Com Class(#Prim_tblo.Column) Name(#Column2) Parent(#Table)
Define_Com Class(#Prim_tblo.Row) Name(#Row1) Parent(#Table)
Define_Com Class(#Prim_tblo.Row) Name(#Row2) Parent(#Table)
Define_Com Class(#Prim_tblo.item) Name(#Item1) Column(#Column1)
Manage(#ImagePanel) Parent(#Table) Row(#Row1) Rowspan(2)
Define_Com Class(#Prim_tblo.item) Name(#Item2) Column(#Column2) Manage(#Employee)
Parent(#Table) Row(#Row1)
Define_Com Class(#Prim_tblo.item) Name(#Item3) Column(#Column2)
Manage(#Department) Parent(#Table) Row(#Row2)

Mthroutine Name(OnAdd) Options(*Redefine)

#Com_owner.Cursor <= #sys_appln.Cursors<Hand>
* Update the design labels using the field values passed in
#Employee := ("&1 &2 (&3)").Substitute( #GiveName #Surname #Empno )
#Department := ("&1 (&2)").Substitute( #Deptdesc #deptment )
* Use the default application MouseOver style
#Com_owner.MouseOverStyle <= #sys_appln.Appearance.TreeMouseOver
Endroutine

Mthroutine Name(OnItemGotFocus) Options(*Redefine)
* Apply the default appliction selection style to
#com_owner.style <= #sys_appln.Appearance.TreeSelected
Endroutine

Mthroutine Name(OnItemLostFocus) Options(*Redefine)
#com_owner.style <= *Null
Endroutine
End_Com
```

Secondly, create a new form called Tree and copy the code below. This defines the tree and the design that it will use. Compile both and execute the form, ensuring that you execute as DirectX.

Once you've seen it run, execute again in debug to see how the design instances are created and how they react with focus change.

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Caption('User Designed Tree')
Clientheight(579) Clientwidth(463) Height(617) Layoutmanager(#Layout) Left(138)
Style(#Background) Top(195) Width(479)
* User Designed Control - Tree
* Individual items are made by adding entries as per typical LANSA list processing
* Fields in the list are defined by the *ListFields parameter of the Design being
made
* Prim_Tree defines the Tree
* #TreeDesign defines the appearance of the items created
Define_Com Class(#PRIM_TREE<#TreeDesign>) Name(#UDCTree) Displayposition(1)
Height(579) Left(0) Parent(#COM_OWNER) Tabposition(1) Tabstop(False) Top(0)
Width(463)
Define_Com Class(#PRIM_ATLM) Name(#Layout)
Define_Com Class(#PRIM_ATLI) Name(#ATLI_1) Attachment(Center) Manage(#UDCTree)
Parent(#Layout)
Define_Com Class(#prim_vs.Style) Name(#Background) Normbackcolor(White)

Evtroutine Handling(#Com_owner.CreateInstance)
Select Fields(#UDCTree) From_File(pslmst)
Fetch Fields(#Deptdesc) From_File(deptab) With_Key(#deptment)
* An instance of the design will be created when the entry is added.
```

```
Add_Entry To_List(#UDCTree)
Endselect
Endroutine

Evtroutine Handling(#UDCTree.ItemGotFocus)
* Update the form caption with the data from the current list item
#Com_owner.Caption := ("&1 &2 (&3)").Substitute( #GiveName #Surname #Empno )
Endroutine
End_Com
```

Using the option to Override File Library to Partition data library in Visual LANSA

V13 contains the following enhancement

CCS#: 0137308 (Enhancement)
Description: Changing the File Library name on Import
Originator: LANSA France

Detailed Description

Selection of the option "Override File Library to Partition Data Library" will import any files into the Partition Data Library associated with the current partition. This feature is particularly useful when importing LANSA files from an export created using an earlier version of LANSA which would otherwise create the file in the library nominated in the imports details column.

It has been reported that in some situations, after import, file libraries are changed to "partition data library" even if "Override File Library To Partition Data Library" is unchecked on the import.

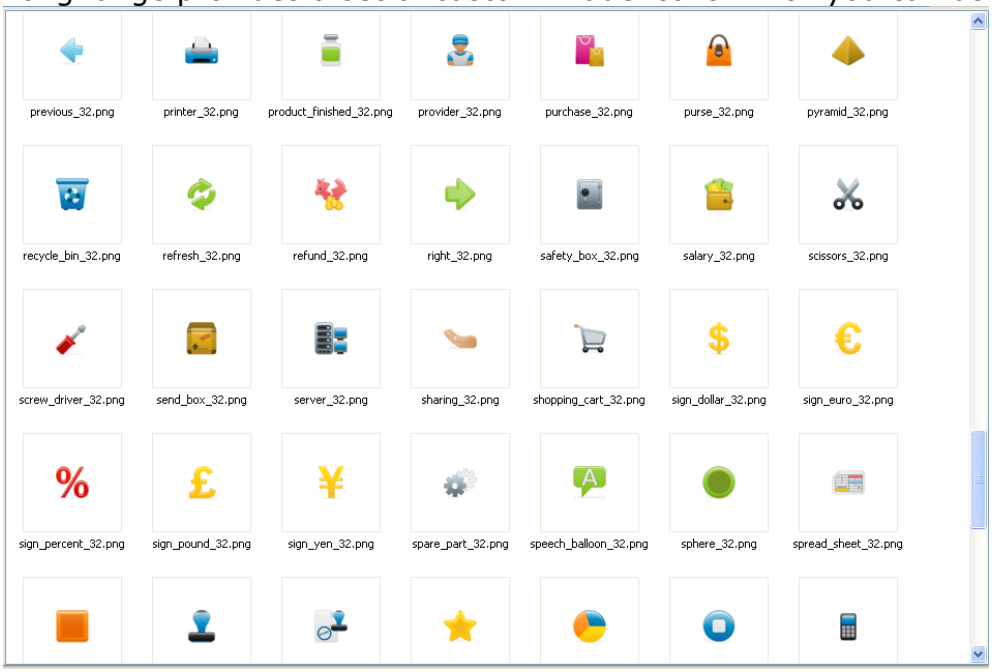
The cause is that in V13, when exporting a file from iSeries with target PC, the substitution variables are set to $$DTALIB$$, and accordingly they are imported to the partition data library when imported into VL irrespective of whether the "Override File Library to Partition Data Library" is checked or not.

The option on the import screen in VL will be enhanced to cater for the extra consideration.

Until the improved screen is available, if you wish to use the "Override File Library to Partition Data Library" to determine the library during an import, you can blank out the Target Library field for all files in option 12 Review objects on the list before exporting the list. This will be $$DTALIB$$ at the moment, and this is what is causing the file library to be overridden to the partition data library.

Icons For Your Mobile Applications

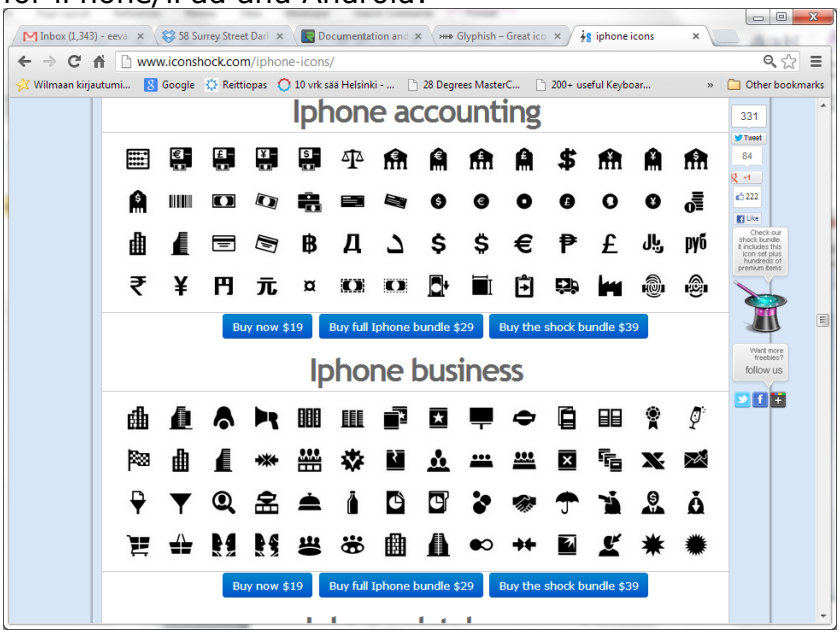
LongRange provides a set of custom-made icons which you can use in your applications:



These icons can be found in the LongRange resource folder:
LongRange RPG: LongRange\resource under your aXes root folder.
LongRange LANSA: LongRange\resource under <lansa root folder>\webserver\images.

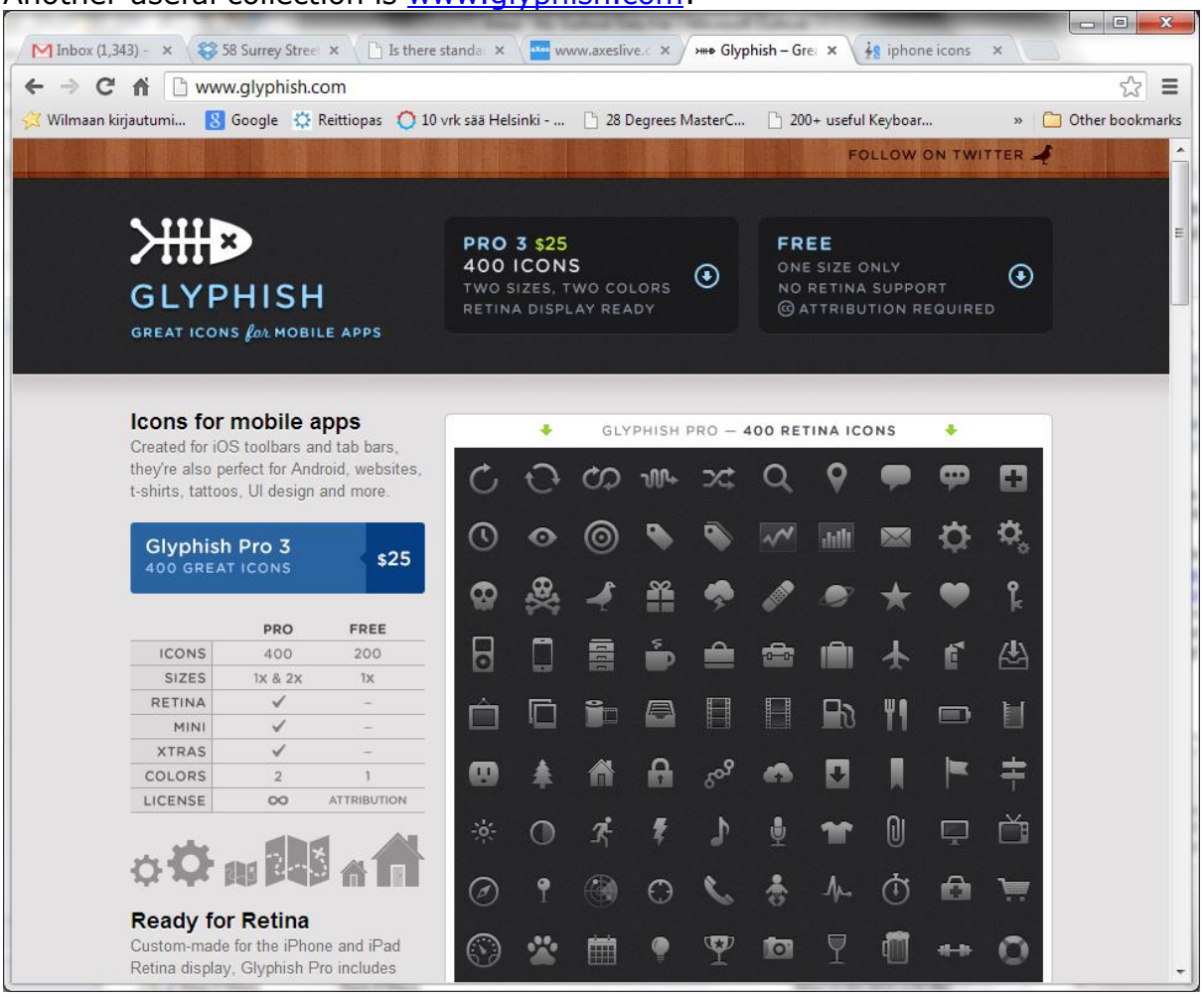
If you cannot find suitable icons in this collection, you can get icon sets on the web for very little.

For example the company which created the LongRange icons, iconshock (www.iconshock.com), sells a huge selection of stock icons and icon sets including icons for iPhone/iPad and Android:

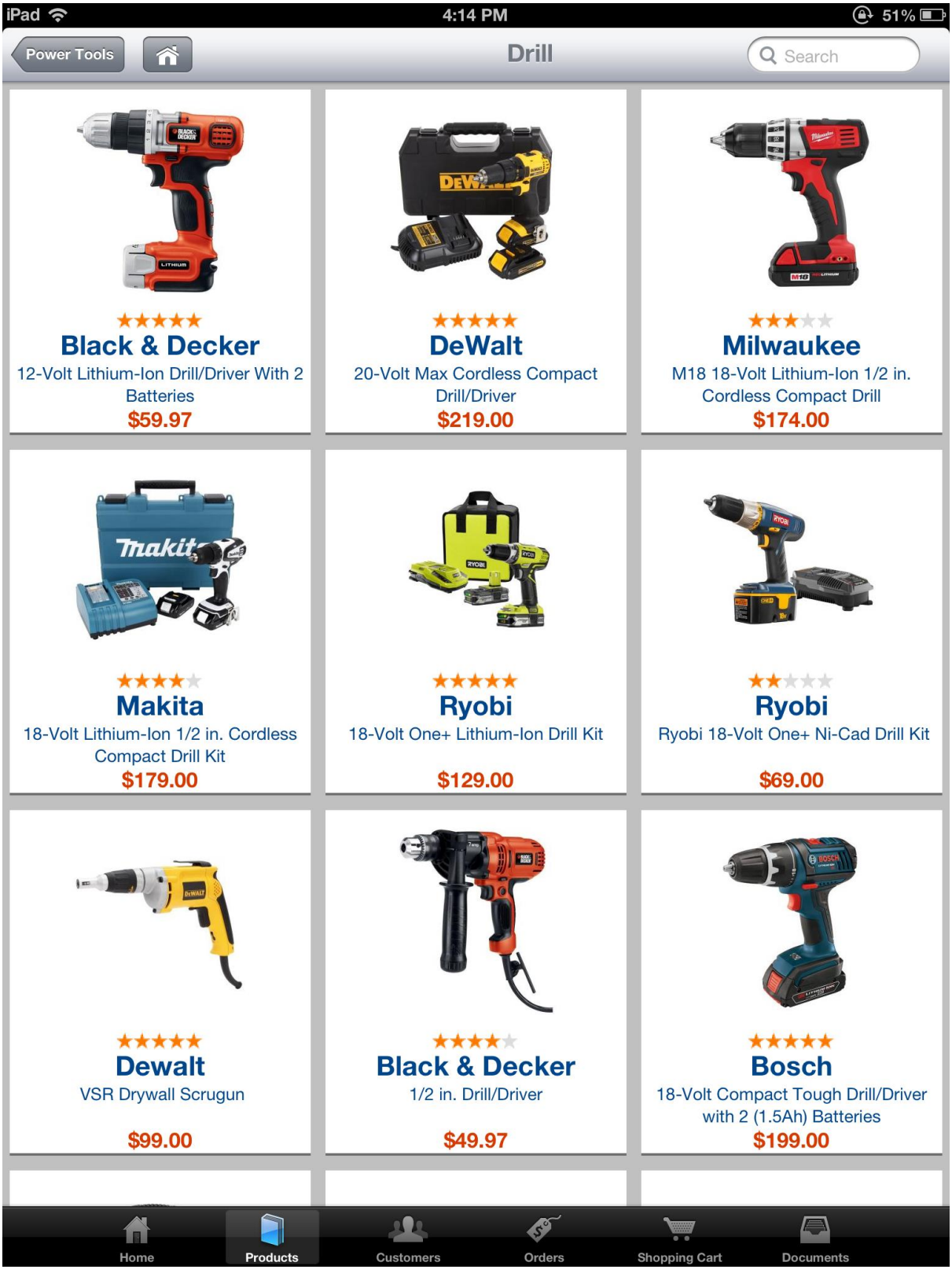


We cannot redistribute these icons for licensing reasons, but you can purchase them for use in your own applications.

Another useful collection is www.glyphish.com:



Here is an example of a LongRange application which uses the Glyphish Pro icon set in the tab bar on the bottom of the screen:



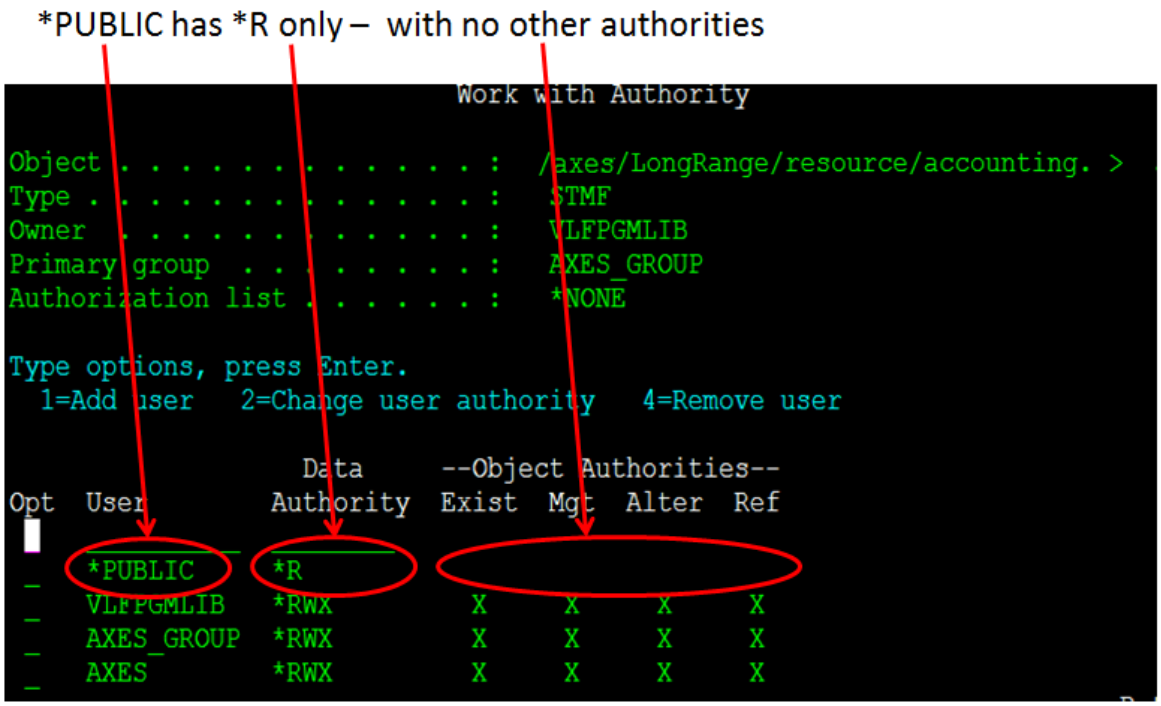
All icons must be copied to the LongRange\resource directory so that the LongRange client running on a mobile device can find them.

Note for LongRange RPG

When using LongRange for RPG you need to make sure that any new icons you add to your Resource folder have user *PUBLIC with *R rights only.

For example use command WRKLNK OBJ('/axes/LongRange/resource') then use 5=Display to display the files in the resource folder:
For each new icon file use 9=Work with authority to check that user *PUBLIC and *R rights only.

When required use option 2=Change user authority to adjust.



LongRange Runtime Version 11 (RV11)

Contents

New Home Icon replaces the word Menu and Titles are centred on IOS

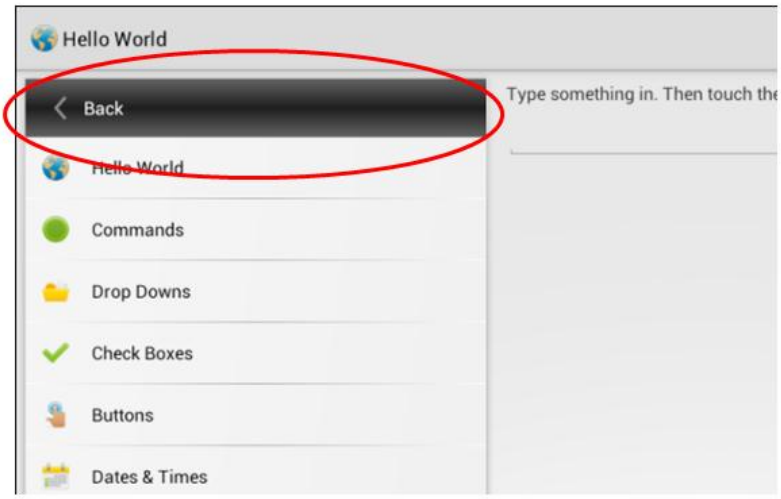
iOS old look …

iOS new look …..

Under iOS the menu can still be viewed by a single finger swipe towards to right across the current form view. On devices Android the  option for showing and hiding menus works as before.

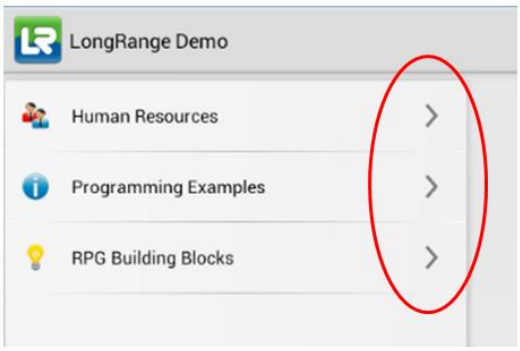
New Back option on child Android menus

The physical back button on Android devices is now used for programmatic back operations - see following section on the *Back Button Area.* Its function has been replaced on child menus by this new child menu item …



New Child Menu Option on Android Menus

The reverse of the previous point is that Android menu items that lead to child menus now indicate this like this:



Android Portrait Mode menus now fill the screen

If you display a LongRange menu in portrait mode on any Android device the menu now fills the screen. You can make the menu appear and disappear by using the usual  option.

New icons for Settings, Refresh/Reload, Close Session and Information in Android

They now look like this:

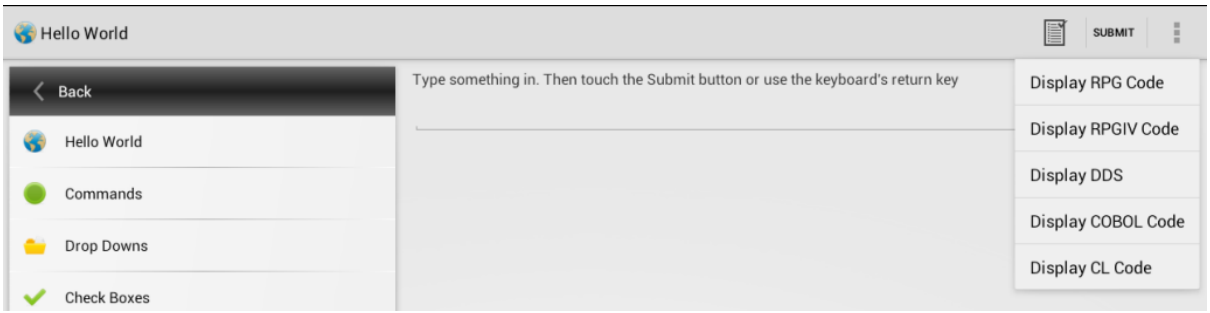
New Overflow Command Menu s

An overflow command menu is indicated on iOS by  and on Android 3.0+ by appearing on the right hand edge of the title bar. When touched an overflow menu of commands appears.

For example - this is the old way to see RPG code …

This is the new iOS way …

Or the new Android way ….



You can add your own commands to the overflow menu. More on information follows.

New Image Properties

The Image element has new properties…

VAlign	Specifies vertical alignment of image inside the element. Requires client runtime version 11 or later

Possible Values

Value	Description
Top	Top alignment
Middle	Middle alignment
Bottom	Bottom alignment

`HAlign`

Specifies horizontal alignment of the image inside the element. Requires client runtime version 11 or later

Possible Values

Value	Description
Left	Left alignment
Center	Center alignment
Right	Right alignment

`AutoScale`

Specifies how the image inside the element should be scaled when the image size is different from the element size. Requires client runtime version 11 or later.

Possible Values

Value	Description
No	Image will not be scaled
Fill	Image will be scaled to fill the whole element in both X and Y direction. Aspect ratio of the image will be ignored so it might appear distorted.
Aspect-fill	Image will be scaled to fill the whole element in both X and Y direction. Aspect ratio of the image will be maintained. As the result, the image might be clipped either horizontally or vertically.
Aspect-fit	Image will be scaled to fill the element, **either** in X or Y direction, maintaining the image aspect ratio. Unless the image has exactly the same aspect ratio as the element, some empty space will be present.

Next appears on Keyboards

iOS keyboards now display a Next button that allows the user to move directly from one input field to the next without having to dismiss the keyboard or touch the next input field.

Note: Android devices always had tab keys and/or next keys.

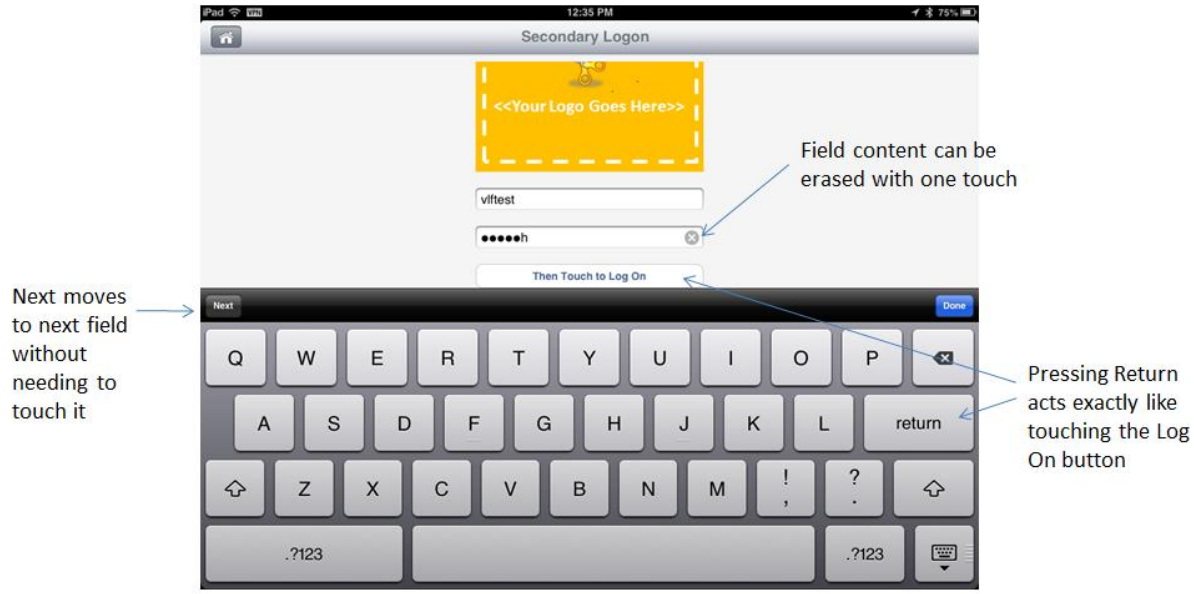
New Textbox options

Text boxes have these new properties...

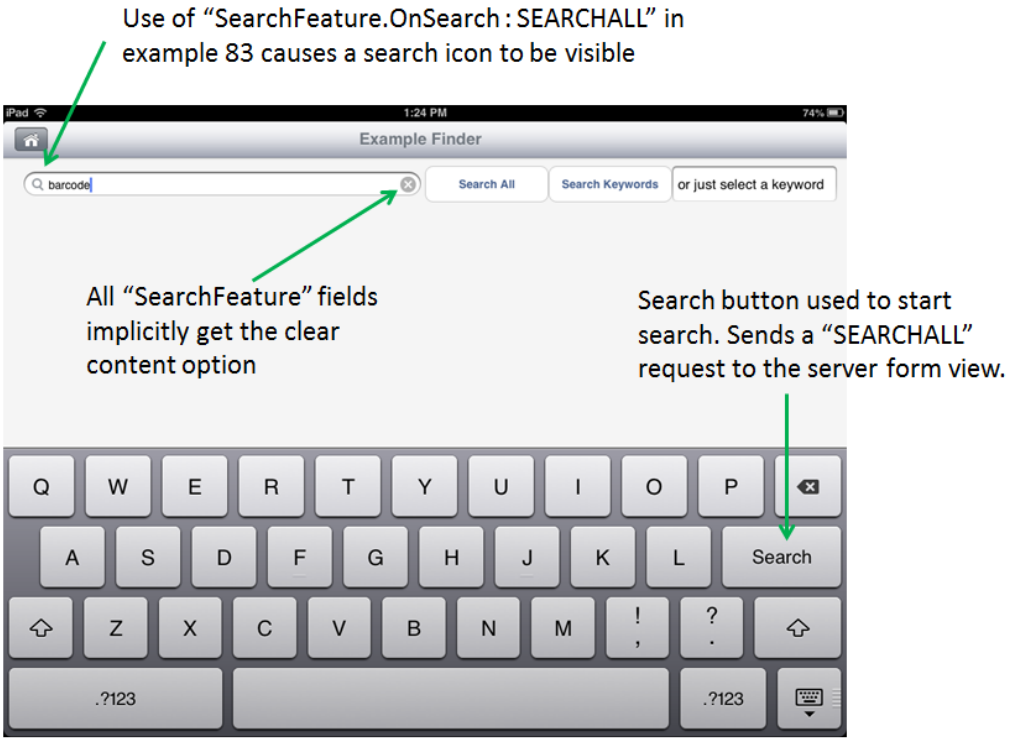
`ShowClearButton`	A Boolean value indicating that a small image that clears the textbox content when touched. Requires client runtime version 11 or later. The clear content image is only visible when the textbox has focus and content. Not applicable to multiline textboxes.
`OnReturnKey` EVENT	An event that occurs when the return key is used within the textbox. Requires client runtime version 11 or later.
`SearchFeature.OnSearch` EVENT	Indicates that small search image should be displayed in the field and the specified event fired when the user touches the 'Search' button on the keyboard. Usually this facility is used only with search fields. Using this property unconditionally sets ShowClearButton to true. The displayed textbox has corners that are more rounded than a normal textbox. Requires client runtime version 11 or later.

The Search feature is used in the shipped Example Finder. However the main Incident, Employee and the Seach and Select RPG Build Block use the more advanced Search command option (more on that later).

The impact of ShowClearButton and OnReturnKey working together can be seen in the improvement to the shipped RPG user logon Building Block ….



Also shipped example 83 (the Example Finder) uses the SearchFeature.OnSearch option ….



Note: This new Search Feature option should not be confused with the new Search Area Command option – more details on that follow.

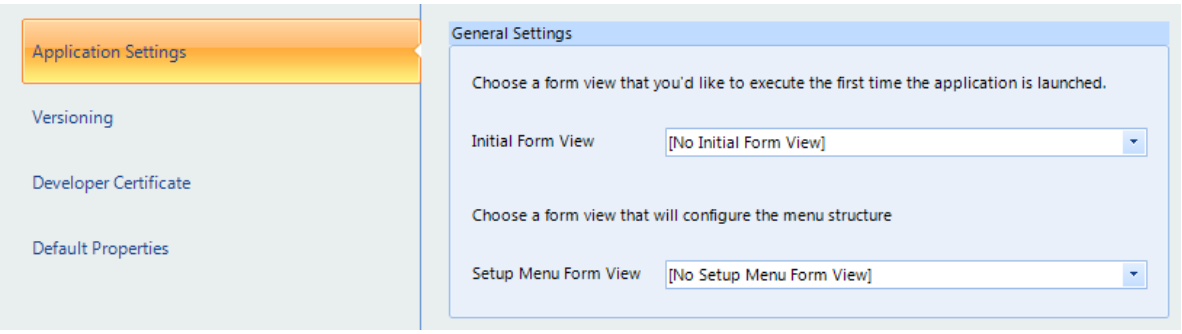
New Device Information

The Mobile Device Details example has been updated to show all available client device information.

The major new items are the **orientation** and the **language**.

New Initial Menu Form View

In LongRange Studio new start up options may be defined ….

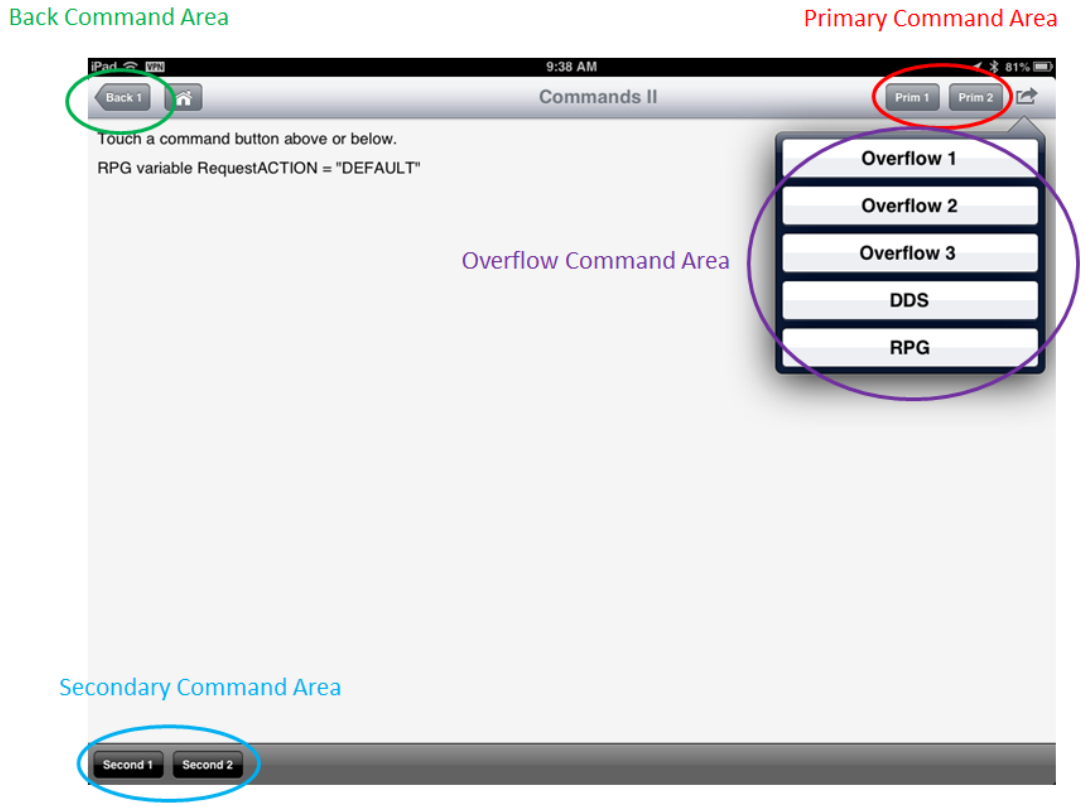


Refer to the documentation for more details.

New Primary, Secondary, Back and Overflow commands

Commands used to be positioned like this….

Now they may be positioned like this ….

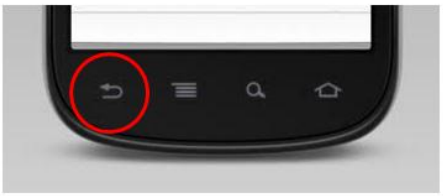


See the new introductory RPG example ***Commands II*** (EXAM0088) in ***Introductory Examples.***

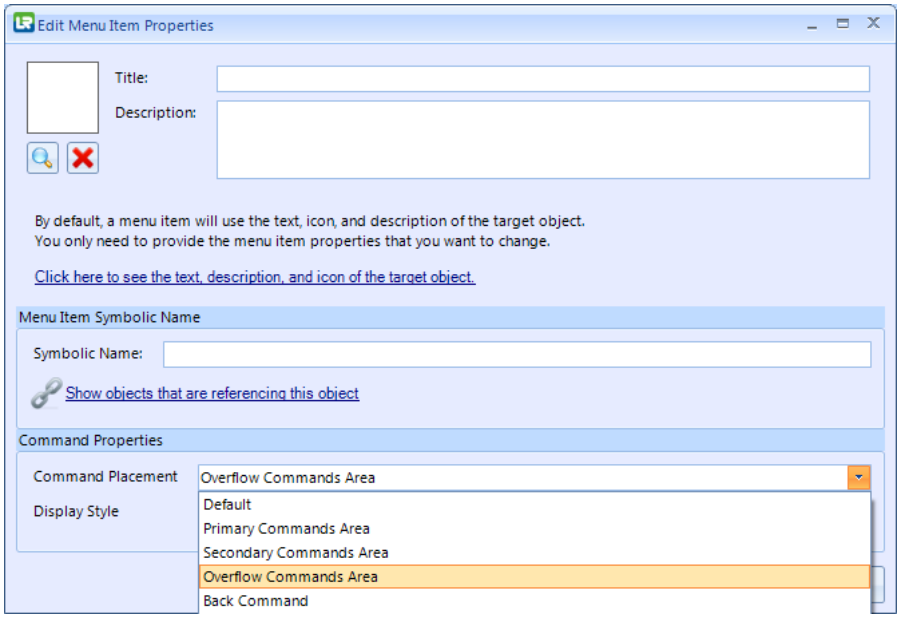
You can only have one back command visible at any point in time.

On Android devices the Back button is not displayed.

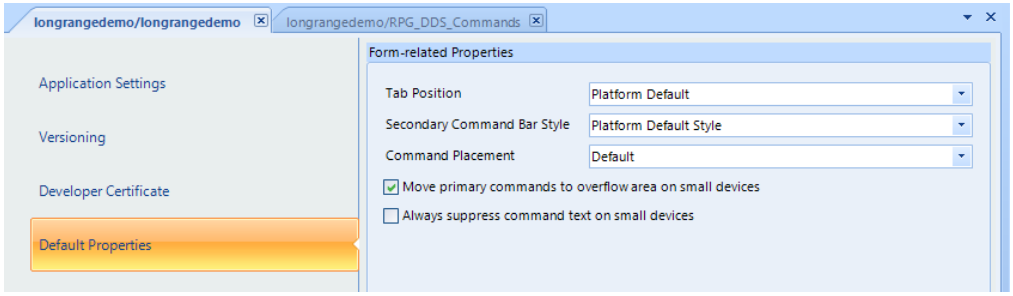
Android users use the standard hardware or software back button …



The new command locations are individually controlled for each menu item by new LongRange Studio options …..



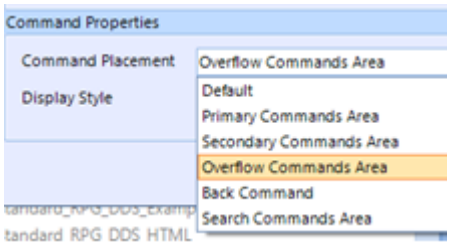
Additionally some default command behaviours can be defined at the schema level….



For commands on the tool bars you can also indicate whether the text should appear, the associated icon, or an automatic switch from text to icons on small screen devices. A very useful option is the automatic movement of primary commands into the overflow menu on small screen devices.

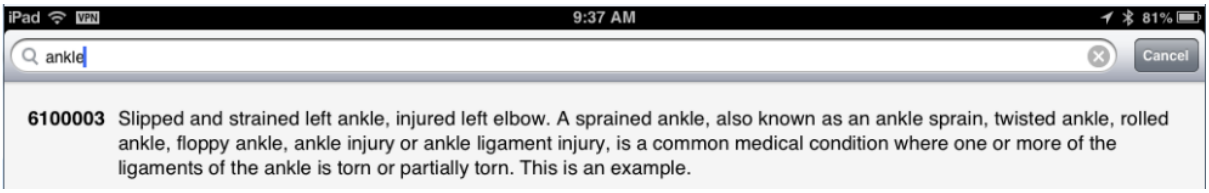
New Search Command Area

The other command area is the ‘Search Command Area’

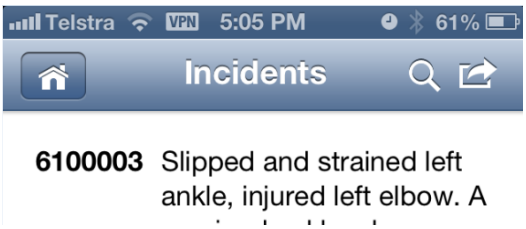


This indicates the command should appear like it does in the new shipped Incidents demo…

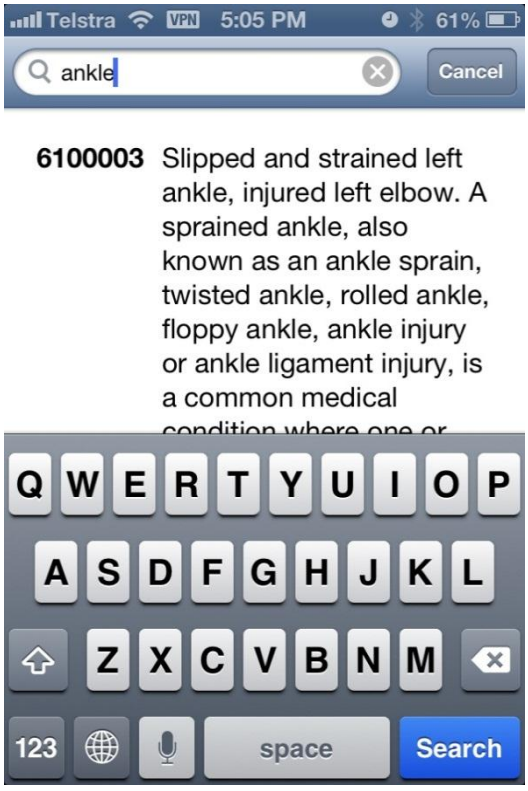
When the search command receives focus it expands to allow easy data entry ….



On a small screen device these two screens appear as …..



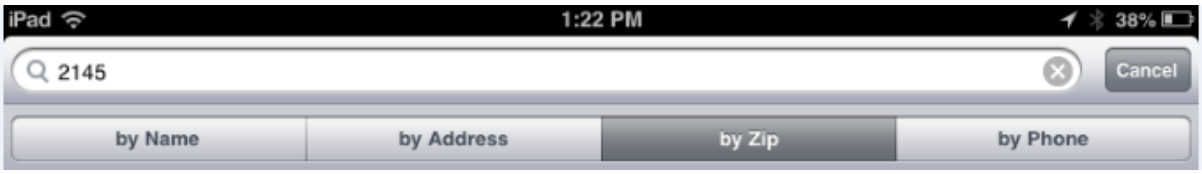
and



Note how the search is initiated by the search button on the keyboard.

You may have multiple Search commands.

Multiple commands appear like this:

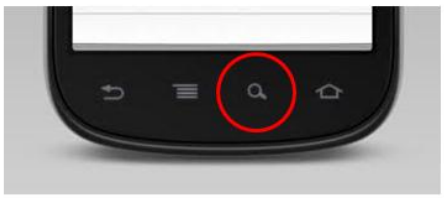


Or

Refer to the new EXAM0090 ***Searching*** found on the ***Use Case Examples*** menu for an example.

Also refer to the Incidents demonstration application and the RPG Building Blocks.

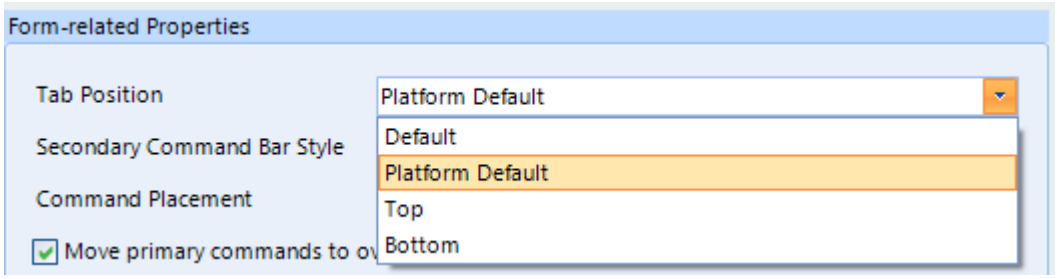
In the older Android versions (pre-Honeycomb) a search button is used instead of the search icon to initiate searches ….



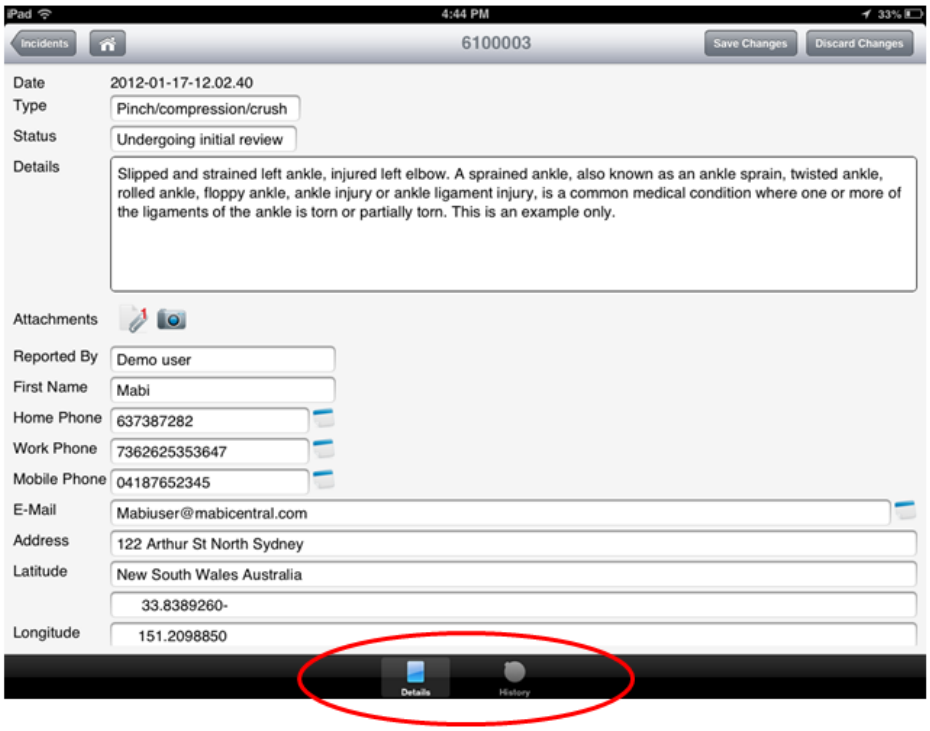
Some pre-Honeycomb devices do not have a search button. In this case the Android standard is to use longer hold of the menu button to indicate you wish to search.

New tabs placement options

There are new tab placement options definable in LongRange Studio ….



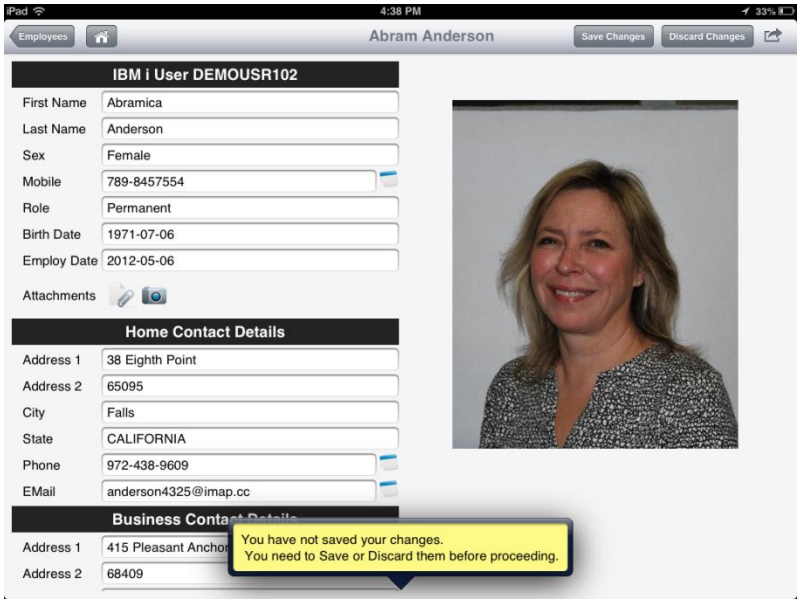
Under iOS you can now position tabs at the bottom of the screen and change their styling…

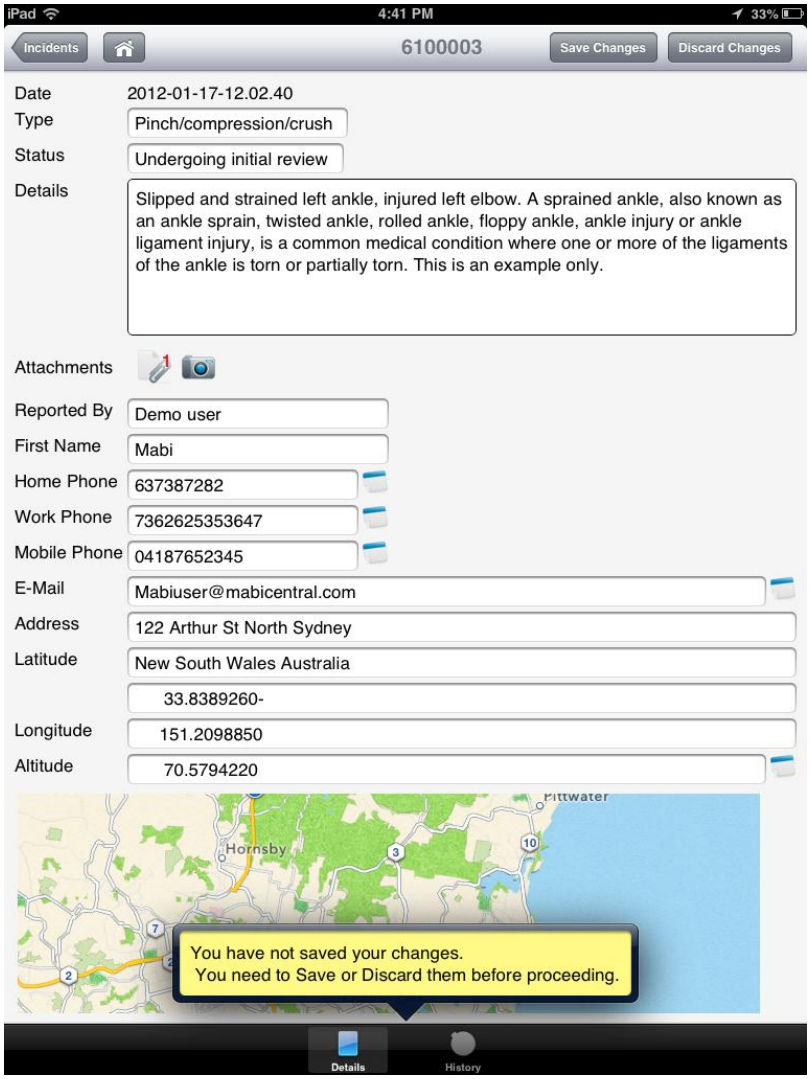


Detection of changes to a form's content

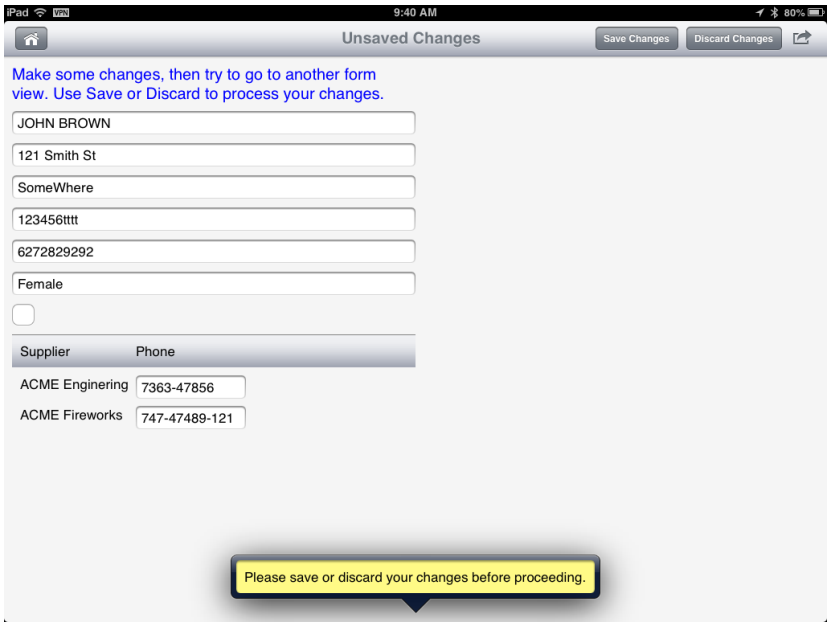
You can detect if the user has changed something on a form.

The shipped Incident Details and Employee Details examples both have this logic built in now ….





A specific advanced example named “Unsaved Changes” is also shipped to demonstrate the simplest way to incorporate such logic …..



Attached Notes and Photos have changed in the shipped demonstration

Instead of a button they now appear as a paperclip image and a camera with a count of the associated attachments overlaid ….

← Has 1 note and no photos attached

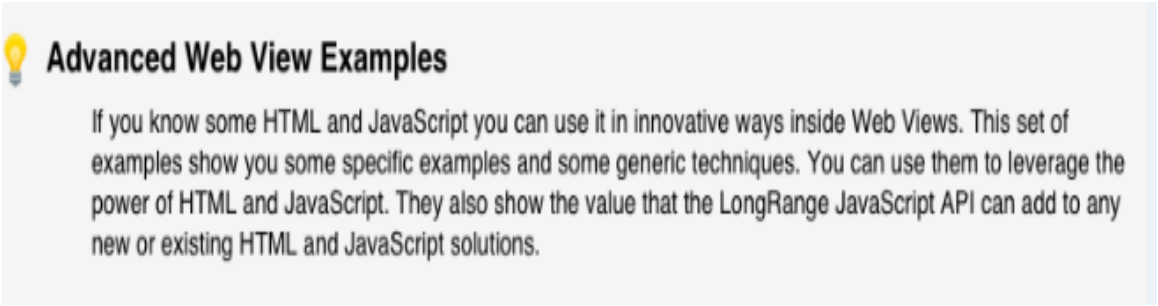
← Has 5 notes and 1 photo attached

New and Improved RPG Building Blocks (shipped in library LRNG_PROJ)

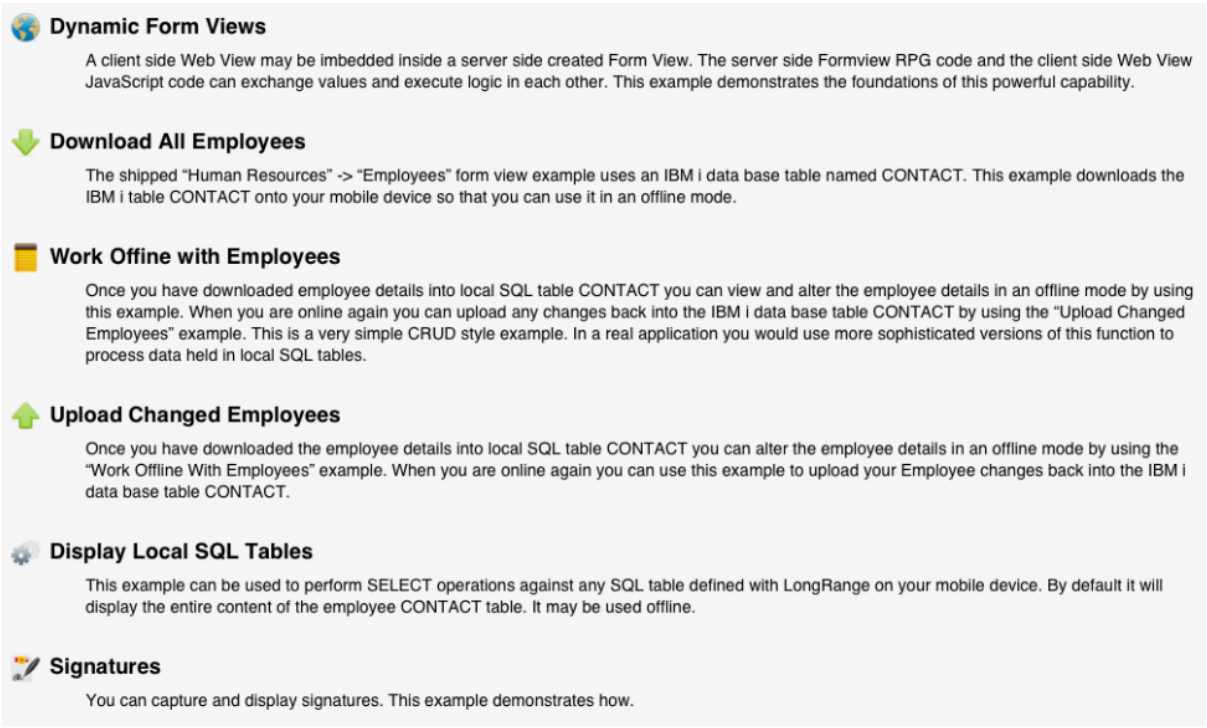
There are several new building blocks worth investigating. Existing building blocks have been upgraded to use new features like the search command area and tabs at the bottom …

Introducing the LongRange JavaScript API - for those that know JavaScript

A new LongRange JavaScript API is now available for iOS and for Android. Why you might use it is expressed by the new Advanced Web View examples…



This option displays this new menu of shipped examples…



Please note ….

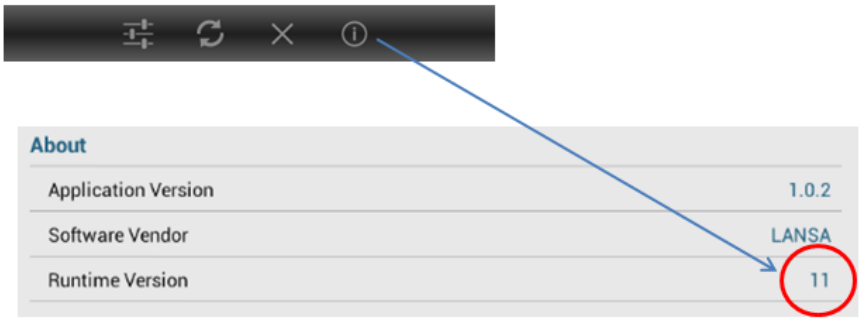
- The offline Employees examples require a LongReach server - like the Document View examples.
- The Dynamic Form View and Signature examples do not require LongReach.
- The Dynamic Form Views example is worth looking at if you know some HTML and JavaScript. It shows an RPG program and a JavaScript program exchanging information – a foundational facility for integrating RPG and JavaScript solutions.
- The Signatures example is an initial solution. It will be largely superseded by an extension to the image element that allows hand drawn images to be captured individually (eg: a signature) or as annotations to an existing image (eg: arrows, lines and dimensions added to a photo).
- The advanced guide *Programming LongRange with JavaScript* is an adjunct to the RPG and LANSA Programming Guides. The guide and a collection of extended example scripts and an example schema may be obtained from your LongRange vendor.

What do you need to use RV11?

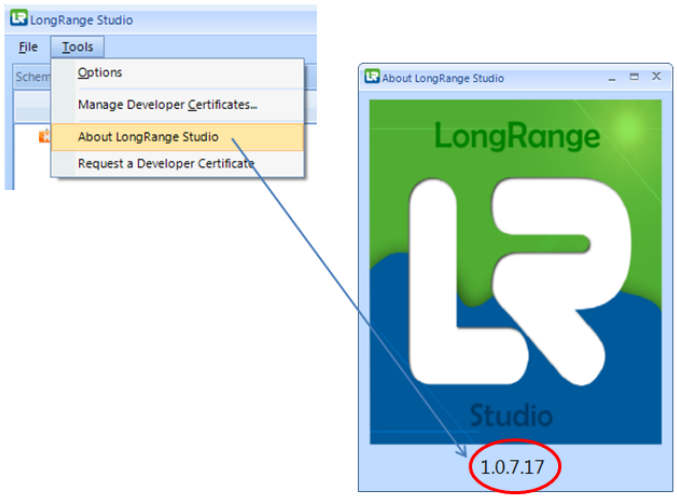
To use the following new features you will need:

- A new iOS client installed on your iPad, iPhone or devices. To confirm this version touch the information icon and check that Runtime version shows 11 (or greater)…

- A new Android client installed on your Android devices. To confirm this version touch the information icon and check that Runtime version shows 11 (or greater)…

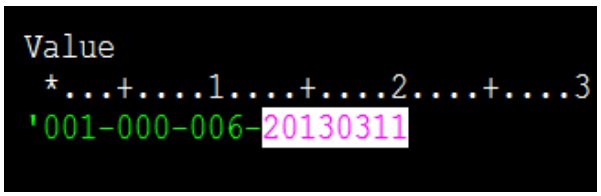


- A new version of LongRange Studio on your development PCs. You do not need to relicense. To confirm this version touch the About Longrange Studio option and check that version shows 1.0.7.17 (or greater)…



- If you program in (RDMLX) you must install a new version of LANSA for LongRange on your IBM I or Windows server.

- If your program in **RPG** or **COBOL** you should install a new version of LANSA for LongRange on your IBM I server. To confirm this version execute the command **DSPDTAARA DTAARA(LRNG_SOFT/VERSION)**. The highlighted portion of the resulting display is the version date in format YYYYMMDD. The date displayed must be on or after 11th March 2013 (20130311):



Visual LANSA v13.0

Adopting DirectX

February 2013

Adopting DirectX

Introduction

Visual LANSA version 13.0 introduces the DirectX rendering engine. DirectX is a collection of APIs embedded in to Windows that provide superior graphics capabilities.

Traditionally, the appearance of Windows applications was defined by relatively simple prefabricated controls such as TreeView, Push button and Grid. Whilst these controls provided the required functionality, and with use of Visual LANSA Themes and Visual Styles could produce aesthetically appealing applications, user interface styling in terms of both the appearance and operation, was fundamentally limited by their prefabricated nature.

However, the emergence of the internet and the typically more marketing-like appearance of websites, and latterly the rush towards touch devices and mobile computing, has helped fuel a change in developer behavior. The aesthetic of applications has become far more important, with the result that for some application requirements, predefined control appearance is no longer sufficient to allow developers to obtain the required results.

With DirectX rendering, these capabilities are now available. Visual LANSA v13.0 introduces a selection of new features including user designed controls, dynamic styles, mouse events, , transparency, gradient colors, animations, popup panels, and a few more besides.

However, when Visual LANSA was first released, these features weren't even dreamed of, and their inclusion in version 13 has caused some minor issues that all developers need to be aware of before they decide to adopt DirectX for their existing applications.

First and foremost, an existing application running in version 13 will continue to behave exactly as it did in version 12. Only by actively choosing to use the DirectX rendering option will the application change. This can be done at application, form or even panel level. For many users, this transition may be almost seamless. The application may look slightly different, but in terms of functionality, the behavior may well be identical. For some however, DirectX rendering may subtly impact the behavior of the existing application.

LANSA has gone to great lengths to ensure that "flicking the DirectX switch" is as simple and uneventful as possible, and that the user interface remains close to that of version 12. However, with such an array of new functionality and the restrictions imposed by the adoption of new underlying technologies, some change is inevitable. This document outlines some of the changes made to Visual LANSA. It explains how these may impact your existing applications and endeavors to provide simple changes that can be made to resolve any issues found.

Adopting DirectX Rendering

Whilst every effort has been made to ensure that as much of the Win32 appearance has been honored as possible, circumstances dictate that it is simply impossible to provide the new features and a DirectX runtime that precisely reflects that of Win32.

Before enabling for DirectX, consider the following. Your applications will continue to run in version 13 as they did in version 12. While DirectX offers a far greater flexibility of UI and many new features, it also comes at a cost. Some of the changes will affect the behavior of your application and may well require code review and modification, testing and all of the other typical tasks associated with software development. In short, moving an existing application to DirectX is not something that should be undertaken lightly.

While LANSA strongly recommends DirectX rendering for all new applications, as all future UI enhancements will be targeted at DirectX, moving an existing Win32 application to DirectX should be considered on case by case basis.

Enabling for DirectX

Visual LANSA allows DirectX to be applied at panel, form or application level. Once DirectX is enabled, all child panels will also use DirectX, specific Win32 controls notwithstanding e.g. ActiveX & graphs.

If you set the runtime to DirectX, all forms, panels and controls within the application will use DirectX rendering.

If you set a form to use DirectX, child panels and controls will use DirectX rendering.

If you set a panel to use DirectX, child controls will use DirectX rendering.

This simple act of setting a property or two belies the underlying complexity. The reality is that Win32 and DirectX don't really work that well together and you should seriously consider using one or the other.

Adoption Strategies

In practical terms, there are two strategies for the adoption of DirectX – Wholesale or piecemeal.

Piecemeal

This is the seemingly simpler strategy and offers a gentler introduction to DirectX. It allows individual forms or panels to start using DirectX related features e.g. new controls or styles, without affecting the remainder of the application.

However, Win32 and DirectX are very different technologies and while LANSA has greatly simplified their merging, the reality is that this solution is far from perfect. Developers may well find that integrating the old and the new does not come without some significant complications and concessions when it comes to the appearance of the application. For example, DirectX mandates the use of TrueType fonts and it may therefore be necessary to change the font for the remainder of the application to ensure consistency.

In practical terms, while simple additions can be made, there are numerous factors, mostly appearance based, that may mean that a better approach is simply to take the hit and use a wholesale approach.

Wholesale

The wholesale approach means that the runtime is switched to use DirectX and that as a result the entire application will render using DirectX.

Whilst this is certainly the more drastic of the two approaches, it may well turn out this it is also the cheaper of the two in the longer term. The initial impact is certainly larger and there will be issues to resolve, but once done all further work is in DirectX. Whereas, by adopting the piecemeal approach, all further development will need to manage the two different underlying technologies and that must lead to increased development costs.

Test, Test, Test

Regardless of the strategy employed to start using DirectX, the majority of applications may well behave slightly differently. For some it will be as subtle as a change in font and text not fitting as it did before; for others, parts of the application will not behave quite the same, e.g. mouse over events on a list causing changes to field values, and this may cause runtime errors.

As stated previously, LANSA has gone to great lengths to ensure that “flicking the DirectX switch” is as simple and uneventful as possible. However, LANSA cannot guarantee that applications will continue to work as before and it is strongly recommended that you perform a full test of your applications before enabling any productions systems.

DirectX Changes

The following sections detail many of the individual changes and explain the reasons for the change and how they may impact existing applications. Where possible, workarounds and simple ways in which these issues can be overcome are specified.

Remember, these situations can only occur if you specifically choose to run with DirectX rendering.

ComponentVersion

Software bugs are inevitable, but for the most part they can be worked around and fixed in the longer term. However, some bugs are subtle and aren't immediately recognised as being bugs. The behaviour seems reasonable enough and the user simply uses the product as they find it.

However, this user reliance on a what is really a flawed product creates a problem when looking to fix bugs and introduce new features that may not work well with the existing software. We cannot simply fix a control or start making things work as they should for the simple reason that existing applications could well look and behave differently after an upgrade.

To allow such changes to be affected, the ComponentVersion property is used. When it is no longer feasible to fit the old in with the new, a new ComponentVersion is made. Existing application code will still use the default version 0, while any new controls dropped on a form will automatically be given the latest version number.

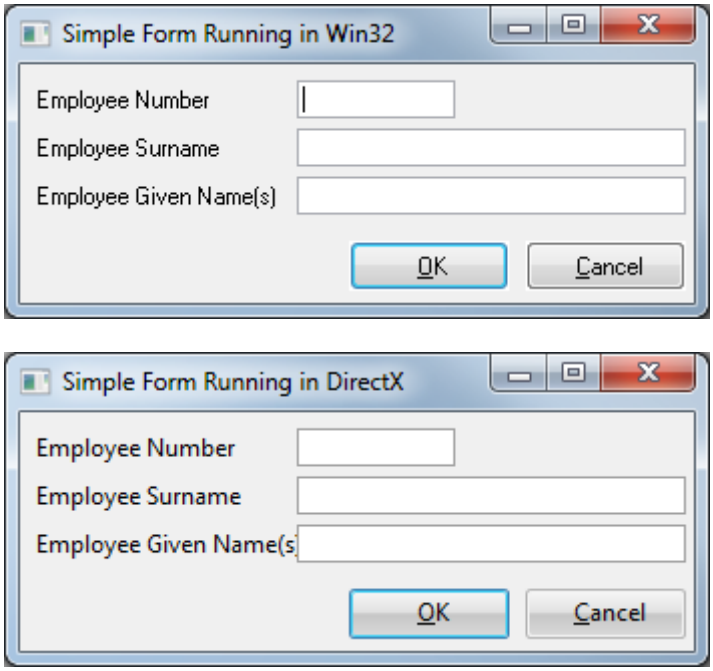
```
Define_Com Class(#prim_trvw) Name(#Tree) ComponentVersion(2)
```

Most controls are still using ComponentVersion(0). However, the controls listed below have later versions. To get the most out of DirectX it is recommended that you use the latest ComponentVersion for these controls.

Control	Primitive	Version
Combo Box	Prim_CMBX	1
Field	Prim_EVEF	1
Grid	Prim_GRID	1
Form	Prim_FORM	1
List View	Prim_LTVW	2
List Box	Prim_LTBX	1
Tab	Prim_TAB	1
Toolbar Button	Prim_SPBN	1

Default Appearance

Below are images of the same form running firstly as it would appear in version 13 using Win32 and secondly as DirectX.



The code for this form is available in the Sample Source section of this document.

Functionally, the two forms are identical. However, the font used for the text is different. Win32 defaults to MS Sans Serif 9 while Direct X defaults to Segoe UI 9.

Ms Sans Serif is an old font and was created a long time ago when screens were much smaller and had much lower resolutions. The result is that on a modern screen, running a modern resolution, it looks rather "blocky" compared to the smooth edges and nicely rounded corners of a modern True Type font using antialiasing.

For most users the change of font may well be of no consequence. However, Segoe UI is slightly wider, and as can be seen in the two images, and this may cause some text to wrap, show ellipses or be truncated.

Blending Win32 & DirectX

There are compatibility issues when trying to work with both Win32 and DirectX in the same UI space. As with fonts, this is a reflection of the underlying technology. The simple explanation is that when using the two technologies, there are essentially two different UI streams that are both unaware of each other. A more detailed explanation is available at http://msdn.microsoft.com/en-us/library/aa970688.aspx.

Those customers who take a piecemeal approach to the adoption of DirectX and perhaps use it for additional functionality such as new dialogs or new panels in existing forms etc., will have

to deal with two different technologies working together. At a low level, blending DirectX and Win32 is far from simple, and while DirectX can exist inside Win32, and Win32 can exist inside DirectX, but both have issues that need to be overcome, and this can result in behaviour that is neither expected nor desirable.

Clipping

Clipping is the term used to describe how the edges of child controls that extend beyond their parent control are hidden.

Win32 controls cannot occupy part of the same render level as DirectX and are therefore placed on a different level. This results in a situation where child Win32 controls that are bigger than their parent will cause scrolling issues. In the image below, the browser is parented to a panel which has been scrolled. See how the top of the browser coincides with the top of the panel scroll bar on the right. As this is a completely Win32 application, the browser is correctly clipped as there is only one UI stream.

The code for this form is available in the Sample Source section of this document.

However, in the same form running as DirectX (below), scrolling the panel causes the ActiveX to move, but not to clip, because there are now 2 UI streams. See how the browser is now above the panel scrollbar on the right.

The only practical solution to this issue is to ensure that the Win32 control is sized appropriately, probably by use of a layout manager, and does not exceed the size of its parent.

ActiveX

ActiveX deserves a special mention. ActiveX controls by definition are Win32. They must therefore exist in a different UI stream and they will never be able to follow the styles and appearance used in a DirectX application.

Historically, it has been a fairly common practice to embed controls such as a browser or a PDF viewer within applications to show web pages or documents. Whilst this is still possible, it is recommended that an alternative approach be taken. Rather than using the control within the application, make a system call using the System_Command BIF and let the operating system work out what best to do with the specified object.

The alternative to this is to use an equivalent .Net Component for the same task. However, it's worthy of note that .Net objects are often just the same ActiveX controls with a new wrapper, and are in practical terms, little or no different to the original.

DisplayPosition

In a scenario where a Win32 control is on top of another, the disconnection of the two UI streams mean that it is no longer possible to use DisplayPosition to bring one to the front. As far as VL is concerned, the DisplayPosition is correct, but like clipping, the Win32 control is unaware of it surroundings and simply displays as defined.

The simplest is not to rely on DisplayPosition, but instead set all but the uppermost control to Visible(False). This has an additional functional benefit in that it stops a Tab going to the "hidden" panels.

Screen Design

The layering of Win32 controls also affects the IDE. This is a DirectX application as well, so the rendering of Win32 controls in the designer can be problematic.

The Designer will position the control correctly. However, because it is floating above its parent and is not part of the UI in the same way that DirectX controls are, a Win32 control can obscure the grabbers making it impossible to resize the control with the mouse.

A simple work around for this is to briefly turn the designer to Win32 mode.

As at the time of release of this document, Memo (Prim_memo), Graph (Prim_grph) and Property Sheet are still Win32 controls.

Themes and Visual Styles

When running as Win32, Visual LANSA plays lots of games under the covers to ensure that the styles and appearance of controls conform to the specified themes. A good example of this is the appearance of panels. A panel by default is gray, but when running in a themed application it can appear blue or as a toolbar with a gradient color. Any panels parented to that panel automatically adopt the toolbar appearance.

However, should the attached panel be flagged as DirectX, it is no longer subject to the same strict implementation of themes. The runtime does its best to pick up a color from the theme, but it is at best a guess. The reason for this is that the application has crossed over from Win32 to DirectX and the two parts of the application, while appearing as though they are one, are actually separate streams.

This presents an issue when adopting DirectX piecemeal as panels on top of toolbar themed panels simply cannot display the toolbar effect.

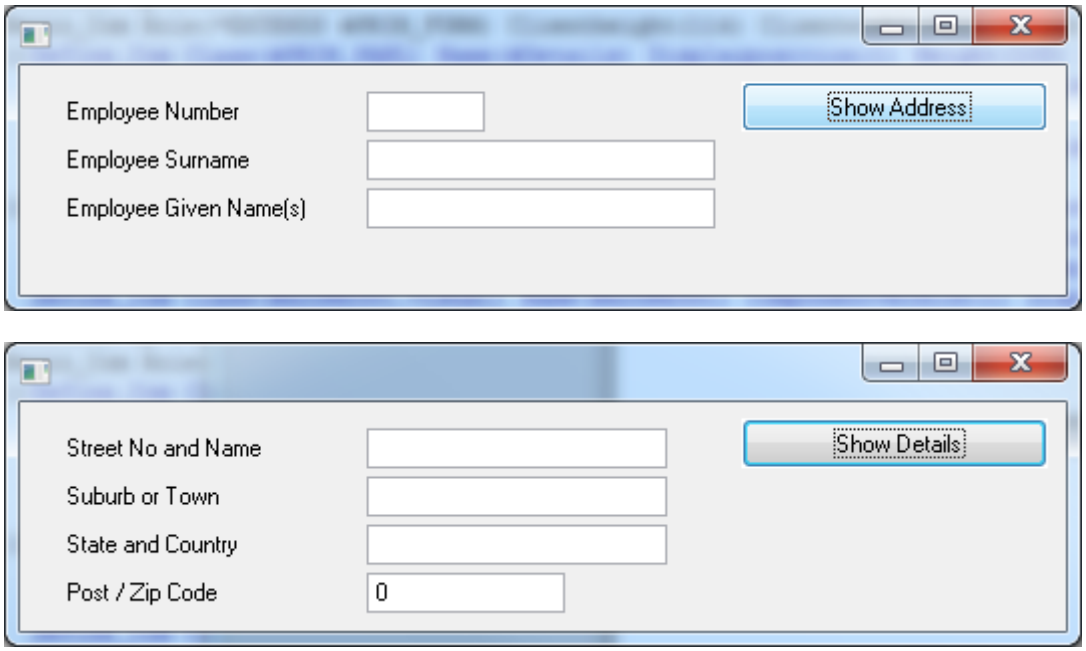
Transparency and Opacity

Transparency and opacity work seamlessly in DirectX, but cannot be employed to see through a DirectX panel to an underlying Win32 environment. As with styles and themes, the DirectX panel is wholly unaware of the Win32 controls that might be beneath it and will appear black.

Transparency and Opacity

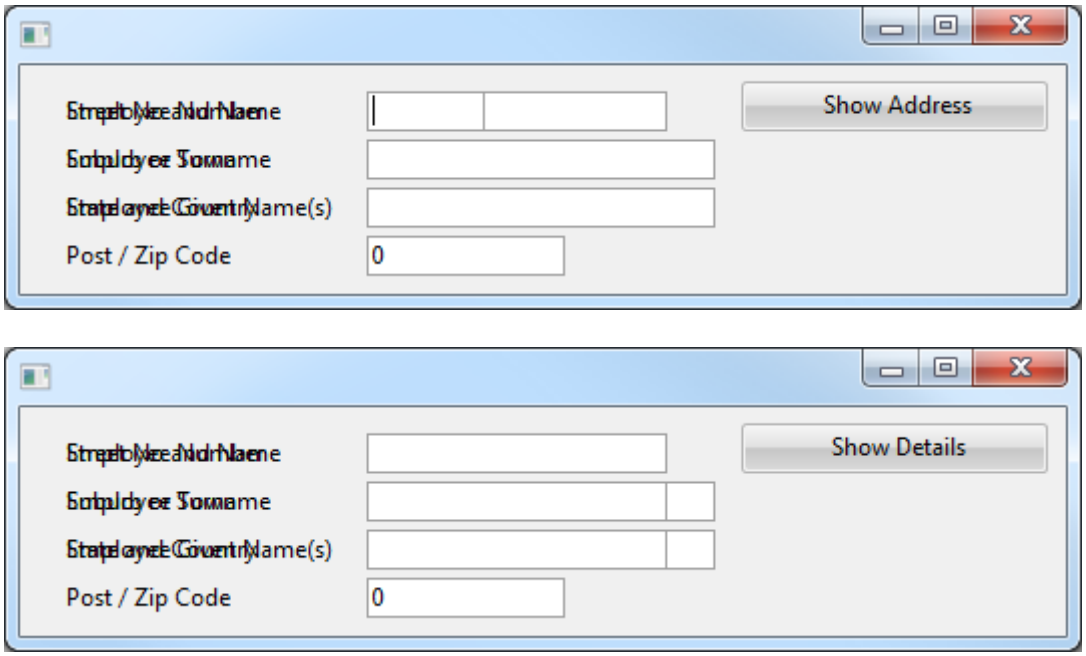
DirectX rendering introduces transparency and opacity. By default in DirectX all panels and labels are considered transparent unless a specific Style has been applied. This new appearance can lead to issues.

Below, a simple form toggles between address and employee details. When the button is clicked, the address details are enabled and brought to the front.



The code for this form is available in the Sample Source section of this document.

However, the results with DirectX rendering are somewhat different.



Regardless of the DisplayPosition of the Address and Details panels, both are plainly visible.

The need for this default stems from the desire to build complex layered forms and to still be able to see watermark images or backgrounds applied to it. If they were opaque, it would be necessary to visit every panel and label and specifically apply a transparent style.

A simple work around for this situation is to set the inactive panel to Visible(False) rather than Enabled(False).

Routed or Bubbled Events

To simplify the coding of complex reusable parts and in particular the design of panels for the new User Designed Controls (UDC), an event detected on a control is now passed up the parent chain.

Typically for UDC, the panels displayed are constructed of little more than labels and images. However, with the existing event processing, each of the labels would take the click event and not pass it on. The result would be that the user would need to code every click event for every child control. By sending the event up the parent chain, coding is greatly simplified.

Of course, it may be necessary to know which of the controls actually fired the event initially. The EVTROUTINE command already has the Com_Sender selector, but this only ever reports the control firing the event in the context of the controls referenced on the EVTRoutine command. As a result, the Origin selector has been added. Regardless of how many layers of parent are used, Origin will contain a reference to the instance on which the event was actually started.

```
Evtroutine Handling(#Com_owner.Click) Origin(#Origin)
…
Endroutine
```

Clearly though, this change of event behavior may have some side effects. In a simple example where there is a click event for both child and parent components, in Win32 the two events would remain separate. However, with DirectX processing and event routing a click on the child would result in both the child click and parent click firing. This is an unusual situation and it is unlikely that many customers will encounter it, but nevertheless it is conceivable and should be catered for.

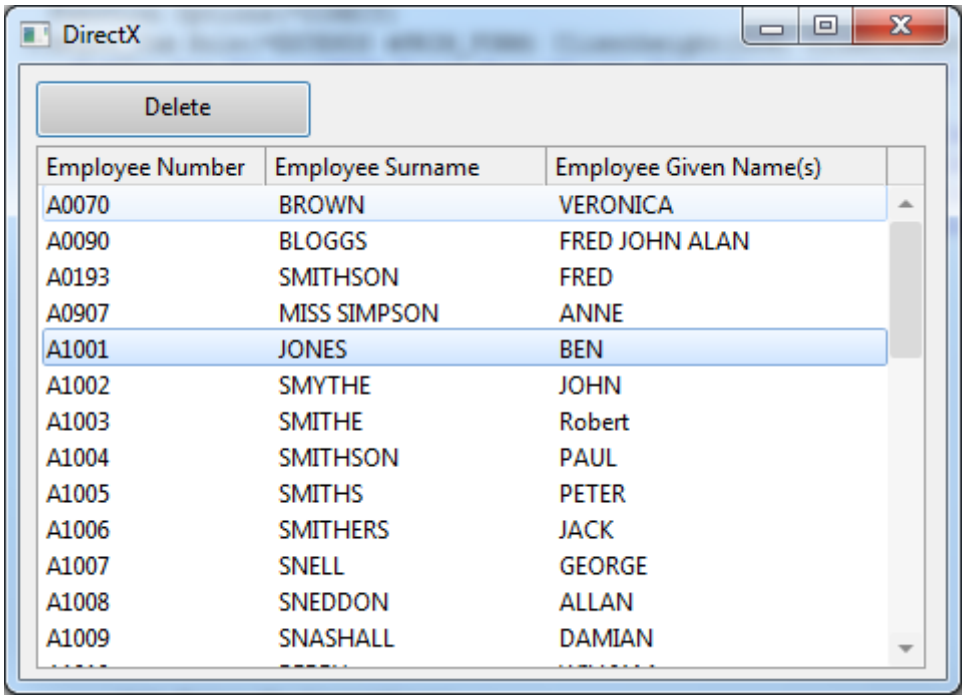
To counter unwanted event propagation, the Handled selector has been added to EVTROUTINE. By setting Handled to true the event is no longer passed beyond the routine being processed.

```
Evtroutine Handling(#Button.Click) Handled(#Handled)

* Stop the event going any further up the parent chain.
#Handled := True

Endroutine
```

Mouse Events, LANSA Lists and CurrentItem

When an application is using DirectX, there are changes to the way in which the list controls (Tree, Grid etc.) appear. Most notably, when the mouse is over an item in the list, the item will be highlighted. In the image below, the 5th item in the list is the FocusItem, i.e. the last item clicked. The 1st item has the mouse over it and is therefore the CurrentItem.

Adopting DirectX



The code for this form is available in the Sample Source section of this document.

All LANSA lists use a CurrentItem concept that maps the equivalent field values from the list and into the equivalent variables. CurrentItem effectively represents the item last processed in the list. This might be the last item clicked, which will also be the FocusItem, or perhaps the last item processed in a Selectlist loop.

Historically, it was common to see processing similar to the image above where the Delete button would cause the deletion of the currently selected item with code similar to the following.

```
Evtroutine Handling(#Delete.Click)

If (#List.CurrentItem *IsNot *null)
Dlt_Entry Number(#List.CurrentItem.Entry) From_List(#List)
Endif

Endroutine
```

This code effectively assumes the CurrentItem and FocusItem are going to be one and the same, and for many scenarios prior to DirectX that would be the case. However, while this may have worked, it wasn't a failsafe mechanism and the reliance on CurrentItem, whilst commonplace, was not best practice.

Mouseover processing in DirectX follows the same rules as all other list based mouse events e.g. ItemGotFocus, ItemGotSelection etc. As soon as the mouse interacts with an item it becomes the CurrentItem. This in turn updates the field values associated with the list.

The result is that it is no longer acceptable to rely on CurrentItem and the existing field values as these can easily be changed by mouse movements. In the previous image, to move from the FocusItem to the Delete button requires moving the mouse over the four items above it.

The last one touched will be the first item, so deleting CurrentItem will therefore delete the first item.

Clicking a popup menu item is similarly affected. As soon as the popup menu closes a mouse event is detected by the list beneath it. Somewhat counter intuitively, this event will precede the click event for the popup menu item.

When running in DirectX it is best practice to strictly adhere to the use of FocusItem. Further, if the code relies on field values being correct at the moment of processing it is prudent to ensure that the code updates the fields from the FocusItem by use of Get_Entry immediately prior to any processing.

```
Evtroutine Handling(#Delete.Click)

If (#List.FocusItem *IsNot *null)

Dlt_Entry Number(#List.FocusItem.Entry) From_List(#List)

Endif

Endroutine
```

True Type Fonts

DirectX rendering only supports True Type fonts. This is simply a reflection of the underlying Microsoft technologies. True Type and Open Type, an extension of True Type, are industry standards and designed to render smoothly regardless of the font size used.

Where a font cannot be rendered, Visual LANSA uses Segoe UI.

Fonts such as MS Sans Serif, which is not True Type, typically have modern True Type alternatives. The MS Sans Serif equivalent is Microsoft Sans Serif.

If you intend to adopt DirectX, it is strongly recommended that you change your application to use a True Type font. This may cause issues with text no longer fitting in the available space and it is recommended that you review any changes you have made.

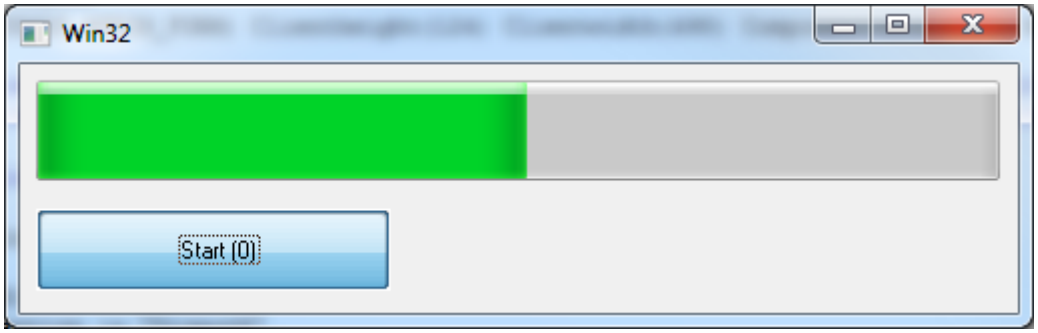
UpdateDisplay

In Win32 the UpdateDisplay method could be called on a control to force the screen to refresh during a long running process. The Win32 runtime was able to address individual controls specifically and in effect could update a small portion of the UI.

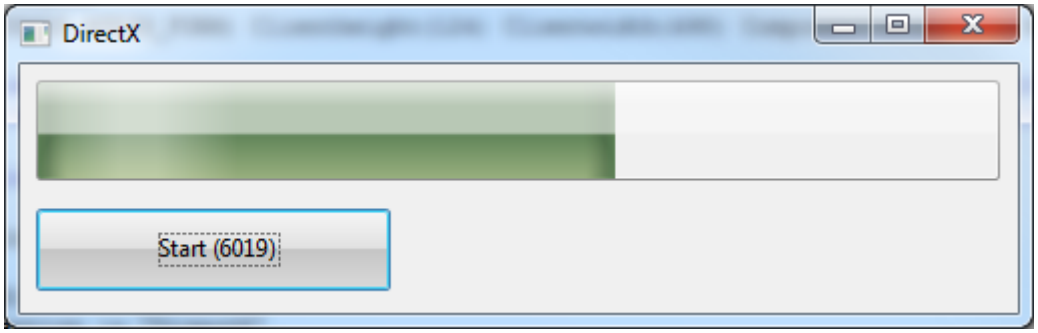
However, the DirectX runtime works in a different manner and this is no longer possible. UpdateDisplay will cause the whole of the form to update.

In most circumstances this will be of little consequence. However, in situations where UpdateDisplay is called repeatedly, this will cause noticeable performance degradation.

A typical situation where that occurs is when a Progress Bar is used. Progress Bars automatically use UpdateDisplay to ensure that they reflect their latest value. In the example below a simple loop is executed and the progress bar and start button caption are updated every iteration.



In Win32 above, the start button is not updated. The UpdateDisplay is specific to the Progress Bar. However, in DirectX, the whole form gets updated.



The code for this form is available in the Sample Source section of this document.

To counteract this situation, rather than updating the progress bar or specifically executing UpdateDisplay every iteration, a simple test can be added so that the update only occurs every 10th time.

Samples Source

Default Appearance

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(119) Clientwidth(336)
Componentversion(1) Height(157) Left(175) Top(215) Width(352)
Define_Com Class(#EMPNO.Visual) Name(#EMPNO) Componentversion(1) Displayposition(1)
Left(8) Marginleft(130) Parent(#COM_OWNER) Tabposition(1) Top(8)
Define_Com Class(#SURNAME.Visual) Name(#SURNAME) Componentversion(1) Displayposition(2)
Left(8) Marginleft(130) Parent(#COM_OWNER) Tabposition(2) Top(32) Width(321)
Define_Com Class(#GIVENAME.Visual) Name(#GIVENAME) Componentversion(1)
Displayposition(3) Left(8) Marginleft(130) Parent(#COM_OWNER) Tabposition(3) Top(56)
Width(321)
Define_Com Class(#PRIM_PHBN) Name(#OK) Buttondefault(True) Caption('&OK')
Displayposition(4) Left(164) Parent(#COM_OWNER) Tabposition(4) Top(88)
Define_Com Class(#PRIM_PHBN) Name(#Cancel) Buttoncancel(True) Caption('&Cancel')
Displayposition(5) Left(252) Parent(#COM_OWNER) Tabposition(5) Top(88)
Evtroutine Handling(#Com_owner.CreateInstance)
Case (#sys_appln.RenderStyle)

When (= DirectX)
#Com_owner.Caption := "DirectX"

When (= Win32)
#Com_owner.Caption := "Win32"

Endcase
Endroutine

End_Com
```

Win32 & DirectX (ActiveX and Clipping)

```
Function Options(*DIRECT)

Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(701) Clientwidth(965)
Componentversion(1) Height(739) Left(189) Top(211) Width(981)

Define_Com Class(#PRIM_PANL) Name(#Panel) Displayposition(1) Height(481)
Horizontalscroll(True) Layoutmanager(#Layout) Left(80) Parent(#COM_OWNER) Tabposition(1)
Tabstop(False) Top(56) Verticalscroll(True) Width(793)

Define_Com Class(#VA_WEBCTL.WebBrowser) Name(#Browser) Displayposition(1) Height(650)
Left(0) Parent(#Panel) Tabposition(1) Top(0) Width(775)

Define_Com Class(#PRIM_ATLM) Name(#Layout)

Define_Com Class(#PRIM_ATLI) Name(#LayoutItem) Attachment(Top) Manage(#Browser)
Parent(#Layout)

Evtroutine Handling(#Com_owner.CreateInstance)

Case (#sys_appln.RenderStyle)

When (= DirectX)
```

```
#Com_owner.Caption := "DirectX"

When (= Win32)

#Com_owner.Caption := "Win32"

Endcase

Endroutine

Evtroutine Handling(#Com_owner.initialize)

#Browser.Navigate( www.lansa.com )

Endroutine

End_Com
```

Transparency and Opacity

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(114) Clientwidth(522) Height(152)
Left(106) Top(204) Width(538)
Define_Com Class(#PRIM_PANL) Name(#Details) Displayposition(1) Height(108) Left(15)
Parent(#COM_OWNER) Tabposition(2) Tabstop(False) Top(13) Width(338)
Define_Com Class(#PRIM_PANL) Name(#Address) Displayposition(2) Enabled(False)
Height(108) Left(15) Parent(#COM_OWNER) Tabposition(1) Tabstop(False) Top(13) Width(338)

Define_Com Class(#PRIM_PHBN) Name(#MoveToFront) Caption('Show Address')
Displayposition(3) Left(360) Parent(#COM_OWNER) Tabposition(3) Top(8) Width(153)

Define_Com Class(#EMPNO.Visual) Name(#EMPNO) Componentversion(1) Displayposition(1)
Height(20) Left(8) Parent(#Details) Tabposition(1)
Define_Com Class(#SURNAME.Visual) Name(#SURNAME) Componentversion(1) Displayposition(2)
Height(20) Left(8) Parent(#Details) Tabposition(2) Top(24) Width(321)
Define_Com Class(#GIVENAME.Visual) Name(#GIVENAME) Componentversion(1)
Displayposition(3) Height(20) Left(8) Parent(#Details) Tabposition(3) Top(48) Width(321)
Define_Com Class(#ADDRESS1.Visual) Name(#ADDRESS1) Componentversion(1)
Displayposition(1) Height(20) Left(8) Parent(#Address) Tabposition(1) Width(300)
Define_Com Class(#ADDRESS2.Visual) Name(#ADDRESS2) Componentversion(1)
Displayposition(2) Height(20) Left(8) Parent(#Address) Tabposition(2) Top(24)
Usepicklist(False) Width(300)
Define_Com Class(#ADDRESS3.Visual) Name(#ADDRESS3) Componentversion(1)
Displayposition(3) Height(20) Left(8) Parent(#Address) Tabposition(3) Top(48) Width(300)
Define_Com Class(#POSTCODE.Visual) Name(#POSTCODE) Componentversion(1)
Displayposition(4) Height(20) Left(8) Parent(#Address) Tabposition(4) Top(72)
Usepicklist(False) Width(249)

Evtroutine Handling(#MoveToFront.Click)

If (#Details.DisplayPosition <> 1)

#Details.DisplayPosition := 1
#Details.enabled := True
#Address.enabled := False
#MoveToFront.Caption := "Show Address"
```

```
Else

#Address.DisplayPosition := 1
#Details.enabled := False
#Address.enabled := True
#MoveToFront.Caption := "Show Details"

Endif

Endroutine

End_Com
```

Mouse Events

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(306) Clientwidth(462)
Componentversion(1) Height(344) Left(103) Top(200) Width(478)
Define_Com Class(#PRIM_Trvw) Name(#List) Columnbuttonheight(19) Componentversion(2)
Displayposition(1) Fullrowselect(True) Haslines(False) Height(261)
Keyboardpositioning(SortColumn) Left(8) Linesatroot(False) Parent(#COM_OWNER)
Tabposition(1) Top(40) Viewstyle(UnLevelled) Width(444)

Define_Com Class(#PRIM_TVCL) Name(#TVCL_1) Displayposition(1) Level(1) Parent(#List)
Source(#EMPNO) Width(27)
Define_Com Class(#PRIM_TVCL) Name(#TVCL_2) Displayposition(2) Level(2) Parent(#List)
Source(#SURNAME) Width(33)
Define_Com Class(#PRIM_TVCL) Name(#TVCL_3) Displayposition(3) Level(3) Parent(#List)
Source(#GIVENAME) Width(40)
Define_Com Class(#PRIM_SPBN) Name(#Delete) Caption('Delete') Displayposition(2) Left(8)
Parent(#COM_OWNER) Tabposition(2) Top(8) Width(137)

Evtroutine Handling(#Com_owner.CreateInstance)

Case (#sys_appln.RenderStyle)

When (= DirectX)
#Com_owner.Caption := "DirectX"

When (= Win32)
#Com_owner.Caption := "Win32"

Endcase

Select Fields(#List) From_File(pslmst)

Add_Entry To_List(#List)

Endselect

Endroutine

Evtroutine Handling(#Delete.Click)
```

```
If (#List.CurrentItem *IsNot *null)

Dlt_Entry Number(#List.CurrentItem.Entry) From_List(#List)

Endif

Endroutine

End_Com
```

UpdateDisplay

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(124) Clientwidth(498)
Componentversion(1) Height(162) Left(261) Top(195) Width(514)

Define_Com Class(#PRIM_PGBR) Name(#ProgressBar) Displayposition(1) Left(8)
Maximumvalue(10000) Minimumvalue(0) Parent(#COM_OWNER) Tabposition(1) Top(8) Value(1)
Width(481)
Define_Com Class(#PRIM_PHBN) Name(#Start) Caption('Start (0)') Displayposition(2)
Height(41) Left(8) Parent(#COM_OWNER) Tabposition(2) Top(72) Width(177)

Evtroutine Handling(#Com_owner.CreateInstance)

Case (#sys_appln.RenderStyle)

When (= DirectX)
#Com_owner.Caption := "DirectX"

When (= Win32)
#Com_owner.Caption := "Win32"

Endcase

Endroutine

Evtroutine Handling(#Start.Click)

#ProgressBar.value := 0

Begin_Loop To(10000)

#ProgressBar.value += 1

#Start.Caption := ("Start (&1)").Substitute( #ProgressBar.Value.Asstring )

End_Loop

Endroutine

End_Com
```