InterShell Developer's Manual

Document notes

Fonts used

Lucida console – mono-spaced font represents file and path names.

OCRA represents code.

Default times roman represents anything else.

Unsure of the ordering here, someone else should consider how to lay out these boards. There are several sections that can be described, and they all need to be covered, but I'm not sure of the order which would be most educational.

InterShell is a high level layout manager for PSI based controls. PSI is largely based on SACK's image library and image rendering drivers. There are many levels at which applications can be developed.

This document is a focus from the top(the highest) down(to the lowest of the high levels).

InterShell has several layers.

The first file that needs to be included is in

$(SACK_BASE)/src/InterShell/intershell_registry.h

There are 3 types of intershell-specific controls that InterShell manages: Buttons, List Boxes, and other.

Buttons, these are high level buttons based on a custom draw button control from the PSI library. The custom drawn button is created and managed as a intershell/widgets::keypad.button. Actually the button namespace is peer to keypad, the InterShell widget keypad itself uses these buttons.

Buttons have a glare set (A set of one or more images that determine what it looks like). These glare sets are managed by InterShell. Buttons have a text layer and a image layer. These layes may both exist, in which case the text is placed on the image. Buttons have two or more colors which may be applied to the glare set and text.

The API for InterShell plugins begins to stray from C syntax. A C syntax highlighting environment may not understand exactly how to color this, and auto completion also fails, as the syntax to define a entry point fails standard C rules.

Methods associated with control events are defined with a name such as OnCreateMenuButton. These names represent a macro defined in `intershell_registry.h` which represents a macro that builds an pre-function which does the registration of the method within the process registry of SACK. (http:///sack/docs/procreg.html, if one existed. `sack/include/procreg.h`)

The next token is a name in parenthesis, such as (“my_class/my_control”). The name chosen for a control is entirely up to the coder, but, this name must be the same for all control events defined. This name is what makes all methods defined in code related. There is no specific structure which must be filled in, and there is no particular extra code that the developer needs to write. The only requirement is that the methods below be used with appropriate syntax. Oh, a note on names, these represent the name that is in the “Create Other…” menu when right clicking in a selected region (additionally, if it is a button, this name may also appear as a control for macro keys and for the startup macro). A name prefix separated by a slash (/ or \ ) will denote a sub-menu of controls. Recommend HIGHLY prefixing all controls within at least one level of path. “User”, “Slot”, “Poker”, “Schedule Editor”. Names may contain spaces. Names may not contain “/” or “\”, as this will be interpreted as a namespace path separation. Names may not contain a nul (‘\0’ or 0), as this ends the name. Spaces and human names are also highly recommended, although phrases should be avoided, described nouns may fit.

Following the control type name are the parameters for the function declared with variable names that the function would like to regard the parameters as: (PMENU_BUTTON this_button). The parameters are special for each type of method, and these are described below.

So, all together it would look like…

```
OnCreateMenuButton( “my_class/my_control” )
    ( PMENU_BUTTON button_parameter )
{
   Return 0;
}
```

Problem: There is no return type specified. There is SO much code which has been developed against this API, that it would be hard to change, though it would be possible to enforce having the developer specify the return type in code, so he would know what sort of return would be required. The compiler would generate errors indicating the

wrong return type (like it does already with wrong parameter types). The types defined for each control event defined below are the only clear documentation of the return types.

Events are defined in some particular order: Creation and destruction, and then everything else, itself in no particular order.

OnCreateMenuButton
Return type: PTRSZVAL
Parameters: PMENU_BUTTON

The value returned is a PTRSZVAL which is any developer specified non zero value. This result is as large as and no larger than a pointer, but the type behaves as an unsigned integer. This value is passed as the PTRSZVAL in all other events. This value is the application developers specific way to identify which control is being referenced. Since the size is large enough for a pointer, a pointer to some allocated structure, to some static element, or to an element of an array may be used. Failing that this value may be an integer representing an index into an array. Examples may be provided of all of these methods. Some buttons do not even care which button instance they are, and may simply return 1. Returning zero from this function causes the control creation to fail, and possibly to continue attempting to create the control as another simarly named control. It would be possible to define an OnCreateMenuButton and an OnCreateListbox that used the same registry name, and one could fail allowing the other to actually do the create. If all methods associated with a particular control name fail, the control is hidden, but remains on the current page. The control is saved, so that it may one day have a creation method which actually works.

The parameter passed is a PMENU_BUTTON. Methods for working with PMENU_BUTTON types are defined in `sack/src/intershell/intershell_exports.h`. This item may be ignored. There are default methods for configuring the button available from InterShell. If a custom configuration method is defined, then one may need to consult how to work with a PMENU_BUTTON type.

Example: OnCreateMenuButton(“Quit POS”);

OnCreateListbox
Return type: PTRSZVAL
Parameters: PSI_CONTROL

Return (See OnCreateMenuButton return)

Parameter PSI_CONTROL methods are defined in sack/include/controls.h. The methods specifically assocated with listboxes (see MakeListbox, AddListItem), are applicable to this listbox.

There are some events defined below that allow a InterShell layer interfacing to the control such that the developer does not need to consider saving the PSI_CONTROL of the listbox.

OnCreateControl

Return type: PTRSZVAL
Parameters: PSI_CONTROL parent_control(frame)
, S_32 x, S_32 y
, _32 width, _32 height

Return (See OnCreateMenuButton return)

This method allows the user to create any particular control within the PSI_CONTROL passed. The size that the control should be and the location within the parent PSI_CONTROL is also specified.

Many events are specifically for these Control types, so that the user has intimate control over what InterShell observes about the control.

OnDestroyMenuButton

Return type: void
Parameters: PTRSZVAL

There is no result desired. There is no usable status about the control, the plugin is required to do any cleanup itself, and no-one cares if it fails or not.

The parameter PTRSZVAL is the result returned from the associated OnCreateMenuButton, OnCreateListbox, or OnCreateControl.

#define OnDestroyListbox OnDestroyMenuButton
#define OnDestroyControl OnDestroyMenuButton

blah, there really does have to be a simple way to say this, I just think that the above two lines are nothing but confusion for developers *boggle*.

Control specific events/methods/…

OnGetControl

Return type: PSI_CONTROL
Parameters: PTRSZVAL

The return is the PSI_CONTROL that was created when OnCreateControl was invoked. (Remember, we make a control within the parent frame, of a certain size. I also mentioned some time ago that InterShell really only managed one type of control – a PSI_CONTROL type. MenuButtons are buttons, which are custom drawn PSI buttons. Listboxes are PSI listboxes. Controls are User defined PSI controls (typically).

The parameter PTRSZVAL is the result returned from the associated OnCreateMenuButton, OnCreateListbox, or OnCreateControl.

OnFixupControl

Return type: void
Parameters: PTRSZVAL

The parameter PTRSZVAL is the result returned from the associated OnCreateMenuButton, OnCreateListbox, or OnCreateControl.

#define OnFixupMenuButton OnFixupControl

Events common to all controls

These events are generally useful when defined for pretty much any control. But, on the other hand, should not be used. There are dozens of ways to ___ and they'll all get you from point A to point B, but consider the work that is actually done when you get further into implementation, overhead of certain directions may be counter productive…. If only I could warn you about that cliff you'll fall over, long before you ever start to ascend from sea level.

OnSaveControl

Return type: void
Parameters: FILE *, PTRSZVAL

FILE * parameter is a currently open “text” file with the current position at the end. The desired effect here is to generate fprintf(file, …) statements to output the parameters of the current control identified the by PTRSZVAL returned from a OnCreate method. These lines should have compatible AddConfigurationMethod() prototypes defined in an OnLoadControl event.

This is called after the creation tag is written to the configuration file, and before the creation tag is ended.

OnLoadControl

Return type: void
Parameters: PCONFIG_HANDLER, PTRSZVAL

The OnLoadControl is a very common method to define. Data may be defined to be saved within the configuration file which InterShell uses for saving location and control type data.

The first parameter, a PCONFIG_HANDLER is used with methods from `sack/include/configscript.h`. Although the application may use the “PTRSZVAL OnCreate result” to load the configuration from some other place, it may add configuration methods to the config handler before it continues to process.

When Called: This method is invoked after a line stating that a generic control should be created. (After a create control tag is handled?), and before any lines after the create are processed.

OnConfigureControl
#define OnEditControl OnConfigureControl

Return type: PTRSZVAL
Parameters: PTRSZVAL, PSI_CONTROL

The return PTRSZVAL value should be the same as the PTRSZVAL passed to the configure function. The module may return a different PTRSZVAL here, and all subsequent invocations of events that take a PTRSZVAL specifying the control will be passed this new value. This allows a configure method to create an entirely new structure for the control upon configuration.

Parameter PTRSZVAL is the result of the creation method.

The PSI_CONTROL passed to this even is the control which should be the parent of the configuration frame that should be created here.

This event is intended for the application to spawn a form which can configure properties on the particular control in question.

This event is called when the 'Edit' option is selected from the right click menu within a control. This event is spawned in a separate thread, and is allowed to wait indefinitely before returning.

This is an expected user interaction defining the properties of the control.

There are methods SetCommonButtonControls() and GetCommonButtonControls() that may be used for a PSI frame with controls with 'standard identifiers' defined on it. These resource IDs are registered under intershell/* when editing the properties of a PSI control. Color, font, text, background image, glare set, next page are all common properties of a button. If you are working with a listbox or a control these may or may not apply. Font is a common thing for listboxes.

OnShowControl

Return type: void
Parameters: PTRSZVAL

The parameter PTRSZVAL is the result returned from the associated OnCreateMenuButton, OnCreateListbox, or OnCreateControl.

#define OnQueryShowControl(name) \

OnQueryShowControl

Return type: LOGICAL
Parameters: PTRSZVAL

Return false(0) or true(not 0). If the return value is false, then the control is not revealed, otherwise, the control is revealed. A subsequent invocation of the OnGetControl method of the control may happen (this may happen before the query happens). This allows the plugin developer to keep controls hidden on demand (for instance no data to populate them such as item selection buttons from a variable list of items).

The PTRSZVAL is the result of the OnCreate method.

This is called on returning from edit mode, and when changing pages. One may cause a page change to the same page to happen (ShellSetCurrentPage(ShellGetCurrentPage());) which will cause all controls to be queried (assuming they registered this method).

Events specific to buttons

`Header: InterShell/include/widgets/buttons.h`

OnKeyPressEvent(name)

Return type: void
Parameters: PTRSZVAL

This event is defined in `widgets/buttons.h`, however InterShell is aware of this method, and creates buttons in such a way that this method is referenced. The namespace of this method registration is different from all other defined registered methods, but for the developer's perspective, the name defined should be the same as the name used for the OnCreateMenuButton. The PTRSZVAL passed is the result of the OnCreateMenuButton event that was invoked when the button was created.

WHY IS THIS UNDERLINED?!

Events for listboxes specifically

These events are specific to the behavior of listbox controls created with OnCreateListbox. If you create a listbox as your control within OnCreateControl these will not be invoked. These also take TWO names instead of just one when registering their events. The othername has significance, I just forget what.

OnSelectListboxItem(name, othername)

Return type: void
Parameters: PTRSZVAL, PLISTITEM

When a item is selected in a listbox (each item selected in a multiselect list), all methods that have been registered, with 'othername' as the name of the control this is, are invoked.

OnCreateListbox("Sessions");
OnCreateListbox("Games");
OnSelectListboxItem("Sessions", "Games")

Assuming both a ‘Sessions” list and a “Games” list have been created, whenever an item is selected in a “Sessions” listbox, All “Games” listboxes are called with this event.

The PTRSZVAL is the result of the OnCreateListbox(). The PLISTITEM is the item that has been selected.

OnDoubleSelectListboxItem(name, othername)

Return type: void
Parameters: PTRSZVAL, PLISTITEM

This works like OnSelectListboxItem, but only when an item is double clicked. (Not really selected, just double clicked)

Events that exist, but are near deprication

OnSaveMenuButton

Return type: void
Parameters: FILE *, PMENU_BUTTON, PTRSZVAL

Called as an option for OnSaveControl. If created with OnCreateMenuButton method, then the PMENU button is also passed here. The PTRSZVAL value should have been sufficient for this to save data, but this was old, and once upon a time everyone knew everything about buttons.

OnEditBegin

Return type: void
Parameters: PTRSZVAL

This is called when configuration mode is begun, and only for those controls currently visible. Once upon a time the edit mode for InterShell left all controls visible and active, allowing them to be visually resized in realtime, performance became a huge hit with a few transparent controls on a transparent background in a transparent frame. This method is not so important anymore, as controls are actually hidden. The clock addon used this method to call ‘StopClock’ so that its time did not update forcing false displays.

OnEditEnd

Return type: void
Parameters: PTRSZVAL

This is called when edit mode is being exited; The escape key is pressed during edit, or the ‘Done’ method is invoked on the popup menu.

Events that are non-control specific.

The registry namespace of common controls is intershell/common/<appname>
They are not intended to have paths. They may not work if a control class path is included. The name is merely to give them a unique identifier, if multiple events are registered with the same name, only one, of a semi-indeterminate order will be done.

OnChangePage

Return type: int
Parameters: void

This event is invoked before a page change is done. The application has the option of returning FALSE which will disallow the page change, and stop calling all other registered OnChangePage events.

OnKeypadEnter

Return type: void
Parameters: void

This event is only performed if the `keypad.isp` is loaded. When the enter key on the keypad is pressed these events are called. All events registered are called.

OnLoadCommon

Return type: void
Parameters: PCONFIG_HANDLER

This event is called before any of the configuration file is read. This allows the plugins to AddConfigurationMethod()s to handle global options. One example of this option is "Wait For Network Drive?".

OnSaveCommon

Return type: void
Parameters: FILE *

This event is called before any controls are saved to the configuration file. This allows the writing of common(global) options.

OnFinishInit

Return type: void
Parameters: void

This event is invoked after the configuration file is read, and closed. This event is invoked before the application initially selects any page. This may change pages to select a different startup page than default, may complete initialization, knowing now exactly how many controls have already been created…

OnInterShellShutdown

Return type: void
Parameters: void

This is invoked when InterShell's quit method is invoked. InterShell_Exit() or clicking the 'POS Quit' button… or the user selecting alt-F4 or alt-X to end the application. Control-C will also be caught and this event will be dispatched at exit.

This method allows modules like tasks.isp to close all tasks before exiting.

OnGlobalPropertyEdit

Return type: void
Parameters: PSI_CONTROL (frame)

The PSI_CONTROL passed to this event is intended to be the parent of the dialog used to configure some global property. This event's class_name is used to create popup menus for editing global properties if the property options are complex.

The name registered for this control will appear in the menu “Other Properties…” on the main menu, right clicking in a non selected, non-control region.

(Usage of parameter… DisplayFrameOver(YourConfigFrame, frame); or LoadXMLFrameOver(frame, “YourConfigFilename.xml”);

A Simple Button

This simple button, when clicked will show a message box indicating that the button was clicked. For button controls there is a default configuration dialog that is used for controls that do not themselves define a OnConfigureControl(OnEditControl) event handler. The default dialog, depending on its current design is able to set all relavent properties common to buttons, such as colors, font, button style, perhaps a page change indicator.

```
OnCreateMenuButton( “basic/Hello World” )( PMENU_BUTTON button )
{
	return 1; // result OK to create.
}

OnKeyPressEvent( “basic/Hello World” )( PTRSZVAL psvUnused )
{
	SimpleMessageBox( NULL  // the parent frame, NULL is okay
		, “Hello World!”   // the message within the message box
		, “Button Clicked” );  // the title of the message box.
	// SimpleMessageBox displays a simple frame with a message
	// and a title, and an OK button. The function waits
	// until the OK button is clicked before returning.
}
```

A Simple Listbox

```
OnCreateListbox( “basic/List Test” )( PSI_CONTROL pc_list )
{
	return pc_list; // result non-zero OK to create.
	// this result can also be used in subsequent events
	// by typecasting it back to the original PSI_CONTROL
	// type that it is.
}

// several implementations of listboxes change their content
// based on the state of other controls around them, and/or
// database content.  The OnShowControl is a convenient place
// to re-populate the listbox with new data.
// There is no requirement to do this in this way.
// Some listboxes may populate their content during OnCreate.

OnShowControl( “basic/List Test” )( PTRSZVAL psvList )
{
```

```
        PSI_CONTROL pc_list = (PSI_CONTROL)psvList;
        ResetList( pc_list );
        AddListItem( pc_list, “One List Item” );
        AddListItem( pc_list, “Another List Item” );
}
```

A Simple Control

There is no such thing as a ‘simple’ control.

```
OnCreateControl( “basic/Other Control” )( PSI_CONTROL frame, S_32 x, S_32 y, _32 w, _32 h )
{
        // this code results with a create PSI control.
        // the PSI control must have been previously registered
        // or defined in some way.
        // the result of creating an invalid control,
        // or of creating a control that fails creation
        // for one reason or another is NULL, which will in turn
        // fails creation of the InterShell control.
        return (PTRSZVAL)MakeNamedControl( frame
                                        , “Some PSI Control type-name”
                                        , x, y  // control position passed to event
                                        , w, h  // control size passed to event
                                        , -1 ); // control ID; typically unused.
}

// For InterShell to be able to hide the control when pages change,
// show the control when pages change, move the control to
// a new position, or to resize the control, this method MUST
// be defined for InterShell widgets created through OnCreateControl.

OnQueryGetControl( PTRSZVAL psv )
{
        // since we know that we returned a PSI_CONTROL from the
        // creation event, this can simply be typecast and returned.
        return (PSI_CONTROL)psv;
}
```

What is a InterShell plugin?

InterShell modules are simply share object libraries (.DLL, .so). By default they should be named with an extension of “`.isp`”, as the default behavior of intershell.core is to load “*.isp” files. Although, with the `<program name>.plugins` file detailed in the InterShell users section, any file may be specified. The file is loaded as a shared library, depending on the platform different methods are used, but no platform supports loading a

program as a shared library. Any created .exe or other program output is unusable by InterShell as a plugin.

The library is only loaded, no specific method is required by InterShell. Many other implementations of plugin components require a specific routine or routines to be defined such as RegisterRoutines() and UnloadPlugin() required by Dekware plugins.

The event handler registration is handled by the PRELOAD() method available through the SACK interface. There are several compiler specific methods that may be used, and SACK abstracts PRELOAD into one of the methods which will work for the platform and compiler used. When the library is loaded, all PRELOAD type functions are executed, which will register event handlers with the names of the controls that they relate to. PRELOAD is also much like a static class constructor. When a library is loaded with static classes, the constructors of these classes are invoked by the loader/library deadstart code.

```
static class blah {
public:
        blah()
        {
                // init code here …
        }

} init_me;
```

In fact, some [C++] compilers used to compile SACK may use a class to accomplish preload, without using some lower level compiler specific feature.

Through some other mechanism names may be registered in the SACK process(procedure) registry tree used by InterShell, however, all registration must occur during library load time, as no other entry point is defined or required.

InterShell Exported Functions

void SetCommonButtonControls(PSI_CONTROL frame)

// magically knows which button we're editing at the moment.
// intended to only be used during OnEditControl() Event Handler
This function is to be used during OnEditControl.
This function takes the current properties from the current control being edited, and sets controls in the dialog passed to reflect the control's current properties.

void GetCommonButtonControls(PSI_CONTROL frame)

// magically knows which button we're editing at the moment.
// intended to only be used during OnEditControl() Event Handler
This function is to be used during OnEditControl.
It reads values from the frame's controls which match the ID of common controls. An edit field with the ID of EDT_CONTROL_TEXT for instance will be read into the button's text field. While this method can be used for ANY control type, it is intended for use with buttons. Buttons will allow setting of colors, choosing of fonts, modifying the text and/or image of the button. Listboxes also have common properties, font for instance, but additionally check for a couple check boxes for multi select and lazy selection options.

The control being currently configured is the one that receives the modifications from reading the dialog.

InterShell Methods for modifying control properties

// a zero (0) passed as a primary/secondary or tertiary color indicates no change. (err or disable)
#define COLOR_DISABLE 0x00010000 // okay transparent level 1 red is disable key. - cause that's such a useful color alone.
#define COLOR_IGNORE 0x00000000
void InterShell_GetButtonColors(PMENU_BUTTON button
, CDATA *cText
, CDATA *cBackground1
, CDATA *highlight);

Returns the colors set in the button specified to the variables. Any NULL passed to this function will of course not result with that color…

void InterShell_SetButtonColors(PMENU_BUTTON button, CDATA cText, CDATA cBackground1, CDATA cBackground2, CDATA cBackground3);

Set's the color attributes of a button. The defined symbol COLOR_DISABLE may be used to remove the color entirely from a button. The defined symbol COLOR_IGNORE may be used to ignore modification of the color. There are not yet buttons which use all 3 background colors.

void InterShell_SetButtonColor(PMENU_BUTTON button, CDATA primary, CDATA secondary);

This function works like InterShell_SetButtonColors, but passed COLOR_IGNORE for the text color and third background color, thereby only modifying the background color of buttons.

void InterShell_SetButtonText(PMENU_BUTTON button, CTEXTSTR text);

Modify what the text on a button says. The '_' character may be used to cause the text to have a newline in it… "Set_Text" will result with two lines the first "Set" and the next "Text" all centered within the button (left-right and top-bottom).

void InterShell_SetButtonFont(PMENU_BUTTON button, Font *font);

This expects a font pointer to be passed, the functions SelectAFont and UseAFont are the preferred methods for getting such a value.

Font* InterShell_GetCurrentButtonFont(void)

Depricated, rely on set/get common button controls instead.

This returns the font of the button currently configured. Needs to be used within OnEditControl event handler.

void InterShell_SetButtonStyle(PMENU_BUTTON button, char *style)

This modifies the glare-set defined for the button. (The images that make it appear round/square/fluffy, etc) The style may be some name not previously known, this will then appear as an available glare-set to modify properties of.

void InterShell_SaveCommonButtonParameters(FILE *file)

Saves the configuration of the current button into the file specified. Intended usage is within OnSaveControl. If the control IS a button, then the save code will automatically save the common properties of the button. This is intended for custom controls which may use some common properties otherwise.

// wake up menu processing... there's a flag that was restart that this thinks
// it might want...
void RestartMenu(PTRSZVAL psv, _32 keycode)

This is used to cause the menu to unload, and reload its configuration and re-display. The parameters psv and keycode are not used, these are merely relics of being attached to a key method (alt-F5?)

void ResumeMenu(PTRSZVAL psv, _32 keycode)

There is no real use for this method. Apparently it wakes up the main process, but then all it will do is go back to sleep….

void UpdateButtonEx(PMENU_BUTTON button, int bEndingEdit)
#define UpdateButton(button) UpdateButtonEx(button, TRUE)

The preferred method it to use UpdateButton(button) . This causes all properties on the button to be flushed to the display. The above methods SetButtonColor, SetButtonFont, SetButtonText, these all have no immediate effect until this is called or the page is changed.

InterShell Interface for Page control

//--
// pages controls here...
//

PPAGE_DATA ShellGetCurrentPage(void);

Result with the current page.

PPAGE_DATA ShellGetCurrentPageEx(PSI_CONTROL pc_canvas);

If you did something silly like made your own sub-canvas, this will get the current page on the sub-canvas.

```
// special names
// start, next, prior are keywords that imply direct
// page stacking.
int ShellSetCurrentPage( TEXTSTR name )
```

Sets the current page, by name. Each page has a name. There are 3 special names: “first”, which sets the current page to the startup/default page, “next” which goes to the next page in order, “here” which sets the current page to the current page, but in the process causes a refresh of all controls on the current page.

```
int ShellSetCurrentPageEx( PSI_CONTROL pc, TEXTSTR name )
```

Same as ShellSetCurrentPage above, but takes a control handle so one can set the pages on their sub-canvas.

```
// a call will push the current page on a stack
// which will be returned to if returncurrentpage is used.
// clear page list will flush the stack cause
// there is such a temptation to call to all pages, providing
// near inifinite page recall (back back back)
int ShellCallSetCurrentPage( TEXTSTR name )
```

Same as ShellSetCurrentPage, but saves the current page in a stack of prior pages, so ShellReturnCurrentPage can be used.

```
int ShellCallSetCurrentPageEx( PSI_CONTROL pc_canvas, TEXTSTR name )
```

Same as ShellCallSetCurrentPage but handles pages within a sub-canvas.

```
void ShellReturnCurrentPage( void )
```

Return to the last page that was pushed on the stack. If no pages are left on the stack, the startup/first/default page is selected.

```
void ClearPageList( void )
```

Clears the prior page list that ShellReturnCurrentPage pops from.

```
// disable updates on the page, disable updating of buttons...
void InterShell_DisablePageUpdate( LOGICAL bDisable )
```

Sometimes one has a significant amount of work to do when updating a page, it may be wise to disable updates while the updates are happening (disallows the actual display of updates), until all updates are completed, then re-enable PageUpdate will flush all changes to the display.

void RestoreCurrentPage(PSI_CONTROL pc_canvas)

Depricated even as it was defined.
Unhide a page. (Unneeded).

void HidePageEx(PSI_CONTROL pc_canvas)

Depricated even as it was defined.
Hides the controls on a sub-canvas. (first stage of changing pages)

```
//---------------------------------------
void InterShell_DisableButtonPageChange( PMENU_BUTTON button );
// this sets a one time flag on a button which disables
// the auto page change which may be assigned to a button.
```

Sets a one-shot disable on page update. Buttons have a possible default behavior of changing pages when they are invoked, if a page to change to is defined for them. This allows code to disable this auto change. (A Tender button for instance, if the sale fails to complete, will want to not change page back to startup conditions).

InterShell Text Label Support

```
//-------------------------------------------------------
// PVARIABLE types are used in Other/Text Label
//   a %<varname> matches the name exactly
//   a revision needs to be done to offer a different variable sepearateor that can
//   imply sub function... suppose a simple xml stealing of <varname> will work
//   implying that optional paramters may be <function param1=value1 ...>
// CreateLabelVariable returns a value that may be used in...
// LabelVariableChanged
//  to cause the update of all boxes which may or may not be visible at the time, and
//  will cause approrpiate refresh
//

typedef struct variable_tag *PVARIABLE;

/* moved from text_label.h
```

```
 these values need to be exposed to peer modules
 */
enum label_variable_types {
        LABEL_TYPE_STRING
                                                        , LABEL_TYPE_INT
                                                        , LABEL_TYPE_PROC
};
typedef CTEXTSTR  *label_string_value;
typedef _32       *label_int_value;
typedef CTEXTSTR (*label_gettextproc)(void);

PVARIABLE CreateLabelVariable(
                char *name
                , enum label_variable_types type
                , POINTER data )
```

This function creates a variable usable by text labels. The variable name will be referenced using '%' followed by the text of the name specified here. The variable will then be de-referenced depending on the type registered.

Label type LABEL_TYPE_STRING expects the 'data' parameter to be a CTEXTSTR *, that is a pointer to a CTEXTSTR. CTEXTSTR is 'const char *'.

Label type LABEL_TYPE_INT expects the 'data' parameter to be a S_32 *, that is a pointer to a S_32 type. S_32 is a signed 32 bit value, int? long? Depends on platform and compiler.

Label type LABEL_TYPE_PROC expects the 'data' parameter to be a CTEXTSTR (*)(void). That is, a function pointer that returns a CTEXTSTR type.

```
void LabelVariableChanged( PVARIABLE );
```

This function will cause any text labels which are currently being displayed that are using the specified variable to update, and re-get the value of the variable.

If passed NULL, all text labels are updated, and all variables re-queried.

```
void LabelVariablesChanged( PLIST );
```

This function expects a PLIST of PVARIABLEs. For each PVARIABLE in the list, LabelVariableChanged is called.

Buttons defined within the sack_widgets library have the ability to have text fields located in arbitrary places with independent colors and fonts. This exposes that functionality through the higher level PMENU_BUTTON instead of having to get the PKEY_BUTTON from the menu button.

```
//-------------------------------------------------------
// layout members which have a position x, y, font, text and color of their own
// may be created on buttons.  They are displayed below the lense/ridge[up/down] and
above the background.
PTEXT_PLACEMENT AddButtonLayout(
                PMENU_BUTTON pKey
                , int x, int y
                , Font *font
                , CDATA color
                , _32 flags );
```

pKey is the PMENU_BUTTON button.
X, y is the position of the text within the button in relative coordinates. Negative coordinates may be used here, this will bias the text from the opposite edge…
A positive X value will be relative to the left edge, and the text field will be left aligned.
A negative X value will be relative to the right edge, and the text field will be right alightned.
A positive Y value may be used to bias the text from the top.
A negative Y value may be used to bias the field from the bottom of the control.
Color – color fo the text.
Font * - the font pointer, SelectAFont and UseAFont may be used to get an appropriate value for this.
Flags – have to consult the widgets/buttons.h for these.

```
void SetButtonTextField( PMENU_BUTTON pKey
                , PTEXT_PLACEMENT pField
                , char *text );
```

This updates the text in a field created previously.

```
PSI_CONTROL InterShell_GetButtonControl( PMENU_BUTTON button )
```

This method allows the user to get the real PSI_CONTROL from the button. This method is NOT for general use, and you really need to have a good reason to use this.
// this provides low level access to a button, let the programmer beware.

void BeginCanvasConfiguration(PSI_CONTROL pc_canvas)

If a control (such as the slot game control) creates a menu canvas type, this allows the configuration loas subsystem to load the configuration from the current configuration file. This is the compliment to SaveCanvasConfiguration.

Should be used within OnLoadControl.

void SaveCanvasConfiguration(FILE *file, PSI_CONTROL pc_canvas)

Used within OnSaveControl. This is used to save a menu canvas' configuration. A slot game control uses this.

// BeginSubConfiguration....
// colntrol_type_name is a InterShell widget path/name for the type of
// other info to save... the method for setting additional configuration methods
// is invoked by thisname.
// Then end_type_name is the last string which will close the subconfiguration.
void BeginSubConfiguration(char *control_type_name, char *end_type_name)

This is used within OnLoadControl.
Some controls have complex sub configurations that may themselves contain control definitions that the save subsystem will need to know about. The control type name is the control_type_name is used to call the appropriate OnLoadControl to condition the configuration evaluator to have the new methods.

Two existing uses of this method are Macors and Tasks. Macros use this for loading the macro elements, which are framed with "Macro Element <button type name>" and "Macro done". The tasks that are auto tasks, are outside of the button configuration, so during OnLoadCommon() the task module registers to handle "Auto Task", "Auto Network Task", and "Command Shell" which begins a sub configuration.

A sub-configuration consists of pushing the current configuration methods defined on a configuration handler, and creating an empty processing stack. Entirely new methods are then added to the configuration handler, which are used to process until the line specified with end_type_name is encountered. Then the current configuration methods are popped off the stack and destroyed, reverting back to the prior processing state.

char * EscapeMenuString(char *string);

This returns a string appropriately escaped for reloading from a configuration file. The character '#' indicates a comment, typically, however if it is preceeded by a '\' then it is a '#' character instead. This means that the character '\' may also not be read exactly as it is, therefore for the literal character '\', '\\' needs to be saved into the configuration file. This function handles inserting '\' before '\' and '#' characters. This uses a single static buffer for this, therefore if you want to escape multiple strings, you need to save the result separately using StrDup, strdup or some other simlar method.

Example of bad usage:

```
fprintf( file, "filename: %s path:%s \n"
        , EscapeString( "simple#name" )
        , EscapeString( "c:\ftn3000\working" ) );
```

The above example will return the same buffer so the pathname will be the same text as the filename… and the output will look like

```
filename: simple\#name path:simple\#name;
```

InterShell Font Methods

typedef struct font_preset_tag *PFONT_PRESET;

```
Font * SelectAFont(
        PSI_CONTROL parent
        , CTEXTSTR *default_name )
```

Select a font allows the user to choose a font. Parameters are the parent frame that the font selection dialog should slave to. Default name is the address of a CTEXTSTR that starts with the original name. This name is updated to the new preset name.

The result is a pointer to a font, this allows the real font pointer to be maintained externally. Do not modify the content (*result or result[0]).

```
Font * UseAFont( CTEXTSTR name )
```

Select a preset font name to use. The resulting font* pointer can be passed to InterShell_SetButtonFont.

InterShell Less than useful methods

void GetPageSize(P_32 width, P_32 height)

Get the current size of the current full page. The parameters are the addresses of two _32 values that will get set with the current height/width. If NULL is passed the respective value is not returned.

Afterword

InterShell vs PSI

This is not really a versus situation, but there are parallels that have conflicts.

PSI Controls

- are passed a PSI_CONTROL which is then dereferenced to get the user's attached data.
- Control data has a unique Type, which allows additional methods to be defined that can check validity of the control passed.
- PSI Controls are primitive, and allow tangible relationships of parent, child, sibling, with child controls being within parent controls.
- Configuration saved in XML.

InterShell Controls

- are passed their user data as a PTRSZVAL which they can immediately re-cast to their user type
- Control data passed through PTRSZVAL is available and known, but Additional methods are not passed a PMENU_BUTTON, so the base structure is known always. Each entry point is for a specifically named class, and no additional methods are available in the same way.
- Only one level of parent is known, the PAGE it is on.
- Configuration saved in clear text… fprintf output.

Using the InterShell surface control as a widget itself.

Widget name: "Menu Canvas"

PSI – during frame editing it is possible to include a "Menu Canvas" control within a dialog.

InterShell plugin – Magic, Create a control with it. Maybe I could do 'create sub-canvas' as a menu option.

The name defined above is the most useful thing.

MakeNamedControl(container, "Menu Canvas", x_offset_in_container, y_offset_in_container, canvas_width, canvas_height);