form•Z

SDK Documentation

Extensions in form•Z

1.0 Introduction

With version 5.0 **form•Z** first introduced the ability to add external functionality through extensions, which can be **plugins** or **scripts**. A plugin is written in the C or C++ computer language and compiled into a shared library (Macintosh) or a dynamic link library (Windows). These libraries are referred to as the plugin files and they are identified by the **.fzp** file extension. A script is written in the **form•Z** script language (FSL) and identified by the **.fsl** file extension. Scripts are compiled into binary files identified by the **.fsb** file extension.

There are 9 types of extensions: **attributes**, **file translators**, **object types**, **renderers**, **commands**, **palettes**, RenderZone **shaders**, **tools**, and **utilities**. All of these are available for plugin development. The latter five are also available for scripts. The rest are too complex for a script to be able to handle.

The **form•Z** SDK documentation consists of 5 chapters. This first chapter is the introduction and discusses issues that affect both plugins and scripts. The second chapter discusses plugin implementation and the third chapter discusses script implementation. Chapters four and five are reference manuals available only in an “on line” html form. These manuals can be viewed and searched using a web browser. We recommend using Internet explorer on Windows and Safari 1.2 (or later) on Macintosh OS X. Chapter 4 contains the reference for the **call back functions** discussed in the implementation chapters (2 and 3). Chapter 5 contains the reference for all the **API functions** that **form•Z** provides for extensions to use.

It is not necessary to read the complete documentation. Which parts one reads depends on whether he/she in interested in developing plugins or scripts. Everybody should read chapter1. Then, plugin developers should read chapter 2, while script developers should read chapter 3. From there on, it should not be necessary to exhaustively read chapters 4 and 5. One should scan through them so that he/she gains a general familiarity with the material to be able to use it as reference for specific functions that need to be called for a task. In all cases, the sample code should prove very effective, especially during the early stages of one’s involvement with the API or script development process. We actually expect that many new plugins and scripts will be developed by simply changing existing sample code.

Stylistically, different fonts are used throughout the first three chapters to distinguish text from examples or keywords in the API. The **Chicago** font is used to identify interface elements in the **form•Z** application. The `Courier` font is used to distinguish keywords and sample code of the **form•Z** SDK.

As already mentioned, sample plugins and scripts that are installed along with the **form•Z** SDK complement this documentation. These can be very valuable as both starting points for development as well as examples of how the **form•Z** API works. The Sample plugin files (.fzp) can be found in the *<formZ application>*\plugins\Samples\ folders and the source files (.c) for the samples can be found in the *<formZ application>*\formZ SDK\Samples\ folder . The sample scripts (.fsl and .fsb) can be found in the *<formZ application>*\Scripts\Samples\ folder.

1.1 How do extensions work

form•Z automatically recognizes extensions by finding them in designated directories at startup. The extension search paths determine the locations that are searched. This is a list of directories on the computer's hard disk (or on the network). By default, **form•Z** looks for extensions in a directory called "Plugins" and in a directory called "Scripts" inside of the **form•Z** application directory. The extension search paths can be customized by the user in the **Extensions Manager** dialog accessed from the Extensions menu.

An extension connects to **form•Z** by providing a set of **call back functions**. These functions are called by **form•Z** to add the extension into the **form•Z** interface and to execute the functionality defined by the extension. Extensions can make use of existing functionality in **form•Z** using the **form•Z API (Application Programming Interface) functions**. This includes standard functionality, such as the **form•Z** interface manager, **form•Z** run time library, math functions, and data management. The core functions for the **form•Z** tool set are also available as **form•Z** API functions.

1.2 Similarities and differences between plugins and scripts

A script is in essence a simplified version of a plugin. It is intended for a novice programmer to get started in adding extensions to **form•Z** without having to set up a full C or C++ based development environment. Therefore, the language used in a script, the **form•Z Script Language** (FSL), follows the C language very closely. It offers the same basic data types, such as integer and floating point numbers, and the same basic statements, such as loops and expressions. Similar to a plugin, a script is organized into functions, which are called from **form•Z** and which execute the respective functionality of the extension. As in a plugin, a script can call the majority of the **form•Z** supplied API functions. A good reason for choosing to develop an extension with a script instead of a plugin would be for an advanced programmer to try out the basic functionality of an extension with a script. This can be done, in general, very quickly. Once the try out stage has been completed, it is fairly straight forward to convert the C like FSL code of the script to real C or C++ code for use in a plugin.

There are a number of differences between plugins and scripts which are important to understand before deciding which type of extension is appropriate for a given task. In general scripts are quicker and easier to develop, however, they offer less functionality and slower performance than plugins. Professional developers are expected to use plugins more frequently while the casual developer are expected to favor scripts. Due to their similar nature it is not difficult to transform a script into a plugin. The following are the issues to consider:

Functionality

Some functionality is not available to scripts either because the complexity of the interface can not be accommodated in the script language or the nature of the task is such that the performance of scripts makes them unrealistic. Commands, shaders and utilities can be developed as plugins and as scripts while attributes, file translators, object types, and renderers can only be developed as plugins. Some **form•Z** API functions are also not available to scripts for similar reasons. The **form•Z** API reference indicates for each function if it is available for plugins or scripts (or both). There are a few functions that are available for scripts only.

The biggest difference in the available API functions is in the interface functions for building dialogs and palettes. The scripts have a simplified interface that limits the ability of a script to generate certain complex interfaces. See section 1.4.6 for more details on the interface.

Ease of use

Script development is self-contained within **form•Z** using the **form•Z** script editor while plugin development requires a third party compiler. Script files are cross platform and do not require separate compilation for each platform as is required for a plugin. The compiled script binary files (.fsb) can be used on either Macintosh or Windows.

Performance

Scripts are always slower that plugins. Plugins perform better because they are complied in native machine code and can be optimized for the processor. This performance difference may or may not be significant, depending on the task that is being performed by the extension. A task that does not contain a lot of computation in the script itself but rather calls a number of **form•Z** API functions will not perform as badly as one that does heavy computation in the script.

1.3 form•Z menu commands that support extensions

The File menu

New Script

A new menu item that has been added as the fourth item in the **File** menu. When selected, it opens a new script editor window and makes it the active window. For details on script editing and the script edit environment, see section 3.7.1.

Open

This menu can now be used to also open script files (.fsl). When a script file is selected from the standard file open dialog, a script editor window is opened and becomes the active window. The contents of the script file (.fsl) are shown in the window. For details on script editing and the script edit environment, see section 3.7.1.

A number of other items in the **File** menu are now sensitive to the script editing environment as described in section 3.7.1.

The Edit menu

Plugins And Scripts

This item has been eliminated from the **Edit** menu. It has been renamed **Extension Manager** and moved to the top of the new **Extensions** menu.

The Display menu

Renderer extensions now appear in their own group, which is the 4th group of the **Display** menu.

The **Palette** menu

Palette extensions now appear in their own group, which is the 3rd group of the **Palette** menu.

The **Extensions** menu

This is a new menu added between the **Palettes** and the **Help** menu. It contains 4 items in the top group. The remainder of the menu may contain additional items or hieractal menus created by extensions. Selecting one of these items performs the corresponding extension defined action.

Extensions Manager...

This item invokes the **Extensions** dialog. This is the same as the previously available **Plugins And Scripts** dialog.

Run Utility...

This item is used to run utility extensions. Utility extensions are designed to execute a task which is either less frequently used or it is not desired to have a menu item for the task appear in the **form•Z** interface. Utility plugins are best used on tasks that are linear in nature (like batch processing). Utility plugins are not loaded by **form•Z** at startup and are not listed in the **Extensions** dialog.

When the **Run Utility...** item is selected, a standard file open dialog is invoked to select the extension file to run. A utility can be a plugin file (.fzp) or a script file (.fsl). Once the file is selected, the utility is executed.

Run Recent Utility

This pop-out menu lists the most recent utilities that were executed using the **Run Utility...** command. Selecting the utility file name from the menu immediately executes the utility.

Enable Script Debugger

This item enables and disables the script debugger. For details on debugging scripts, see section 3.7.2.

1.4 Common concepts

There are a number of common concepts that affect plugins and scripts. These are discussed in the following sections.

1.4.1 UUIDs

A Universal Unique Identifier (UUID) is a 16-byte string that is generated using an algorithm that guarantees a unique sequence of bytes (string). These ids are unique no matter what machine they are generated on. UUIDs are used throughout **form•Z** for uniquely identifying items and avoiding naming collisions. The use of UUIDs guarantees that plugin developers will not create identically named plugins or collide with any of the names used by **form•Z**.

A UUID can be represented as a series of formatted hexadecimal numbers in between braces (e.g. “{72c37192-25fc-4443-9bdc-4613a4933764}”). It can also be represented as a string of hexadecimal characters using the \x escape sequence (e.g. "\x72\xc3\x71\x92\x25\xfc\x44\x43\x9b\xdc\x46\x13\xa4\x93\x37\x64"). **form•Z** expects UUIDs in the latter format. The functions `fzrt_UUID_to_string` and `fzrt_string_to_UUID` are provided for converting between the UUID formats.

A utility is provided for generating UUIDs. The utility is in the form of a plugin called **formZ UUID Generator**. This plugin is installed with the **form•Z** SDK sample plugins. To run the plugin, select **Run Utility...** from the **Extensions** menu and navigate to the /Plugins/Samples/Utilities folder and select the formZ UUID Generator.fzp plugin. The UUID utility generates a UUID and displays it in a dialog. The dialog contains two text fields that show the generated UUID in both the traditional format and the hexadecimal format used by **form•Z**. The text of the UUID can be copied from these text fields and inserted in script files or plugin source code as needed. Pressing the **Generate** button at the bottom of the dialog can generate a different UUID. The text fields are updated to contain the new UUID. Selecting **OK** closes the dialog and terminates the UUID generator utility.

1.4.2 form•Z resource files

The **form•Z** resource file format is an ASCII text file that stores the interface strings (resources) for use in **form•Z** and extensions. The interface strings are stored in these external files, rather than in the code itself, so that they can be localized. This avoids the significant overhead of generating separate applications or extensions for each supported language. The resource file is essentially a repository of strings, designed by the extension author, which can be used whenever a string parameter is needed by a function in the **form•Z** API.

For example, the **FUIM** (**form•Z User Interface Manager**) functions (see section 1.4.6) require a string for the title or text of most interface elements. To create a check box in a dialog, the title for the check box must be provided by the extension. The string could be embedded directly in the function call as in the following plugin example that creates a check box named **My Option**:

```
fz_fuim_new_check(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
            FZ_FUIM_FLAG_NONE, "My Option", NULL, NULL);
```

While this meets the requirements of the creation of the check box, it makes the string part of the extension and not easily localizable. The preferred solution is to store the string in a **form•Z** resource file. The above example could be stored in the file “`my_plugin.ENU.fzr`” as follows:

```
FZRF, 40, CHAR=MAC,

STR#, 1,
"My Option",
FZND,
```

Note that the name of the fzr file must follow the format described below. To use the resource file, when the check box is created, the string is loaded from the resource file and then passed onto the check box creation function as follows:

```
char                my_str[256];
fzrt_fzr_ref_td     rsrc_ref;
fzrt_error_td       err = FZRT_NOERR;

if((err = fzrt_fzr_open(floc, "my_plugin", &rsrc_ref)) == FZRT_NOERR)
{
```

```
    fzrt_fzr_get_string(rsrc_ref, 1, 1, my_str);
    fz_fuim_new_check(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
              FZ_FUIM_FLAG_NONE, my_str, NULL, NULL);

    fzrt_fzr_close(rsrc_ref);
}
```

In this example `floc` is a reference to the location on the disk where the resource file is located. The recommended location is the same folder as the extension file (.fzp or .fsl/fsb). The plugin function `fzpl_plugin_file_get_floc` and the script function `fz_script_file_get_floc` can be used to retrieve the location of the extension on the disk at runtime. These functions are discussed in chapters two and three respectively. Note that, since most extensions will contain multiple strings that are loaded at various times, it is more efficient to open the resource file once in the initialization of the extension and close it when the extension is terminated.

Localization is achieved by the creation of a parallel .fzr file for each desired language. **form•Z** automatically loads the translated string based on the language that is in use. The above fzr file would be translated into the file “`my_plugin.ESP.fzr`” for a Spanish translation as follows:

```
FZRF, 40, CHAR=MAC,

STR#, 1,
"Mi Opción",
FZND,
```

File format

The resource file uses the comma-separated value (CSV) file format that consists of contiguous, sequential blocks of resource data. The CSV specification is a simple set of rules used to organize text data in ASCII files. Each text field may be enclosed in quotes but it is not required. If there is a quote character (") within a text field, it is represented with two quotes (""). Commas separate each field of a record. The following is a sample fzr file that contains a single string list.

```
FZRF, 40, CHAR=MAC,

STR#, 1,                            /* Star Strings     */
"Script Star",                          /* Command name     */
"Help for Script Star",             /* Help string          */
"Script Star Options",              /* Tool options name    */
"Click a point to place star.",         /* Prompt           */
"Base Type",
"Radius",
"Ray Ratio",
"Dynamic",
"Preset",
FZND,
```

form•Z resource files are cross platform, hence they can be created on Macintosh or Windows and used on either. To support proper translation between character sets on the different platforms, the file must contain information on the platform that the file was created on (see FZRF header below). **It is important that a file not be edited on different platforms**. This will result in mixed character sets within the file and this is not supported. A file can be converted to a system’s native format by using the function `fzrt_fzr_write_file` to write a clean copy of the file.

The function `fzrt_fzr_open` opens a fzr file and returns a runtime index for the file. This index is used in all other functions to reference the file. `fzrt_fzr_close` should be called when the file is no longer needed. Note that, for efficiency, parts or all of the fzr files are cached once they are opened. It is not efficient to frequently open and close fzr files. It is recommended that an extension only open and close the file once. There are specific functions for reading each type of resource stored in the file.

form•Z resource file names must have the format *filename.<LANG>.FZR*. The file extension is ".FZR" and on a Macintosh the file should have the Finder type, 'TEXT'. *<LANG>* is a 3-character constant that defines the language of the file. The following are currently supported:

`CHS` - Simplified Chinese
`CHT` - Traditional Chinese
`DEU` - German
`ELL` - Greek
`ENU` - U.S. English
`ESP` - Spanish
`FRA` - French
`ITA` - Italian
`JPN` - Japanese
`KOR` - Korean

For example:

"QTVR.ENU.FZR" is the complete filename for the U.S. English version of the QuickTime VR **form•Z** resource file, and

"QTVR.ESP.FZR" is the complete filename for the Spanish version of the QuickTime VR **form•Z** resource file.

form•Z looks for resource files using the language identifier based on the current language as selected in the **Language** preference section of the **Preferences** dialog. For example if Spanish is the current language, then **form•Z** will look for resources in files that end in .ESP.fzr. Localization is supported using parallel files. That is, for each supported language there is a file with the same file name, but different language identifier, which contains the same resource data in the corresponding language. Note that if a resource is not found in the file corresponding to the current language, **form•Z** will look for an English resource in a file (.ENU.fzr). When opening resource files using the `fzrt_fzr_open` function, only the base file name should be used (i.e. without the language identifier and .fzr extension).

The content of a resource file is organized in **blocks**, which can be of different types and are properly identified.

Block format

Each block in a resource file has three components: a **block type identifier**, a **block ID**, and the **block data**. A C language style comment enclosed by "/*" "*/" may be included within the block definition.

The block type is a 4-character constant in the first CSV field of the block. These are predefined names. There is currently no accommodation for user-defined block types. The supported types of blocks are described below.

The block ID is the numeric ID of the specific block type. This is an ASCII string of a decimal number (e.g. 123) in the second CSV field of the block. Block IDs must be unique within a file for all blocks of the same block type.

The block data is the resource or string data. The format of the data is dependent on the type of block as described below. Inclusion of a comment within the block data is allowed only if the block type specification allows it.

Block specifications

FZRF (form•Z resource format header)

The first block of any **form•Z** resource file is the FZRF block. If this block is not found at the top of the file, the fzr file will not be recognized as a valid **form•Z** resource file. The block ID for the FZRF block tells **form•Z** what version of the **form•Z** resource format specification is used in the file. This number should always be 40. The data for this block is a sequence of CSV fields in the format of keyword=value. The following keywords are supported.

`CHAR`	Platform on which the file was created. Along with the language identifier in the filename, this determines the character set that is used in STR# and MENU resources. This is required. Identifiers recognized are: `MAC` for Macintosh and `WIN` for Windows.
`FVER`	Identifies the version of the file. This is the version of the data, not of the file-format. This is ignored by resource functions, but available through the function fzrt_rsrc_get_info;

A sample **form•Z** resource format header block looks like this:

```
FZRF, 40, CHAR=MAC,
```

STR# (string array)

This block defines an array (list) of strings. The block data contains a list of strings with each string as a CSV field. The list must end with FZND as a marker to the end of the list. Hard returns (new lines) are permitted within a string. Comments may be included within the CSV list of strings. A string value which contains the sub-string "/*" should be enclosed with single quotes so it is not interpreted as a comment. For example:

```
STR#, 100,
string 1 /* first string with comment following*/,
"string 2 /* with comment-like string */",
the last string,
FZND,
```

Items from STR# resources are accessed in extensions using the function `fzrt_fzr_get_string`. The following plugin example shows the loading of a string for the creation of a check box:

```
char                my_str[256];
fzrt_fzr_ref_td     rsrc_ref;
fzrt_error_td       err = FZRT_NOERR;

err = fzrt_fzr_get_string(rsrc_ref, 100, 1, my_str);
err = fz_fuim_new_check(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
             FZ_FUIM_FLAG_NONE, my_str, NULL, NULL);
```

MENU (string array)

This block defines a menu resource. The block data contains a list of menu item strings with each string as a CSV field. The list must end with FZND as a marker to the end of the list. Each string is interpreted as the text for a single menu item. The special string "-" is interpreted as a menu separator, which is the horizontal line dividing a menu to sections. The first string in the list is the menu title. Comments are supported as with the STR# block type. For example:

```
MENU, 101,
menu title,
menu item 1,
menu item 2,
"-" /* separator */,
menu item 3,
FZND
```

Menu resources are accessed by an extension using the function `fzrt_fzr_get_menu`. Note that this function loads all of the items in the list into the menu rather than just one string as with `fzrt_fzr_get_string`. The following plugin example shows the loading of a menu for the creation of a menu in a FUIM template:

```
fzrt_menu_ptr       my_menu;
fzrt_fzr_ref_td     rsrc_ref;
fzrt_error_td       err = FZRT_NOERR;

err = fzrt_fzr_get_menu(rsrc_ref, 101, &my_menu);
err = fz_fuim_new_menu(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
             FZ_FUIM_FLAG_NONE, my_menu, NULL, NULL);
```

1.4.3 Platform detection

form•Z runs on Macintosh computers on the OS X operating system and on PC's running Windows. The **form•Z** API is cross platform, however, there are occasions when it is necessary to know what platform the extension is running on. Plugins and scripts can detect the platform at runtime using the function `fzrt_get_platform`. This function returns FZRT_PLATFORM_MAC or FZRT_PLATFORM_WIN. The function `fzrt_get_os_attrs_func` can be used to get specific information about the version of the operating system.

```
If(fzrt_get_platform()==FZRT_PLATFORM_MAC)
{
   /* Macintosh specific code goes here */
}

If(fzrt_get_platform()==FZRT_PLATFORM_WIN)
{
   /* Windows specific code goes here */
}
```

Plugins can use compile time platform detection by checking the `__PLATFORM__` macro, which is defined in the header file platform. This has the value of `__PLATFORM__ == MACINTOSH` for Macintosh compilation and `__PLATFORM__ == WINDOWS` for Windows compilation.

```
#if (__PLATFORM__ == MACINTOSH)
   /* Macintosh specific code goes here */
#endif

#if (__PLATFORM__ == WINDOWS)
   /* Windows specific code goes here */
```

```
#endif
```

The form•Z SDK is 100% cross platform compatible. There are a few functions that do not function on one platform or the other due to lack of support in the operating system. These functions are provided so that plugin source code can remain platform independent. These functions are documented in the form•Z API reference (chapter 5).

1.4.4 Memory allocation

The **form•Z** runtime library (fzrt) provides a memory manager that is more efficient than the operating system (or ANSI) allocations (malloc, NewPtr etc…). The operating system must manage all of the blocks of memory allocated by an application all at once. When a large number of blocks are allocated, the time to perform an additional allocation or free can be significantly slower than the same operation when very few blocks are allocated. This is because each block of memory that is allocated must be tracked and managed adding additional overhead to the management.

The **form•Z** runtime library memory manager uses a concept called memory zones. Memory zones work by allocating larger (but fewer) blocks of memory from the operating system (a memory zone) and managing zone blocks individually. The memory zone that is being used must be provided when allocating or freeing a block of memory. **form•Z** creates a number of zones that are used for allocating memory for areas of related functionality.

Memory zones can be static or dynamic. Static zones are created with a fixed size. Multiple blocks can be allocated from a static zone but the total size of all of the requested blocks can not exceed the zone size. Dynamic zones are unbounded in their size and can grow to consume all available memory.

It is strongly recommended that plugins use memory zones when allocating more than 2 persistent memory blocks during the life of the plugin. This will not only make the plugin perform better, but it will keep the plugin from harming the overall performance of **form•Z**. Memory zones are best used for blocks that are persistent. Memory that is needed for a local or temporary context should be allocated and freed using `fzrt_new_ptr` and `fzrt_dispose_ptr` functions.

1.4.5 Units

Internally to **form•Z** the number 1.0 is an inch or a centimeter (cm). If the working units are set to English units, then 1.0 = 1 inch and if they are set to Metric units, then 1.0 = 1 cm. The working units dialog presents a number of options to the user for controlling unit display. All of these options are just display options to the user. The **form•Z** user interface manager (fuim) automatically converts displayed numbers to and from the base unit (inch or cm).

Angles are stored in radians where 1.0 = 1 radian. Angle values can be converted between radians and degrees using the macros FZ_MATH_DEG_2_RAD and FZ_MATH_RAD_2_DEG. The working units dialog presents a number of options to the user for controlling angle display. All of these options are just display options to the user. The **form•Z** user interface manager (FUIM) automatically converts displayed numbers to and from radians.

1.4.6 Interface

The **form•Z** API includes support for common interface features such as dialogs, alerts, palettes, wait cursor, key cancel detection and progress bars. The **form•Z** user interface manager (FUIM) manages these interfaces. The prefix `fz_fuim_` is used for all of the FUIM API entities (functions, types, constants etc.).

The layout of interface elements (buttons, menus, text, etc.) found in dialogs and palettes is called a **FUIM template**. The template contains the definition of the interface elements, the definition of dependencies between the elements, and the connection to data storage (variables) in the extension. The **form•Z** template manager handles the graphic layout of the template automatically and deals with all platform specific issues. The template definition is hierarchically organized in the form of a tree. That is, each element has a parent element and may have multiple sibling elements and child elements. The interface elements are implicitly dependent on their parent. That is, if the parent element is disabled, all of its descendents are also disabled.

The FUIM template capabilities are very different between plugins and scripts. The script FUIM templates are simpler to use but do much less. The plugin templates can do all that the scripts can do plus much more; however, they are a bit more complex to use. The following are the significant limitations of script FUIM templates:

- Dependency of elements in the template can only be described through the implicit hierarchy of the elements. Plugins can define additional dependency independent of the hierarchy tree.
- Complex interface elements like lists (such as those used in **form•Z** in the Objects and Lights palettes and in dialogs) and previews (such as those used in **form•Z** in the Sweep, Revolve, and Rounding tool dialogs) are not available to scripts.
- Scripts can not create custom interface elements. Custom elements are interface elements that allow virtually total customization of the interface by a plugin.

Templates are defined through a **FUIM template function** that is provided to **form•Z** by the extension. The template function defines the template by calling **form•Z** API functions to create the interface elements, define relationships between items, and bind the data storage (variables) from the extension to the elements. The template function is provided to **form•Z** when a dialog is invoked through a dialog driver, or through specific call back functions provided by **form•Z**. These call back functions vary by the type of extension and are discussed in section 2.8 for plugins and 3.7 for scripts.

The details of using the FUIM manager are discussed separately for plugins and scripts. The Plugin description is in section 2.6 and scripts in section 3.5.

1.4.7 Naming conventions

There are a number of naming conventions used throughout the **form•Z** SDK to make the code easier to use and understand. In general, names are descriptive in nature. To keep the names from becoming too long, each term in a name is abbreviated, if it is more than six characters in length. Each term in a name is separated by an underbar "_" character.

The naming convention is also used to group similar things together. For example all of the functions that manage **form•Z** modeling objects start with `fz_objt_`. All the object creation functions start with `fz_objt_cnstr_` and the editing functions with `fz_objt_edit_`. Terms relative to the content of an entity follow these to complete the names. These include the different **form•Z** object types and operations. For example, `opts` stands for *options* and `parm` for *parameter*. The following example shows the names for the functions corresponding to the **form•Z** Sweep tool.

```
fz_objt_cnstr_sweep_axial
```

```
fz_objt_cnstr_sweep_2source
fz_objt_cnstr_sweep_2path
fz_objt_cnstr_sweep_boundary
fz_objt_cnstr_sweep_opts_init
fz_objt_cnstr_sweep_opts_get
fz_objt_cnstr_sweep_opts_set
fz_objt_cnstr_sweep_opts_finit
fz_objt_edit_sweep_parm_get
fz_objt_edit_sweep_parm_set
```

The following additional rules are applied:

• All names defined in the **form•Z** SDK start with fz_, fzrt_ or fzpl_. For example:

```
fz_objt_cnstr_sweep_axial
fzrt_boolean
fzpl_plugin_add_fset
```

• Functions, enumerators (enum in C), type names (typedef in C) are in lower case letters . For example:

```
fz_objt_cnstr_sweep_axial
fz_objt_sweep_type_enum
```

• Structure names end in _td (in C all structures are defined by typedef's). For example:

```
fz_xyz_td
fz_rgb_td
```

• Function types end in _func. For example:

```
fz_fuim_item_func
fz_fuim_item_cust_func
```

• Pointer types end in _ptr. For example:

```
fzrt_floc_ptr
fz_fuim_tmpl_ptr
```

• Enumerated lists (enum) end with in _enum. For example:

```
fz_fuim_icon_enum
fz_objt_sweep_type_enum
```

• Constants (#define in C) are in all capital letters. For example:

```
FZRT_NOERR
FZPL_VERS_MAKE
```

• Members of enumerated lists (enum) are in all capital letters. For example:

```
FZ_OBJT_SWEEP_TYPE_AXIAL
FZ_OBJT_SWEEP_TYPE_2SOURCE
FZ_OBJT_SWEEP_TYPE_2PATH
FZ_OBJT_SWEEP_TYPE_BOUNDARY
```

1.4.8 Error handling

The API and call back functions provided by form•Z for both plugins and scripts perform rather extensive checks in order to protect the system from crashes that may be caused by bad data or other undesirable conditions. When improper conditions are encountered, an error message is returned by the function and needs to be properly handled by an extension. How this ought to be done is discussed in this section.

Errors generated by form•Z API functions

With a few exceptions almost all **form•Z** API functions return an error code. When writing a plugin, the error code is of type `fzrt_error_`, which is defined as a `long` integer. When writing a script, the return type is a `long` integer. Note that there is no real difference between these two declarations, since `fzrt_error_` is essentially a long. Thus functions that return error codes are typically declared as long, which works fine with assignments to both long and `fzrt_error_` variables.

When an API function executes successfully, ity returns an error code of `FZRT_NOERR`, which is defined as `0` (zero). If it does not return `FZRT_NOERR`, some error occurred in the API function and it returns the respective error code. When an extension calls a **form•Z** API function, it is recommended to always check for errors and to structure the flow of the code accordingly, as shown in the following plugin example:

```
fzrt_error_td rv;
fz_xyz_td wdh,origin;
fz_objt_ptr   obj;

    wdh.x = 10.0;
    wdh.y = 10.0;
    wdh.z = 50.0;

    origin.x = 100.0;
    origin.y = 100.0;
    origin.z = 0.0;

    rv = fz_objt_cnstr_cube(windex,&wdh,&origin,NULL,&new_obj);
    if ( rv == FZRT_NOERR )
    {
        rv = fz_objt_add_objt_to_project(windex,new_obj);
    }
```

Note that a script example would be identical with the above plugin example, except for the declaration of the error variable rv, which would be:

`long   rv;`   instead of   `fzrt_error_td   rv;`

In the example above, of course, there is little chance for an error, because the input parameters to the `fz_objt_cnstr_cube` function are hard-coded values, which will always succeed. The only possibility for an error would be if the system runs out of memory. In other instances, however, the input parameters for API functions may come from other sources, such as user input. In these cases, the values may not always be clean and error checking becomes important for the stability of the plugin or script.

Errors generated by callback functions

In general, it is not that critical for an extension to know what kind of error occurred. What is more important is to pass back to **form•Z** the error codes from API functions that are executed inside of an extension's callback function. As with the API functions, the error codes that callback functions return are assigned to `fzrt_error_td` variables in plugins or to `long` variables in scripts.

Depending on what kind of callback function is called, **form•Z** will take a different action, when an error occurrs. If the callback function is executed at startup time and an error occurs, the respective plugin or script will not be loaded. For example, a plugin may be opening a resource file in the tool's init function in order to extract the strings needed for dialogs. If the resource file cannot be found, an error is generated and should be passed back to **form•Z**. Below is such an example for a tool script.

```
long   fz_tool_cbak_init( )
{
   long         err = FZRT_NOERR;
   fzrt_floc_ptr    floc;

   if ((err = fzrt_file_floc_init(floc))  == FZRT_NOERR &&
       (err = fz_script_file_get_floc(floc)) == FZRT_NOERR )
   {
      err = fzrt_fzr_open(floc,"tool_star",star_rsrc_ref);
      fzrt_file_floc_finit(floc);
   }

   return(err);
}
```

Since the tool cannot exist without strings, if the resource file with the strings is not found and `fz_tool_cbak_init` returns an error, **form•Z** will not load the tool script.

For other callback functions, **form•Z** may not execute a certain functionality, if an error occurs. That is the case, for example, when the copy attribute callback function returns an error. The attribute simply will not be copied.

If an error occurs in one of the major call back functions, listed below for the different extension types, **form•Z** will post an error message in a dialog.

Tool
   `fz_tool_cbak_prompt`
   `fz_tool_cbak_click`
   `fz_tool_cbak_select`

Project command
   `fz_cmnd_cbak_proj_select`

System command
   `fz_cmnd_cbak_syst_select`

Project utility
   `fz_util_cbak_proj_main`

System utility
   `fz_util_cbak_syst_main`

File translator
   `fz_ffmt_cbak_basic_read`
   `fz_ffmt_cbak_basic_write`
   `fz_ffmt_cbak_data_model_read`
   Any of the data model write functions that write to the file.

   `fz_ffmt_cbak_imag_vect_read_frame`
   `fz_ffmt_cbak_imag_vect_read`
   Any of the image vect write functions that write to the file.

   `fz_ffmt_cbak_imag_bmap_read_info`
      `fz_ffmt_cbak_imag_bmap_read`

Any of the image bmap write functions that write to the file.

Note that, for structured file translators , there are several callback functions that write data to a file. If any one of these functions returns an error, the export is aborted and the error is posted to the user.

form•Z will not post an error for functions not listed above. It is the responsibility of the plugin to inform the user of significant errors. This is described in more detail in the next section.

Retrieving more detail about an error

In some instances it may be desirable to post an error inside an extension. For example, when the initialization of a plugin at load time fails for very specific reasons, the plugin may choose to notify the user about the error. In this case, the return value of the **form•Z** API function that caused the error is passed into the API function `fzrt_error_get_info`, which returns error details. Among those is a string, which can be used in an alert dialog. For example :

```
long   fz_tool_cbak_init( )
{
   long          err = FZRT_NOERR;
   fzrt_floc_ptr    floc;
   string        str1,str2;

   if ((err = fzrt_file_floc_init(floc))  == FZRT_NOERR &&
       (err = fz_script_file_get_floc(floc)) == FZRT_NOERR )
   {
      err = fzrt_fzr_open(floc,"tool_star",star_rsrc_ref);
      fzrt_file_floc_finit(floc);
   }

   /* AN ERROR OCCURRED. POST IT TO THE USER */
      if ( err != FZRT_NOERR )
   {
         strcpy(str1 ,"Unable to open tool_star resource file. Reason :");
      fzrt_error_get_info(err, str2,255,NULL, NULL,NULL, NULL,NULL,0, NULL);
         strcat(str1,str2);
            fz_fuim_alrt_std_confirm(str1, FZ_FUIM_ALRT_CONFIRM_OK);
      }

   return(err);
}
```

In the second part of the above call back function, code is provided for properly issuing a message, if the value of `err` is not `FZRT_NOERR`. First, `strcpy` (the string copy) function assigns a string to `str1`. Note that this string contains an incomplete message and needs some explanation of a reason to be added to its end. The explanation string is picked by the `fzrt_error_get_info` function and stored in string variable `str2`. Next, `strcat` (the string concatenate function) concatenates `str2` into `str1`. The message is now complete and `fz_fuim_alrt_std_confirm` displays it in a dialog.

Care should be taken when posting errors. Only significant errors should be posted and they should be posted once only. For example, if an error occurs inside a loop, the alert dialog should not be posted for each iteration of the loop, even if the error occurs multiple times. This would require the user to repeatedly hit the **OK** button in the alert dialog. As a rule of thumb, an extension should post those errors that are not obvious to a user. The posted message should assist the user in not repeating the error a second time.

Errors created by a plugin

A plugin can define its own error codes and associated strings. This allows **form•Z** to post the appropriate error message, if a plugin fails, not because an error was returned by an API function called by the plugin, but because of an error condition that occurred directly in the plugin. To facilitate this **form•Z** defines an error context specifically for plugins, `FZRT_ERROR_CONTEXT_PLUGIN`. Plugin defined error codes can be any long integer value. When setting an error to a plugin defined error code, `fzrt_error_set` or `fzrt_error_set_with_detail` should be called with the `err` parameter set to the plugin defined error code, the `context` parameter set to `FZRT_ERROR_CONTEXT_PLUGIN`, and the `context_id` parameter set to the plugin's runtime ID, as follows:

```
#define MY_ERROR_CODE   1

err = fzrt_error_set(MY_ERROR_CODE, FZRT_ERROR_SEVERITY_ERROR,
                           FZRT_ERROR_CONTEXT_PLUGIN, plugin_runtime_id);
```

In order for **form•Z** to supply a string to an error alert dialog, the plugin must supply a function which maps the error code to a string. This is an example of such a function:

```
#define MY_MESSAGES 1

fzrt_boolean  my_error_str_func(
             long   err,
             char   *str,
             short  str_len)
{
   char   msg[256];

   /* Get the string from the plugin's resource file. */
   fzrt_fzr_get_string(_fz_rsrc_ref, MY_MESSAGES, err, msg);
   strncpy(str, msg, str_len);

   /* If we sucessfully got the string, return TRUE; otherwise, return FALSE.
*/
   return(msg[0] == '\0' ? FALSE : TRUE);
}
```

The `err` parameter is the plugin defined error code, `str` is the string explaning the error, and `str_len` is the maximum length of `str` in bytes. Error strings should be stored in **form•Z** resource (.fzr) files, so that they may be localized easier.

The `my_error_str_func` function is registered with **form•Z** through the `err_str_fcn` parameter of `fzpl_plugin_register` as follows:

```
err = fzpl_plugin_register(
          MY_PLUGIN_ID,
          my_plugin_name,
          MY_PLUGIN_VERSION,
          MY_PLUGIN_VENDOR,
          MY_PLUGIN_URL,
          FZFN_TOOL_TYPE,
          FZFN_TOOL_VERSION,
          my_error_str_func,
          0,
          NULL,
          &plugin_runtime_id);
```

Error Logging

When `fzrt_error_set` or `fzrt_error_set_with_detail` are called, the error is logged to the file, "formz log.txt" in the **form•Z** application folder. This provides a history of all errors that occurred whether they result in an error being displayed to the user or not. For each error, this file is opened, the error is appended to the end of the file, then the file is closed. This makes sure the file is written to disk in the case of a crash. This file is not deleted between runs of **form•Z**. Therefore, it is a good idea to periodically delete this file. If the file doesn't exist, **form•Z** will create a new one.

This file contains the time the error was set, the error string, the error severity, the error context and context id and the error code. A single entry in the error log looks something like this:

```
Tue Apr 27 12:51:28 2004
    69533)  An object with no faces encountered.
        error code = 4 severity = Error context = Plugin:Wavefront OBJ File
translator              detail id = 1108
```

The first line contains the date and time when `fzrt_error_set` or `fzrt_error_set_with_detail` was called. The second line contains the id of the error (the falue returned from `fzrt_error_set` or `fzrt_error_set_with_detail`) and the error string. The third line (which wraps around in the above example) contains the error code (passed into `fzrt_error_set` or `fzrt_error_set_with_detail`), the severity, the context and the detail id which is only set by `fzrt_error_set_with_detail`. If `fzrt_error_set` was called, the detail id will be 0. The Wavefront File Translator sample plugin assigns a unique detail id for each error it generates. This can help identify where in the code an error occurred.

1.5 Data organization

Information in **form•Z** is divided between system and project. System data is global and does not change regardless of which project is active. The preferences, key shortcuts, and tool options are examples of system information. Project information has an instance of the data per project. The user interacts with the project information of the active project through the **form•Z** interface. The working units and project colors are examples of project level information.

Some project information has a single instance for the entire project and other information has an instance per project window. The working units and project colors are examples of a single instance. When these options are changed, all windows of the project are affected. The window options, rulers, underlay and display options are examples of information that has an instance per project window. That is, each window maintains its own individual setting for these options.

Extensions can get and set a variety of data and perform operations at the system and project level through **form•Z** API functions. **form•Z** project data is referenced from the runtime index of a **form•Z** project window. This parameter is always called **windex** and is present as the first parameter in many of the **form•Z** API functions. Project level call back functions receive the windex parameter from **form•Z**. The plugin or script should use this value when calling any **form•Z** API function that requires a windex parameter. Since the windex value is a runtime value it will vary from session to session and will be different for each window of a project. This value should not be stored in a persistent fashion (global variable or file).

The provided windex will often be the windex of the active project window, however, this is not always the case as **form•Z** may perform operations on the non active project window. The windex parameter can represent a special project such as the clipboard which is a hidden project managed by **form•Z**. Multi-threading is expected to be supported in the future which will enable background processing and hence the windex parameter would represent a window that is being processed in the background.

System level plugins and scripts do not get the windex parameter, however, they can traverse the list of projects using the **form•Z** API functions, as shown in the following example.

```
fzrt_error_td my_walk_project_windows_func(void)
{
    long          windex,start_windex,nprojs,l;
    fzrt_error_td    err= FZRT_NOERR;

    if((err = fz_proj_get_count(&nprojs)) == FZRT_NOERR)
    {
        for(i=0;i< nprojs;i++)
        {
            if((err = fz_proj_get_windex(i, &windex)) == FZRT_NOERR)
            {
                start_windex = windex;
                do
                {
                    if((err = fz_wind_get_next(windex, &next) )) == FZRT_NOERR)
                    {
                        windex = next;
                    }
                } while(windex !=  start_windex && err == FZRT_NOERR);
            }
        }
    }
}
```

1.5.1 Model object representation

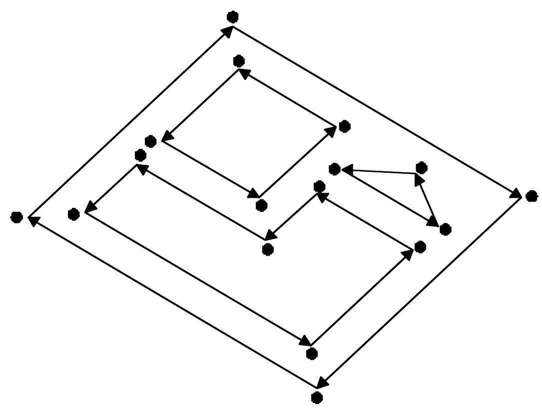
form•Z offers a large number of API functions which allow a plugin or script developer to construct and modify objects. In all of these functions, parameters are passed to the function, which describe the shape of the object to be constructed or the type of changes to be made to the object. In either case, the developer never has to worry about the actual structure of the object, as this is taken care of in the API function. For example, the API function that moves an object `fz_objt_edit_move_objt` not only moves the geometry of the object, but also moves associated object data, such as attributes with positional values, parameters of controlled objects etc. While using API functions is a safe way to create and edit objects, there may be instances, where the content of an object must be accessed directly. For example, there may not be an API function, which constructs an object of a particular shape, or there may not be an API function, which changes the shape of an object in a particular way. For these cases, **form•Z** offers API functions which give the developer direct access to the underlying data structures of an object.

Object topology and geometry

Before these API functions are discussed, it is necessary to understand the structure of a **form•Z** object. The user interface of **form•Z** already reveals the basic object structure. There are 5 levels of topology, which correspond to the 5 levels of the pick tool : **point**, **segment**, **outline**, **face**, and **object**. Note that the group level is not part of the object, as it is its own organizational level, which does not contain any geometric data.

Each object consists of one or more faces. A face, by its nature, is a closed shape. However, by convention, it may also be open, as in the case of an open line. If the object model type is facetted, all closed faces are assumed to define a plane. However, faces may also be non planar, in which case the actual plane definition is ambiguous. Such faces may be triangulated, which decomposes them into smaller planar faces. If the object model type is smooth, a face's underlying geometry may be a plane, a cylinder or cone, a sphere, a torus, or a spline surface. Open faces do not define a surface, and they are called **wires**. There may also be closed wire faces, which are faces of smooth objects, whose surface has been removed.

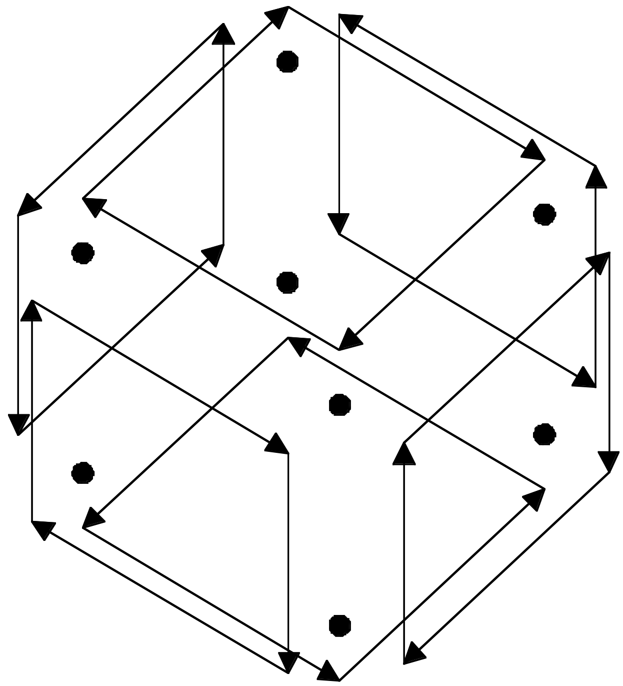
A face is defined by one or more outlines. The first outline is always the outer boundary of a face's geometric surface. If a face has more than one outlines, the remaining outlines define holes, which are contained within the outer boundary. The directions of the outlines are significant. The outer outline is always defined in clockwise direction, when looking down on the face's surface. All hole outlines have a counter clockwise direction. This is shown in the figure below.



The topology of a 2d surface with three holes

An outline is defined by a linked list of segments or edges. If the face is an open wire, there is a start and an end segment. Otherwise, the segments form a closed loop. If the object model type is facetted, a segment is always a straight line. In other words, the segment's geometry is defined by its start and end point and the line between those is assumed to be straight. If the object model type is smooth, the line between the segment's start and end points may be a straight line, an arc, or a spline curve. A segment can be thought of as having a direction, as it always point from its start point to its end point. The end point of a segment is always identical with the start point of the next segment. Segments also may have a coincident partner segment, which runs in the opposite direction. We call this a segment's reversely coincident segment. This is the case in an object that consists of several faces, which are stitched together, as in any solid object, for example. In such an object, a user will only see one segment, where there are really two segments in the object's data structure. Since they are occupying the same 3d space, only one can be shown. For example, a cube appears to have 12 segments for a user, but really has 24 segments in its data structure (6 faces with one outline each = 6 outlines with 4 segments each = 24 segments). Faces, which do not have a neighboring face are defined by segments, which do not have a reversely coincident segment. For example, a simple rectangle object has 4 segments.

The final topological level is the point or vertex. Each segment has a start and end point, which it references through an index. Note, that points are shared by segments. For example, a cube has 8 points, but 24 segments. That means, that three segments have the same start point. This can easily be verified, by drawing an exploded view of the topology of a cube object, as shown in the figure below.



The topology of a cube

Smooth versus facetted objects

The topology described above fits both smooth and facetted objects. By convention, in **form•Z**, smooth objects, in addition to their smooth topology, they also carry the corresponding facetted topology. In other words, they are stored with two representations. **form•Z** may use either topology, depending on the operation involved. For example, when a modeling operation is executed on a smooth object, the smooth topology is used. This allows, for example, a Boolean union between two smooth objects to result in a smooth object. When a smooth object is drawn on the screen, many rendering modes use the facetted topology or a combination of smooth and facetted topology. For example, Quick Paint shows only the facetted faces of a smooth object. Wireframe display draws the facetted faces in a lighter color and draws the edges of the smooth faces in a darker color. The facetted topology of a smooth object is always kept up to date by **form•Z** to faithfully represent the smooth shape of the object. The end user, is therefore never allowed to manipulate the facetted topology of a smooth object directly, as any change would be wiped out the next time **form•Z** regenerates the facets of a smooth object. Nevertheless, the facetted topology is completely defined in a smooth object and can be accessed by a plugin or script developer, as if the object were a facetted object.

1.5.2 Tracing the topology of an object

A plugin or script developer may want to trace the topology of an object for either extracting information from the various levels or for applying some opertation. **form•Z** offers a number of API functions for that purpose.

Tracing faces

To traverse all faces of an object, a simple for loop can be written. The example below extracts the geometry type of each face of an object. Note, that the function that retrieves the number of faces of an object takes an argument, which determines whether the facetted or smooth number of faces is retrieved. If `FZ_OBJT_MODEL_TYPE_UNSPEC` is passed for the argument, the function returns the number of smooth faces, if the object is smooth, or the number of facetted faces, if the object is facetted. If the object is smooth, it may be desirable to trace the facetted representation of the smooth object. In this case, the argument must be passed as `FZ_OBJT_MODEL_TYPE_FACT`. If the developer wants to trace the smooth faces of a smooth object, the argument can be set to `FZ_OBJT_MODEL_TYPE_SMOD` or `FZ_OBJT_MODEL_TYPE_UNSPEC` . For facetted object, the `FZ_OBJT_MODEL_TYPE_SMOD` is invalid and will result in an error code passed back by the API.

```
fz_objt_get_face_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&nface);

for(i = 0; i < nface; i++)
{
    /* GET THE GEOMETRY TYPE OF THE FACE */
        fz_objt_alys_get_face_geom_type(windex,obj,i,&geom_type);

        /* DO SOMETHING WITH IT */
        ...
}
```

Tracing outlines

To access the outlines of an object, two methods may be used. The outlines can be accessed directly from the object. Note that in the API, Outlines are referred to as **curves**. In this case, the loop looks similar to the face loop:

```
fz_objt_get_curve_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&ncurv);

for(i = 0; i < ncurv; i++)
{
    /* GET THE PERIMETER OF AN OUTLINE */
        fz_objt_alys_get_curve_circumference(windex,obj,i,&circ);

        /* DO SOMETHING WITH IT */
        ...
}
```

In the **form•Z** representations, all outlines belonging to the same face are linked with "previous" and "next" pointers. Each face also contains a pointer to its first outline, which is always an outer outline. This is linked to the remainder outlines, which are all holes, if they exist at all. The second method for tracing the outlines is based on this structure of the faces and is illustrated by the following example:

```
fz_objt_get_face_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&nface);

for(i = 0; i < nface; i++)
{
    /* GET THE FIRST CURVE OF THE OBJECT */
    fz_objt_face_get_cindx(windex,obj,i,
        FZ_OBJT_MODEL_TYPE_UNSPEC,&chead);
    cindx = chead;

    do
    {
        /* DO SOMETHING WITH THE CURVE */
        ...

        /* GET THE NEXT CURVE */
        fz_objt_curv_get_next(windex,obj,cindx,
            FZ_OBJT_MODEL_TYPE_UNSPEC,&cindx);

    } while ( cindx != chead );
}
```

Note, that the outline loop is a do while loop, as the outlines of a face form a circular linked list.

Tracing segments

As with the outlines, the segments of an object may be traced in two ways: directly or through the topology hierarchy. A direct loop is shown below:

```
fz_objt_get_segt_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&nsegt);

for(i = 0; i < nsegt; i++)
{
        fz_objt_alys_get_segt_length(windex,obj,i,&length);
}
```

To trace along the topological hierarchy, the nested outline loop from above is expanded with another nested loop for all the segments of each outline. This again is based on that all the segments of an outline are linked with each other and each outline contains a pointer to its first segment. The second method of tracing is illustrated in the following example:

```
fz_objt_get_face_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&nface);

for(i = 0; i < nface; i++)
{
    /* GET THE FIRST CURVE OF THE OBJECT */
    fz_objt_face_get_cindx(windex,obj,i,
        FZ_OBJT_MODEL_TYPE_UNSPEC,&chead);
    cindx = chead;

    do
    {
        /* GET THE FIRST SEGMENT OF THE CURVE */
        fz_objt_curv_get_sindx(windex,obj,cindx,
            FZ_OBJT_MODEL_TYPE_UNSPEC,shead);

        sindx = shead;

        do
        {
            /* DO SOMETHING WITH IT */
            ...

            /* GET THE NEXT SEGMENT */
            fz_objt_segt_get_next(windex,obj,sindx,
                FZ_OBJT_MODEL_TYPE_UNSPEC,sindx);

        } while ( sindx != shead && sindx != -1 );

        /* GET THE NEXT CURVE */
        fz_objt_curv_get_next(windex,obj,cindx,
            FZ_OBJT_MODEL_TYPE_UNSPEC,&cindx);

    } while ( cindx != chead );
}
```

Again, the segment loop is implemented as a do while loop, as segments are also forming a linked list. The list may be closed, if the curve is a closed curve. In this case, the next segment of the last segment of a curve points to the first segment of the curve. Thus the terminating condition `sindx != shead` of the while loop. A linked segment list may also be open, if the curve is an open curve. in this case, the last segment does not point to another segment, but returns -1 for the next segment index. The terminating condition for the do while loop is expanded with `sindx != -1`, for open curves.

There is one significant difference when tracing segments of an open curve of smooth and facetted objects. For a facetted object, the last segment of an open curve does not have a valid end point, only a start point. That is, there is an invisible "dummy" segment at the end of an open curve. For example, a simple vector line with three visible segments really has four segments. The last only serves the purpose to store the point index of its start point, since there are four points in a three segment open wire and segments of facetted objects only store the point index of the segment start, not the segment end. In a smooth object however, there is no dummy segment at the end of open curves. A smooth segment stores both, start and end point indices.

Tracing points

To get to the points of an object, the nested outline and segment loops can be used or the points can be accessed directly as before.

```
    fz_objt_get_pnt_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&ncord);

    for(i = 0; i < ncord; i++)
    {
        fz_objt_point_get_xyz(windex,obj,i,FZ_OBJT_MODEL_TYPE_UNSPEC,&pnt);
    }
```

The nested loops are as follows :

```
fz_objt_get_face_count(windex,obj,FZ_OBJT_MODEL_TYPE_UNSPEC,&nface);

for(i = 0; i < nface; i++)
{
    /* GET THE FIRST CURVE OF THE OBJECT */
    fz_objt_face_get_cindx(windex,obj,i,
        FZ_OBJT_MODEL_TYPE_UNSPEC,&chead);
    cindx = chead;

    do
    {
        /* GET THE FIRST SEGMENT OF THE CURVE */
        fz_objt_curv_get_sindx(windex,obj,cindx,
            FZ_OBJT_MODEL_TYPE_UNSPEC,shead);

        sindx = shead;

        do
        {
            /* GET THE SEGMENT'S START POINT INDEX */
            fz_objt_segt_get_start_pindx(windex,obj,sindx,
                FZ_OBJT_MODEL_TYPE_UNSPEC,pindx);

            /* GET THE POINT'S COORDINATE VALUE */
            fz_objt_point_get_xyz(windex,obj,pindx,
                FZ_OBJT_MODEL_TYPE_UNSPEC,&pnt);

            /* GET THE NEXT SEGMENT */
            fz_objt_segt_get_next(windex,obj,sindx,
                FZ_OBJT_MODEL_TYPE_UNSPEC,sindx);

        } while ( sindx != shead && sindx != -1 );

        /* GET THE NEXT CURVE */
        fz_objt_curv_get_next(windex,obj,cindx,
            FZ_OBJT_MODEL_TYPE_UNSPEC,&cindx);

    } while ( cindx != chead );
}
```

Note that the latter example traces each point of an object more than once, when the same point is an initial point of more than one segment, which is the norm. To trace the points only once, the code needs to be extended to include some marking method that will allow passing a point if it has already been traced.

Tracing through all faces of a smooth face

A developer may want to access all the facetted faces, which represent a face of a smooth object. This can be done with the following code :

```
    fz_objt_face_smod_get_fact_faces(windex,obj,findx,&fstart,&nface);

    for(j = fstart; j <= fstart + nface; j++ )
    {
        fz_objt_face_get_cindx(windex,obj,j,FZ_OBJT_MODEL_TYPE_FACT,&cindx);

        /* ETC */
        ...
    }
}
```

Note, that the call to `fz_objt_face_get_cindx` now uses the `FZ_OBJT_MODEL_TYPE_FACT` identifier, as the face index is guaranteed to be that of a facetted face.

Getting the sample points of smooth segments

The segments of smooth objects may be curved. For example a segment may have an arc or a spline curve as the underlying geometry. To display such a segment for example, it is necessary to generate sample points that represent the shape of the segment. This is not unlike the facetted faces of a smooth object, which represent the shape of the smooth geometry. **form•Z** stores the sample points of smooth segments and they can be accessed with a **form•Z** API call. This is shown below:

```
    fz_objt_segt_get_num_wire_pnts(windex,obj,sindx,&npnts);
    for(i = 0; i < npnts; i++)
    {
        fz_objt_segt_get_wire_pnt(windex,obj,sindx,i,&pt_xyz);
        ...
    }
```

Note, that the `fz_objt_segt_get_num_wire_pnts` function only works on smooth objects and the `sindx` parameter (which is the index of the segment) passed in must be that of a smooth segment. The coordinate value of a sample point on a smooth segment is retrieved with the function call `fz_objt_segt_get_wire_pnt`. Again, the object must be a smooth object and the segment index passed in must be that of a smooth segment.

1.6 Methods for constructing objects

Objects can be constructed in three distinctly different ways. . The first method constructs an object one face at a time. That is, a set of coordinate points in 3D space are defined and are then conncted to form segments, outlines, and faces. At the end the faces are linked to form an object, which is also made a member of a project. .That is, the new object is initially tagged as a temporary object and it becomes permanent when it is made part of a project by calling the `fz_objt_add_to_project` function. This is the most general method for generating objects as it can produce objects of any shape. However, it is restricted to facetted objects only and is typically tedious to execute. This method is discussed in more detail in section 1.6.1.

The second method derives specific shapes of objects directly, based on sets of parameters provided. It is based on the many API functions that **form•Z** offers, which more or less correspond to the construction tools available through the graphic interface of the program. Examples would be all the primitives, an object of revolution from a given profile shape and an axis, a swept object from a source and a path shape, etc. Here again, the new object is always initially tagged as a temporary object and it becomes permanent after it is made a member of a project.

The third method is known as **constructive solid geometry** (CSG). It consists of first generating a number of basic shapes and then using Boolean or other sculpting operations to construct a new object. This process takes advantage of the fact that form•Z can generate both temporary and permanent objects. The original basic objects that are used as operands are temporary objects and are never given the status of a permanent object. They can thus be easily deleted after they have performed their job, thus avoiding overloading the program memory. Only the final object is kept by being elevated from temprorary to permanent status, by calling the `fz_objt_add_to_project` function. After this call, the object shows up in the Objects palette, is drawn on the screen, and an undo record is generated for it, which allows a user to reverse the creation of the object.

1.6.1 Point-by-point object construction

Just as it is possible to trace the topology of an object directly, it is also possible to construct the topology and geometry of an object one ntity at a time. However, as it has already been mentioned, this is only possible for facetted objects, not for smooth objects. The latter have to be generated using one of the other two methods.

The low level construction of an object involves three major steps :

1. A new, empty object is created.
2. The points of the object are generated and loaded.
3. The segments, outlines, and faces of the object are constructed.

Creating a new object is done with function:

```
fz_objt_cnstr_objt_new(windex,&obj);
```

This new object is empty. It does not contain any faces, outlines, segments, or points.

The points for the object are loaded with function:

```
fz_objt_fact_add_pnts(windex,obj,pnts,npts);
```

The coordinate values of the points are generated and stored in an array of type `fz_xyz_td`. The size of the array is passed in via the last function argument. Once the object contains points, they must be connected to form segments, outlines, and faces. This is done with function:

`fz_objt_fact_create_face(windex,obj,pindx,npts,&findx);` for closed faces
or
`fz_objt_fact_create_wire_face(windex,obj,pindx,npts,&findx);` for open faces.

These function calls create one face at a time. For multiple faces, they must be called repeatedly, until all points are connected properly. The arguments to these functions are an array of point indices and a counter that indicates how many point indices are used. The function then constructs one face with one outline, connecting the points, identified in the point index array in the order they appear in the array. After the last face is created, the segments of the object are linked together with function:

```
fz_objt_fact_link_faces(windex,obj);
```

This sets the reversely coincident links of each segment. A complete construction of a cube using low level API functions is shown below. A more elaborate example can be found in the star plugin, which creates the faces of a star shaped objects using this method.

```
long      pindx[4];
fz_xyz_td pts[8];
fz_objt_ptr   obj;

/* MAKE A NEW EMPY OBJECT */
    fz_objt_cnstr_objt_new(windex,&obj);

/* ADD 8 POINTS */
pts[0] = {10,10, 0};
pts[1] = {60,10, 0};
pts[2] = {60,60, 0};
pts[3] = {10,60, 0};
pts[4] = {10,10,50};
pts[5] = {60,10,50};
pts[6] = {60,60,50};
pts[7] = {10,60,50};
    fz_objt_fact_add_pnts(windex,obj,pts,8);

/* CREATE 6 FACES TO MAKE A CUBE */
pindx[0] = 0; pindx[1] = 1; pindx[2] = 2; pindx[3] = 3;
    fz_objt_fact_create_face(windex,obj,pindx,4,NULL);
    pindx[0] = 0; pindx[1] = 4; pindx[2] = 5; pindx[3] = 1;
    fz_objt_fact_create_face(windex,obj,pindx,4,NULL);
pindx[0] = 1; pindx[1] = 5; pindx[2] = 6; pindx[3] = 2;
fz_objt_fact_create_face(windex,obj,pindx,4,NULL);
pindx[0] = 2; pindx[1] = 6; pindx[2] = 7; pindx[3] = 3;
fz_objt_fact_create_face(windex,obj,pindx,4,NULL);
pindx[0] = 3; pindx[1] = 7; pindx[2] = 4; pindx[3] = 0;
fz_objt_fact_create_face(windex,obj,pindx,4,NULL);
pindx[0] = 4; pindx[1] = 7; pindx[2] = 6; pindx[3] = 5;
fz_objt_fact_create_face(windex,obj,pindx,4,NULL);

/* LINK FACES */
fz_objt_fact_link_faces(windex,obj);

/* ADD OBJECT PERMANENTLY TO THE PROJECT */
fz_objt_add_objt_to_project(windex,obj);
```

1.6.2 Generating objects directly

An object can be generated directly by calling one of the many API functions that are made available to both plugin and script developers.   These functions correspond to the object generation operations that are available in the regular **form•Z** code. New such functions can also be written as plugins or scripts and made available to other plugins and scripts .

One of the simplest forms is, of course, the cube. An example of how to generate a cube directly follows. Recall that the example in the previous section, which discusses the point-by-point construction method, also generated a cube and you can now compare the two methods.

```
/* CREATE A CUBE */

wdh = {50,50,50};

fz_objt_cnstr_cube(windex,wdh,NULL,NULL,obj1);

fz_objt_add_objt_to_project(windex, obj1);
```

The cube is constructed by calling function `fz_objt_cnstr_cube`, after `x, y,` and `z` values are assigned to variable `wdt`, which is of type `fz_xyz_td`. The latter variable is included in the argument list of the call of the function, which also includes windex and obj1.

`windex` is the index of a window and determines the window (and project) on which the new object will be displayed, once it becomes a permanent object.

`obj1` is a pointer to an object structure, where the new object will be stored after its generation.

The two `NULL` values included in the function call correspond to variables that would normally carry translation (motion) and rotation parameters. Since no values are provided by the call these default to 0 values, which have no effect on the new object.

The object is first generated as a temporary object. The call to the `fz_objt_add_objt_to_project` function makes it permanent, which causes it to be displayed on the screen, listed in the Objects palette, and an undo record is created for it.

1.6.3 Constructive solid geometry

The method of constructive solid geometry comprises the generation of objects from other objects by applying different operations to them, most typically Boolean operations. While the method initially refers to solids only and operations that apply to solids, it can be extended to a broader range of object types.

An example of generating a simple object using the (CSG) method is shown below. The sample code creates two cuboids, which overlap to form a cross. They are unioned together to create the final shape. At the end, the cross is added to the project while the two temporary objects are deleted.

```
/* CREATE TWO CUBES WHICH OVERLAP */

wdh = {100,50,50};
fz_objt_cnstr_cube(windex,wdh,NULL,NULL,obj1);
wdh = {50,100,50};
fz_objt_cnstr_cube(windex,wdh,NULL,NULL,obj2);

/* UNION THE TWO CUBES TOGETHER */

fz_objt_list_create(objt_list);
fz_objt_edit_bool_union(windex,obj1,obj2,FALSE,objt_list);
fz_objt_list_get_objt(objt_list,0,cross_obj);
fz_objt_list_delete(objt_list);

/* DELETE THE CUBES AND ADD THE CROSS TO THE PROJECT */

fz_objt_edit_delete_objt(windex,obj1);
fz_objt_edit_delete_objt(windex,obj2);
fz_objt_add_objt_to_project(windex,cross_obj);
```

The calls in the first part are similar to the ones in the example of the previous section. In the second part, `objt_list`, which is a list of objects, is created first and is used by the `fz_objt_edit_bool_union` function (which executes the union of the two cubes) to store the resulting object. Even though in this case only one object is returned as the result of the union operation, the Boolean operations may return more than one object as their result. Because of this a list is used in order to be able to store all the objects. The object is read out of the list using the `fz_objt_list_get_`objt function. It is now called `cross_obj.` After it is read and the list is not needed anymore it is deleted to save memory.

In the third part of the example, the two cubes are deleted, again to save memory, since they are not needed anymore. The new object, `cross_obj`, is made part of the project, is displayed on the screen and in the Objects palette, and becomes undoable.

1.6.4 Editing objects

As with constructing objects, **form•Z** offers a large number of API functions which can change the shape of existing objects. These functions generally fall into two categories. Simple editing operations work on the object directly. They can be performed on any object. To move an object would be an example of a simple editing operation. The second type of editing operations are API functions, which retrieve and set the parameters of controlled objects. These API functions all follow a similar pattern. The API function to change the radius of an existing sphere object falls into this category.

The constructive solid geometry method that involves Boolean operations executed over solid objects may be considered as a third category of special complex editing operations. However, we have preferred to view it as a special construction method, as was discussed in the previous section.

Simple editing operations

Simple editing operations perform basic changes on an object. In general, any kind of object is allowed to be passed in as an argument to the respective **form•Z** API. The sample code below shows how to create a cube, copy it, move it and then delete it afterwards. The move, copy and delete API functions are simple editing operations.

```
/* CREATE A SIMPLE CUBE */
wdh = {10.0, 10.0, 10.0};
fz_objt_cnstr_cube(windex,wdh,NULL,NULL,obj);

/* MAKE A COPY */
fz_objt_edit_copy_objt(windex,obj,TRUE,copy_obj);

    /* MOVE THE COPY */
trl = {100.0, 0.0, 0.0};
fz_objt_edit_move_objt(windex,copy_obj,trl);

/* DELETE THE ORIGINAL CUBE */
fz_objt_edit_delete_objt(windex,obj);
```

Changing object parameters

Controlled objects in **form•Z** maintain the parameters with which they were initially created. These parameters can later be edited to modify the shape of the object. This can be achieved, for example, through the respective **Edit** dialog , which is invoked from the **Query** dialog by pressing the **Edit** button. **form•Z** offers two API functions for each object type: one to get and one to set a parameter. The functions all follow the same pattern. The only difference is that the parameter identifiers passed to the functions are unique for each object type. For example, to get the radii and partial on/off parameters of a sphere object the following call is made in a plugin.

```
fz_type_td   data;
fz_xyz_td radii;
fzrt_boolean  partial;

fz_objt_edit_sphr_parm_get(windex,obj,FZ_OBJT_SPHR_PARM_RADII,&data);
```

```
fz_type_get_xyz(&data,&radii);
fz_objt_edit_sphr_parm_get(windex,obj,FZ_OBJT_SPHR_PARM_PARTIAL,&data);
fz_type_get_boolean(&data,&partial);
```

The object passed to `fz_objt_edit_sphr_parm_get` must be a sphere object, otherwise an error will be generated. The x, y, and z radii of the sphere are initially stored in the data parameter and are retrieved with the API call `fz_type_get_xyz`. Depending on which parameter of a controlled object is retrieved, the data argument will contain values of a different type. For the radii parameter, it is an `fz_xyz_td`. For the partial on/off parameter, it is a Boolean value, etc. Which value type is associated with which parameter can be found in the html documentation of the get/set function for each object type. When using the get / set function in a script, it is not necessary to use the data argument, but the variable in which the value will be stored is passed directly to the get/set function. The same code from above written in a script looks as follows :

```
fz_xyz_td radii;
fzrt_boolean  partial;

fz_objt_edit_sphr_parm_get(windex,obj,FZ_OBJT_SPHR_PARM_RADII,radii);
fz_objt_edit_sphr_parm_get(windex,obj,FZ_OBJT_SPHR_PARM_PARTIAL,partial);
```

Setting an object parameter is very similar to getting it. For a plugin, the data argument is first filled with a value and then passed to the set function.

```
fz_type_td    data;
fz_xyz_td radii;
fzrt_boolean  partial;

radii.x = 10.0;
radii.y = 10.0;
radii.z = 10.0;
fz_type_set_xyz(&radii,&data);
fz_objt_edit_sphr_parm_set(windex,obj,FZ_OBJT_SPHR_PARM_RADII,&data);

    partial = TRUE;
    fz_type_set_boolean(&partial,&data);
fz_objt_edit_sphr_parm_set(windex,obj,FZ_OBJT_SPHR_PARM_PARTIAL,&data);

fz_objt_edit_parm_regen(windex,obj);
```

For a script the same code would look as follows :

```
fz_xyz_td radii;
fzrt_boolean  partial;

radii.x = 10.0;
radii.y = 10.0;
radii.z = 10.0;
fz_objt_edit_sphr_parm_set(windex,obj,FZ_OBJT_SPHR_PARM_RADII,radii);

partial = TRUE;
fz_objt_edit_sphr_parm_set(windex,obj,FZ_OBJT_SPHR_PARM_PARTIAL,partial);

fz_objt_edit_parm_regen(windex,obj);
```

Note, that when setting an object parameter, it is necessary to call the api function `fz_objt_edit_parm_regen`, to regenerate the shape of the object. This allows a plugin or script to change several object parameters at the same time and only regenerating the shape one time.

1.6.5 Working with object lists

Some editing operations create new objects. For example, the Boolean difference (fz_objt_edit_bool_difference), takes two operands, and may yield zero or more new objects. Since it is usually not known before the operation is executed how many new objects are created, the resulting objects are stored in a list. The calling code must first create an empty object list. It is then passed as an argument to the edit function. The number of objects and object pointers can be extracted from the list. Finally, the calling code must delete the object list. An example for the use of an object list is shown below :

```
fz_enty_list_ptr obj_list;
fz_objt_ptr      new_obj;
long          i,num_new_objs;

fz_objt_list_create(&obj_list);

fz_objt_edit_bool_difference(windex,obj1,obj2,obj_list);

num_new_objs = fz_objt_list_count(obj_list);
for(i = 0; i < num_new_objs; i++)
{
    fz_objt_list_get_objt(obj_list,i,&new_obj);

    /* DO SOMETHING WITH THE NEW OBJECT */

}

fz_objt_list_delete(&obj_list);
```

An object list may be reset with the API call `fz_objt_list_reset`. Resetting the list means that it is emptied and made ready to be used with another operation. That is, the list is not deleted, but all the objects it contains are removed from it. After resetting it, the list is at the same state it was when it was freshly created. This allows a reuse of the same list for a number of different editing operations in a row. Note that resetting the list does not delete the objects in the list themselves. This can be done using the function `fz_objt_edit_delete_objt` for each object in the list prior to resetting the list.

1.6.6 Working with groups

A group table is a generic mechanism that organizes entities in a hierarchical fashion. Grouping is currently supported for three sets of entities : objects, lights and layers. The grouping hierarchy is visualized in the respective palette, for example the Objects palette. A few naming conventions are outlined below which help in understanding the api functions that deal with groups :

Root : A group which is at the top of the group hierarchy. From the root all other groups can be accessed.
Child : A given group that is contained in another group.
Parent : A group that contains a given group. The given group is a child of the parent.
Sibling : A group that has the same parent as another group.
Node : A group that contains at least one other group
Leaf : A group that does not contain any other groups. However, it references exactly one entity, such as an object, layer or light.

A set of api functions are provided by form•Z, that allow the developer to traverse the group hierarchy and manipulate groups. In the naming and documentation of the api function the above outlined terms are used extensively to categorize the different kinds of groups. The data struture of a group table is called :

`fz_grup_table_ptr`

To get the group table for objects, the api function `fz_objt_grup_get_table` is used. `fz_layr_get_grup_table` gets the layer group table of a project and `fz_lite_get_grup_table` gets the light group table. Once a group table is retrieved with any of these three api functions, a set of group table api functions work on that table, regardless of which set of entities it organizes (objects, lights or layers).

An individual group is represented by the data type :

`fz_grup_ptr`

The internal structure of a group table and the hierarchy shown to the user in the palette differ slightly from each other and it is important to understand the differences to use the group table api functions properly. At the top of each group table is the root group. It contains everything shown in a particular palette, but the root itself is not displayed in the palette. For a given group table, the root is retrieved with the api call `fz_grup_get_root`. To get to all the groups contained inside another group, a simple loop can be written :

```
fz_grup_table_ptr   gtable,
fz_grup_ptr         root,grup;

fz_objt_grup_get_table(windex,&gtable);
fz_grup_get_root(windex,gtable,&root);

fz_grup_get_child(windex,gtable,root,&grup);
while ( grup != NULL )
{
        …
        fz_grup_get_next(windex,gtable,grup,&grup);
}
```

In the example above, first the group table for objects is retrieved with `fz_objt_grup_get_table`. Next, the root group for that group table is acquired with `fz_grup_get_root`. The first group inside the root group is returned by `fz_grup_get_child`. The while loop iterates through all sibling groups of the first child of the root. The example above would access all groups that are shown on the first "level" in the

Objects palette. Usually, groups are nested inside other groups. In order to traverse all groups in a nested structure. the above sample code needs to be modified as a recursive function :

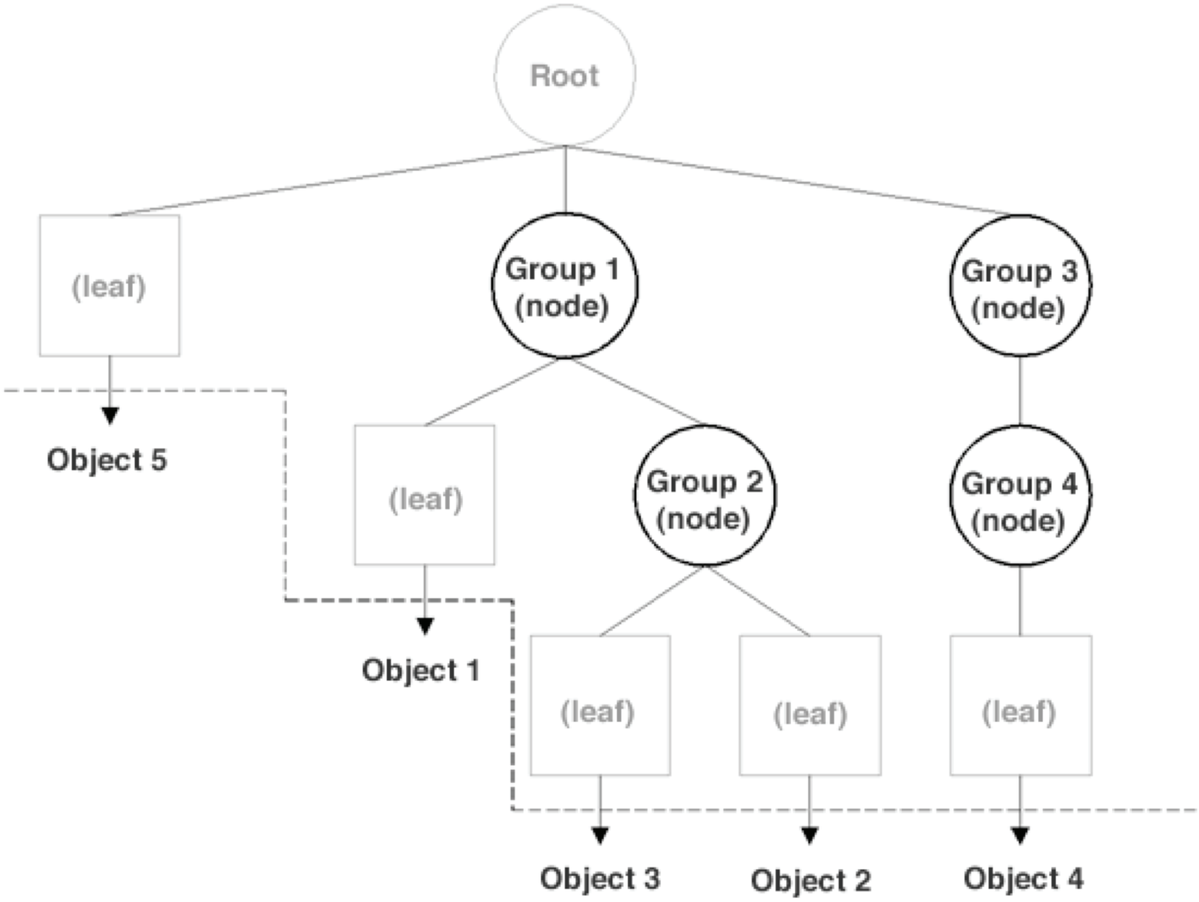
```
void    my_group_traverse(
        long                windex,
        fz_grup_table_ptr   gtable,
        fz_grup_ptr         parent
        )
{       fz_grup_ptr         grup;
        fz_grup_type_enum   grup_type;
        fz_objt_ptr         obj;
        fz_tag_td           entity_tag;

        fz_grup_get_child(windex,parent,&grup);
        while ( grup != NULL )
        {
                fz_grup_get_type(windex,gtable,grup,&grup_type);

                /* IT IS A NODE GROUP,  */
                if ( grup_type == FZ_GRUP_TYPE_NODE )
                {
                        /* CALL my_group_traverse RECURSIVELY */
                        my_group_traverse(windex, gtable, grup);
                }
                /* IT IS A LEAF NODE */
                else
                {
                        /* GET THE TAG OF THE REFERENCED OBJECT */
                        fz_grup_get_leaf_tag(windex,gtable,grup,&entity_tag);

                        /* GET THE OBJECT FROM THE TAG */
                        fz_objt_tag_to_ptr(windex,&entity_tag,&obj);

                        /* DO SOMETHING WITH THE OBJECT */
                        …
                }

                /* GET THE NEXT SIBLING OF THE CURRENT GROUP */
                /* IT WILL BE NULL FOR THE LAST GROUP, WHICH WILL */
                /* STOP THE LOOP */
                fz_grup_get_next(windex,gtable,grup,&grup);
        }

}
```

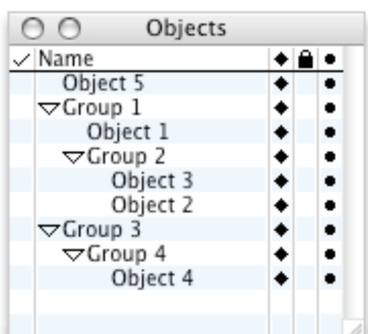
To initiate the traversal of all groups, the recursive function shown above can be called with the root group as input :

```
fz_objt_grup_get_table(windex,&gtable);
fz_grup_get_root(windex,gtable,&root);
my_group_traverse(lwindex, gtable, root);
```

In addition to the root not being shown in the palette, there is a second important difference between the palette and the group table. A leaf is a group, which does not contain any other groups. Its sole purpose is to establish the link to the entity that is referenced (i.e. the object, layer or light). A leaf group is also not shown in the palette. This may be confusing at first, because in the palette it appears that an object that is inside a group, should be referenced by that group. THIS IS NOT THE CASE ! The group visible in the palette contains an invisible leaf group, which in return references the object. A diagram of a group table and the corresponding palette is shown below. The diagram uses circles for nodes and squares for leafs.

Note, that only the groups in circles appear in the palette, except, of course, for the root, which is a special node.





The dashed line in the diagram represents the border between the group world and the world of the referenced entities, in this case the objects. The generic group api functions operate entirely inside the group world, and therefore can be used with any of the three grouping structures (objects, lights and layers). Sometimes it is necessary to cross the border from objects into the group world. That is, the developer wants to know, which group contains a given object. For this purpose, the object api function `fz_objt_get_grup_tag` needs to be used (similar functions exist for layers and lights). The tag returned by the function belongs to a leaf group, which can easily be seen in the diagram, but not in the palette. In order to get to the group in the palette, that is shown as containing that object, the parent of the leaf group needs to be retrieved. This is shown in the sample code below :

```
fz_tag_td     grup_tag;
fz_grup_ptr   leaf_grup,grup;

fz_objt_get_grup_tag(windex,obj,&grup_tag);
fz_grup_tag_to_ptr(windex,gtable,&grup_tag,&leaf_grup);
fz_grup_get_parent(windex,gtable,leaf_grup,&grup);
```

A common mistake would be to not get the parent of the leaf group and then to try to access information about the group. For example, a developer may want to get the name of the group which contains a given object. If the api function `fz_objt_grup_get_parm_data` would be called with a leaf node as input to get the name, an empty string would result.

The existence of leaf groups also allows the developer to move a referenced entity from one node group to another. Instead of offering separate api functions that would, for example, move an object (or layer or light) to a new node, one generic group api function is sufficient. `fz_grup_move_by_parent` specifies a new parent group for a given group. If the given group is a leaf node, the action performed is equivalent to the user moving an object inside another node group. One can also move an object outside of any node group by specifying the root group as the new parent of the leaf group. If the given group is a node group, it is equivalent to the user moving a group in the palette inside another group.

While `fz_objt_get_grup_tag` crosses from the object world into the group world, the group api function `fz_grup_get_leaf_tag` goes the opposite direction. Note, that it is a generic group function. It retrieves the tag of the references entity. To get the entity pointer one must call the appropriate conversion function, as shown in the sample code below :

```
/* GET THE GROUP TYPE */
fz_grup_get_type(windex,gtable,grup,&grup_type);

if ( grup_type == FZ_GRUP_TYPE_LEAF )
{
	/* GET THE TAG OF THE REFERENCED OBJECT */
	fz_grup_get_leaf_tag(windex,gtable,grup,&entity_tag);

	/* GET THE OBJECT FROM THE TAG */
	fz_objt_tag_to_ptr(windex,&entity_tag,&obj);

	/* DO SOMETHING WITH THE OBJECT */
	…
}
```

If the sample code were to work with lights, it would be the same, except, that `fz_lite_tag_to_ptr` would be used.

2.0 Writing form•Z Plugins

2.1 Introduction

A **form•Z** plugin is an extension to **form•Z** in the form of a compiled machine code. A single plugin represents a group of functionality that appears to the user as a single package. This allows the user to enable or disable the plugin in the **Extensions** dialog. The plugin is written in the C or C++ computer language and compiled into a shared library (Macintosh) or a dynamic link library (Windows). These libraries are referred to as the plugin file and they must have a .fzp extension (and 'fzpl' signature on Macintosh) to identify them as a **form•Z** plugin. A single plugin file may contain multiple plugins. This allows for multiple plugins to be delivered in a single .fzp file.

form•Z automatically recognizes plugins by finding them in designated directories at startup. The default directory is the "Plugins" folder the **form•Z** application folder. The plugin directories and associated options are controlled in the **Extensions** dialog. When **form•Z** finds a file with the .fzp extension (or 'fzpl' signature) it validates the plugin. The validation process prevents a non-plugin file with a .fzp extension from producing undesirable results. Once the plugin is validated, **form•Z** communicates with the plugin file through the plugin file entry function. The entry point is a function in the plugin file which receives and handles a number of messages from **form•Z**. The plugin file entry function is called to register the plugins and their functionality. The plugin file entry function is also called when **form•Z** quits to unregister the plugins in the file.

The remainder of the communication between **form•Z** and a plugin is done through pointers to functions. The function pointers are grouped in C language structures called function sets. Each function set contains functions of related subjects. Function sets are divided into two types: API and call back. API function sets contain functions that **form•Z** provides for the plugin to use. Call back function sets contain functions that are implemented by the plugin. These functions are called by **form•Z** as needed to perform the plugins tasks. Call back functions are registered with **form•Z** during plugin registration through the plugin file entry function. The **form•Z** plugin manager (FZPL) is used to manage the plugin file and the access and definition of function sets.

form•Z uses UUID's (Universal Unique Identifier) throughout for uniquely identifying items and avoiding naming collisions. A UUID is a 16-byte string that is generated using an algorithm that guarantees a unique sequence of bytes (string). Plugins must use UUID's in various places to guarantee that they do not collide with other plugins or **form•Z**. For example, when a plugin is registered it must provide A UUID. This distinguishes it from other plugins and also allows **form•Z** to retain information about the plugin (for example, its user controlled enable state in the **Extensions** dialog). **form•Z** comes with a utility plugin to automatically generate UUIDs which is of particular use for extension developers. It is not recommended to create a UUID by "making one up" without a computer.

2.2 Plugin File Validation

form•Z validates each plugin file to be sure that it is in fact a **form•Z** plugin and not another file that has been given the .fzp extension. This prevents **form•Z** from crashing when attempting to load an invalid file. **form•Z** looks for three exported symbols in the plugin file shared library (Macintosh) or a dynamic link library (Windows). The first is a global string variable called fz_descriptor with a value of "formZ_Plugin". The second is a global plugin manager version variable named fz_API_version. The value of this variable is the version of the **form•Z** plugin API that the plugin was built with. The value of this symbol is used by **form•Z** to properly handle API version differences between a plugin and **form•Z**. If the version is greater than the version supported by **form•Z**, then the plugin file will not be loaded. The third validation is the presence of the plugin entry function that must be named fz_plugin_entry. All three of these symbols must be present in the plugin file and exported.

The definition of these exported symbols should look like the following:

```
FZPL_PLUGIN_DATA(char)           fz_descriptor[] = "formZ_Plugin";

FZPL_PLUGIN_DATA(fzpl_vers_td)   fz_API_version = FZPL_VERS_MAKE(5,0,0,0);

FZPL_PLUGIN_FUNC(fzrt_error_td)  fz_plugin_entry(
      const fzpl_fset_glue_fset * const            fzpl_glue,
      fzpl_command_td                              message,
      const fzpl_host_config_td * const            hostConfig );
```

These are defined for the plugin developer in the provided C header file “fz_plugin_glue_api.h”. This file should be included in one (and only one) of the plugin source C files.

2.3 Plugin File Entry Function

The plugin file entry function is used by the plugin manager (FZPL) to establish the communication between **form•Z** and the plugin. This function receives and handles a number of messages from **form•Z**. A return value of `FZRT_NOERR` indicates that the message was handled properly. A return message of anything other than `FZRT_NOERR` indicates that the message was not handled properly.

```
FZPL_PLUGIN_FUNC(fzrt_error_td) fz_plugin_entry(
      const fzpl_fset_glue_fset * const      fzpl_glue,
      fzpl_command_td                        message,
      const fzpl_host_config_td * const      hostConfig )
{
      fzrt_error_td err = FZRT_NOERR;

      switch (message)
      {
            case FZPL_PLUGIN_CHECK:
                  ...
            break;

            case FZPL_PLUGIN_INITIALIZE:
                  ...
            break;

            case FZPL_PLUGIN_EXIT:
                  ...
            break;
```

```
    }
    return ( err );
}
```

fzpl_glue

This parameter is a function set (structure of function pointers) that connects the plugin and **form•Z** together. The functions in this function set include functions for accessing other function sets, defining call back function sets for **form•Z** and for registering an unregistered plugin with **form•Z**. This function set is fully documented in the API Reference.

message

This parameter is the message or command that is sent from **form•Z**. There are currently three messages that are sent to the entry function:

`FZPL_PLUGIN_CHECK`

This message is sent at startup of **form•Z** and indicates that a plugin file should check to see if there is any condition that would prevent the plugin from functioning properly. This includes checking things like the version of **form•Z** that's loading it, validating any licensing being used, allocating any needed global memory, loading resources, and determining if all the required **form•Z** API function sets are available. If there is anything that could prevent the plugin(s) from operating correctly, an error should be returned so **form•Z** can unload the plugin file. Otherwise `FZRT_NOERR` should be returned to indicate that the plugin(s) should be loaded.

`FZPL_PLUGIN_INITIALIZE`

This message is sent at startup of **form•Z**, after the `FZPL_PLUGIN_CHECK` message. This message indicates that the plugins in the file should be registered and any needed call back function sets should be defined. If an error occurs it should be returned from the entry function and the plugin will be unloaded. However, it is preferable that potential errors and dependencies are checked in the `FZPL_PLUGIN_CHECK` message as this is more efficient.

The function `fzpl_plugin_register` must be called for each plugin in the plugin file to register the plugin with **form•Z**. The registration process installs the plugin into the **form•Z** Extensions Manager and facilitates the binding of the plugin implemented callback function sets (see section 2.4). Each call to `fzpl_plugin_register` creates a plugin entry in the Extensions manager. All of the functionality that is associated with a plugin can be enabled and disabled by the user in the Extensions Manager dialog.

The following code snippet shows a call to `fzpl_plugin_register` for a tool plugin.

```
    err = fzpl_glue->fzpl_plugin_register(
        MY_PLUGIN_UUID,                 /* UUID for my plugin */
        my_name,                        /* name string for my plugin */
        MY_PLUGIN_VERSION,              /* version of my plugin */
        MY_PLUGIN_VENDOR,               /* name string for my company */
        MY_PLUGIN_URL,                  /* url string for my company */
        FZ_TOOL_EXTS_TYPE,              /* UUID of formZ tool plugin */
        FZ_TOOL_EXTS_VERSION,           /* version of formZ tool plugin*/
        my_plugin_error_string_func,    /* error string function */
        0,                              /* number of dependencies */
        NULL,                           /* pointer to dependency list */
        &my_plugin_runtime_id);         /* runtime id for my plugin */
```

The first parameter is a UUID for the plugin. This distinguishes the plugin form any other extension. The second parameter is the name of the plugin. The plugin name should be loaded from a **form•Z** resource file (.fzr) through the `fzrt_fzr_get_string` function. This allows the name of the plugin to be localized. The name of the plugin is shown in the **Extensions Manager** dialog. The **form•Z** resource file format and functions are fully documented in section 1.4.2.

```
err = fzrt_fzr_get_string(my_plugin_rsrc_ref,
            MY_STRINGS,
            MY_NAME_STR,
            my_name);
```

The third parameter is the version of the plugin and shown in the **Extensions Manager** dialog. The fourth parameter is the name of the vendor (author) of the plugin. The vendor name is shown in the **Extensions Manager** dialog. The fifth parameter is the URL for the vendor (e.g. www.mygreatplugin.com). The URL is displayed in the **Plugin Information** dialog accessed from the Extensions Manager.

The sixth parameter is the UUID of the type of the plugin. **form•Z** supports a variety of plugin types as described in section 2.6. The seventh parameter is the version for the implementation of the type of the plugin. This informs form•Z what version of the SDK the plugin was built with. The UUID and version definitions can be found in the **form•Z** API header files.

The eighth function is a function name (pointer) for a function that accesses error messages for the plugin. The function is registered with the **form•Z** runtime error manager by the plugin manager.

```
fzrt_boolean my_plugin_error_string_func(long err, char *str, short str_len)
{
	char msg[STRING_SIZE];

	fzrt_fzr_get_string(my_plugin_rsrc_ref, MY_ERROR_STRINGS, err, msg);
	strncpy(str, msg, str_len);

	return(TRUE);
}
```

The ninth and tenth parameters are used when plugins depend on each other. Since the loading order of plugins is not guaranteed to be consistent, plugin to plugin dependencies can not be checked until all plugins are registered. The eighth parameter is the number of dependent plugins and the ninth is a list of information about dependent plugins (one record per dependent plugin). Note that dependent plugins need access to some common information (usually in a C header file) so that they can have basic information about the dependent plugin. At a minimum, this information needs to include the UUID of the plugin, the name of the plugin, the version of the plugin, and the plugin's vendor name and URL. If a dependent plugin is missing, form•Z will issue an error alerting the user of the problem and the depending plugin will not be loaded.

The following code snippet shows a call to `fzpl_plugin_register` for a tool plugin dependent on another plugin identified by `DEPEND_PLUGIN_ID`, `DEPEND_PLUGIN_VERSION`, `DEPEND_PLUGIN_NAME`, and `DEPEND_VENDOR_URL`.

```
	Long                        num_depends = 1;
	fzpl_plugin_dependency_td  depends[1];

	fzrt_UUID_copy(DEPEND_PLUGIN_ID, depends[0].plugin_id);
```

```
strncpy(depends[0].plugin_name,DEPEND_PLUGIN_NAME,FZPL_NAME_SIZE);
strncpy(depends[0].plugin_vendor_name,DEPEND_VENDOR_NAME,FZPL_NAME_SIZE);
strncpy(depends[0].plugin_URL,DEPEND_VENDOR_URL,FZPL_NAME_SIZE);
depends[0].plugin_version  = DEPEND_PLUGIN_VERSION;

err = fzpl_glue->fzpl_plugin_register(
      MY_PLUGIN_ID,                     /* UUID for my plugin */
      my_name,                          /* name string for my plugin */
      MY_PLUGIN_VERSION,                /* version of my plugin */
      MY_PLUGIN_VENDOR,                 /* name string for my company */
      MY_PLUGIN_URL,                    /* url string for my company */
      FZ_TOOL_EXTS_TYPE,                /* UUID of formZ tool plugin */
      FZ_TOOL_EXTS_VERSION,             /* version of formZ tool plugin*/
      my_plugin_error_string_func,      /* error string function */
      num_depends,                      /* number of dependencies */
      depends,                          /* pointer to dependency list */
      &my_plugin_runtime_id);           /* runtime id for my plugin */
```

The final parameter is the runtime id of the plugin which is returned from the function. The runtime id is generated by the plugin manager and is used in subsequent plugin manager function calls.

`FZ_PLUGIN_EXIT`

This message is sent when **form•Z** is unloading all plugins (at quit or exit). This indicates that the plugins should release any memory, resources, or API function sets that were loaded during the `FZPL_PLUGIN_CHECK` or `FZPL_PLUGIN_INITIALIZE` messages or during the execution of the plugin.

`hostConfig`

This parameter is a structure that contains information about the **form•Z** application that is using the plugin file (name, version, etc). This structure is fully documented in the API Reference.

2.4 Working with function sets

A function set is a structure that contains function pointers of related functionality within **form•Z**. Function sets are divided into two types: API and call back. API function sets contain functions that **form•Z** provides for the plugin to use. Call back function sets contain functions that are implemented by the plugin. The **form•Z** plugin manager manages function sets.

Each function set's structure is defined in its respective C header file. **form•Z** function set structure names are of the form "`fz_..._fset`". Since function sets will change over time as **form•Z** functionality is added or changed, it is important that a plugin request the version of the function set that it is complied for. When new functions are added to a function set, they are always added to the end of the function set and the function set version is incremented. Existing functions will always be present and the names will never change. In the rare event that a function becomes deprecated (no longer supported), it will still exist in the function set, but it will always return an error. This system of function set management insures that plugins will not be broken by the evolution of **form•Z**. it also enables plugins to take advantage of new functionality as it becomes available.

For each function set, the C language header file that contains the structure definition, contains 3 constants that are used to identify the function set. The type of the function set is a UUID that uniquely identifies the function set. The name of the function set is the textual description of the function set. The version is the version for the function set definition. These constants should be used in plugin manager calls (fzpl) when referring to the corresponding function set. The constants for the math function set are shown below.

```
#define FZ_MATH_FSET_TYPE  \
       "\x67\x12\xef\xc6\xd4\x77\x46\xfc\xa6\xc6\x6d\xea\x20\xff\x81\x63"
#define FZ_MATH_FSET_NAME         "formZ math funcs"
#define FZ_MATH_FSET_VERSION      FZPL_VERS_MAKE(1,0,0,1)
```

Accessing an API function set and calling functions

API function sets contain functions that **form•Z** provides for the plugin to use. Calling the plugin glue function "fzpl_fset_acquire" acquires a function set for use within the plugin. This locates the function set and fills the plugin's copy of the function set structure. The plugin can declare the function structure as a global variable so that the functions can be accessed throughout the plugin. If a plugin requests a function set that is not available, or a version of a function set that is not available, an error is returned. The following shows a call to access the math function set.

```
fz_math_fset  math_funcs;   /* global */

...

       fzrt_error_td        err;


       err = fzpl_glue->fzpl_fset_acquire(
              FZ_MATH_FSET_TYPE,              /* UUID of math function set */
              FZ_MATH_FSET_VERSION,           /* version of math function set */
              FZRT_UUID_NULL,
              FZPL_TYPE_STRING(fz_math_fset), /* type string for function set */
              sizeof(math_funcs),                    /* size of function set
       structure*/
              (fzpl_fset_td *) &math_funcs ); /* address of function set */
```

The first parameter is the type constant (UUID) of the function set. The second parameter is the version constant for the function set. The third parameter is the UUID of the module that defines the function set. `FZRT_UUID_NULL` should be used for **form•Z** function sets. Function sets can be used to share functionality between plugins as described in a following section. In this case the parameter is used to identify the plugin that created the function set. The fourth parameter is a string that identifies the name of the function set structure. The macro `FZPL_TYPE_STRING` converts the type name into a string. This parameter makes sure that the structure type provided matches the expected type. The fifth parameter is the size of the structure. . The C macro `sizeof()` should always be used to get the proper size for the plugins instance of the structure. The final parameter is the address of the plugin's instance of the structure. This is the structure in the plugin that is filled with the contents of the function set. The `fzpl_fset_acquire` function is fully documented in the API Reference.

Once a function set is acquired, the function pointers can be used to call the desired function. The following is an example of call to the math function to set an identity matrix.

```
       fz_mat4x4_td  mat;

       math_funcs.fz_math_4x4_set_identity(&mat);
```

When a function set is no longer needed, it can be released by calling the plugin glue function `fzpl_fset_release`. This unloads the function set from the plugin and informs the plugin manager that you no longer need this functionality. If a function from a plugin set is only needed once, then it is recommended to acquire the function set, call the function, and then release the function set. As most plugins will call multiple functions from multiple function sets throughout the plugin, it is recommended to load all required function sets in the entry function while handling the `FZPL_PLUGIN_CHECK` message and to release all function sets in the entry function while handling the `FZ_PLUGIN_EXIT` message. The following is an example of call to `fzpl_fset_release` to release the math function set.

```
fzpl_glue->fzpl_fset_release( (fzpl_fset_td *)&math_funcs );
```

Defining a call back function set

Call back function sets contain functions that are defined by the plugin. These functions are called by **form•Z** as needed to perform the plugins tasks. Call back functions are registered with **form•Z** in the entry function while handling the `FZPL_PLUGIN_INITIALIZE` message. Calling the plugin glue function "`fzpl_plugin_add_fset`" defines a function set. The following is an example of the definition of a project function set for a tool called star.

```
        err = fzpl_glue->fzpl_plugin_add_fset(
                        star_tool_plugin_runtime_id,
                        FZ_TOOL_CBAK_FSET_TYPE,
                        FZ_TOOL_CBAK_FSET_VERSION,
                        FZ_TOOL_CBAK_FSET_NAME,
                        FZPL_TYPE_STRING(fz_tool_cbak_fset),
                        sizeof (fz_tool_cbak_fset),
                        star_tool_fill_fset,
                        FALSE);
```

The first parameter is the runtime id of the plugin which is returned from a previous call to `fzpl_plugin_register` which registered the plugin with the plugin manager. The second parameter is the type constant (UUID) of the function set. The third parameter is the version constant for the function set. The fourth parameter is a string that identifies the name of the function set structure. The macro `FZPL_TYPE_STRING` converts the type name into a string. This parameter makes sure that the structure type provided matches the expected type. The fifth parameter is the size of the structure. The C macro `sizeof()` should always be used to get the proper size for the plugins instance of the structure. The sixth parameter is the name of (pointer to) a plugin-defined function that fills the function set structure (see next section). The final parameter is reserved for future use and should always be `FALSE`. The `fzpl_fset_acquire` function is fully documented in the API Reference.

The plugin manager calls the fill function when **form•Z** requests the functions from the plugin. This function fills the function set structure with the names of (pointers to) the plugin defined functions. The following is how the star tool fill function would look. In this function the call to the glue function `fzpl_fset_def_check` is used to be sure that the proper data was requested by **form•Z**. The fill function (`fzpl_fset_def_get_fset_func`) is fully documented in the API Reference.

```
fzrt_error_td star_tool_fill_fset (
                const fzpl_fset_def_ptr fset_def,
                fzpl_fset_td * const fset )
{
        fzrt_error_td       err = FZRT_NOERR;
        fz_tool_cbak_fset   *tool_func;

        err = plugin_stuff.fzpl_glue->fzpl_fset_def_check (
```

```
                fset_def,
                FZ_TOOL_CBAK_FSET_VERSION,
                FZPL_TYPE_STRING(fz_tool_cbak_fset),
                sizeof ( tool_func ),
                FZPL_VERSION_OP_NEWER );

        if ( err == FZRT_NOERR )
        {
                tool_func = (fz_tool_cbak_fset *)fset;

                tool_func->fz_tool_cbak_info      = star_tool_info;
                tool_func->fz_tool_cbak_name      = star_tool_name;
                tool_func->fz_tool_cbak_uuid      = star_tool_uuid;
                tool_func->fz_tool_cbak_icon_file= star_tool_icon_file;

                ...
        }

        return err;
}
```

In this example, each of the functions `star_tool_info`, `star_tool_name`, etc., for which there is an assignment, would need to be implemented by the plugin developer. Note that, depending on the requirements of a function set, some functions in a call back function set may be optional. This means that the implementation of the function is at the discretion of the plugin developer. In the case of optional functions, **form•Z** detects the presence of (or lack thereof) the optional functionality and handles it accordingly. If a required function is not provided, then the plugin will not load. If a function is not provided, its value in the function set is defined to be `NULL`. All functions in a function set are initialized to `NULL` in `fzpl_fset_def_check()` so optional functions do not need to be explicitly set to `NULL` by the plugin if they are not provided. Please see the specific documentation for the each call back function set for details of what is required and what is optional.

Sharing function sets between plugins

Although the primary function of function sets is to provide a linkage between **form•Z** and a plugin, function sets can be used to shared functionality between plugins. In this case, the definition of the function set structure and required constants is done in a header file that is accessible to both plugins. The plugin which is to publish or define the function set calls the function `fzpl_plugin_add_fset` to define the function set using the function set information from the common header file. The plugin which wants to subscribe or use the function set calls `fzpl_fset_acquire` using the function set information from the common header file and the UUID of the plugin that created the function set.

2.5 Compilers

It is important that **form•Z** plugins are built with a compiler that is compatible with the **form•Z** header files. The sample code included with the **form•Z** SDK contains project files to build the sample plugin files. These project files are for the following recommended compilers. The following are the currently supported compilers:

Macintosh

form•Z is a Mach O application on the Macintosh platform. **form•Z** is built with CodeWarrior 9.2 and this is the recommended compiler for building **form•Z** plugins on the Macintosh. CodeWarrior

8.3 has also been tested for building **form•Z** plugins and was found to work properly. We did observe problems debugging plugins under CodeWarrior 8.3 on OS X. These problems do not occur with CodeWarrior 9.2.

Windows

form•Z is built with Microsoft Visual C 6.0 and this is the recommended compiler for building **form•Z** plugins on Windows. Microsoft Visual C 7.1has also been tested for building **form•Z** plugins and was found to work properly.

2.6 Interface

The **form•Z** API includes support for common interface features such as dialogs, alerts, palettes, wait cursor, key cancel detection and progress bars. The **form•Z** user interface manager (FUIM) manages these interfaces. The prefix `fz_fuim_` is used for all of the FUIM API entities (functions, types, constants etc.).

The layout of interface elements (buttons, menus, text, etc.) found in dialogs and palettes is called a **FUIM template**. The template contains the definition of the interface elements, the definition of dependencies between the elements, and the connection to data storage (variables) in the extension. The **form•Z** template manager handles the graphic layout of the template automatically and deals with all platform specific issues. The template definition is hierarchically organized in the form of a tree. That is, each element has a parent element and may have multiple sibling elements and child elements. The interface elements are implicitly dependent on their parent. That is, if the parent element is disabled, all of its descendents are also disabled.

Templates are defined through a **FUIM template function** that is provided to **form•Z** by the extension. The template function defines the template by calling **form•Z** API functions to create the interface elements, define relationships between items, and bind the data storage (variables) from the extension to the elements. The template function is provided to **form•Z** when a dialog is invoked through a dialog driver, or through specific call back functions provided by **form•Z**. These call back functions vary by the type of extension and are discussed in section 2.7.

Note that for clarity the strings in the example in this section are shown directly in the code rather than using the recommend method of retrieving the strings from .fzr files, as described in section 1.4.2.

2.6.1 Alerts

Alerts are simple dialogs that get the user's attention by beeping and presenting information or posing questions. They are frequently used for error notification or for asking the user to make decisions at critical times. Alerts usually consist of a simple message and one or more buttons for the user to select the desired response. An icon is shown in the alert to indicate that the alert represents an error, a question or just useful information. The alert is closed when the user selects one of its buttons. A set of standard alerts is provided and custom alerts can be created using a set of functions to build and display the alert as follows

Standard confirmation alert

```
long  fz_fuim_alrt_std_confirm(
       char                       *prmt_str,
       fz_fuim_std_conf_enum      confirm_flags
       );
```

This alert contains a single prompt text string and up to two buttons. This is useful for posting a simple notification or asking a simple OK/Cancel or Yes/No question. The `prmt_str` parameter is the prompt text for the alert. The `confirm_flags` parameter indicates which buttons the alert should have as follows:

`FZ_FUIM_ALRT_CONFIRM_OK`: The alert has a single button with a title of OK.
`FZ_FUIM_ALRT_CONFIRM_OK_CANCEL`: The alert has a button with a title of OK and a button with a title of Cancel.

FZ_FUIM_ALRT_CONFIRM_YES_NO: The alert has a button with a title of Yes and a button with a title of No.

The alert remains on the screen until the user selects one of the buttons in the alert. The function returns FZRT_STD_OK if an OK or Yes button is pressed or FZRT_STD_CANCEL if a Cancel or No button is pressed. The following as and example of a standard confirmation alert used to ask the user if they wish to proceed with an operation.

```
long          rv;

rv = fz_fuim_alrt_std_confirm(
              “Are you sure you want to proceed?”,
              FZ_FUIM_ALRT_CONFIRM_OK_CANCEL);

if(rv == FZRT_STD_OK)
{
       /* perform action here */
}
```

Standard name alert

```
long fz_fuim_alrt_std_name (
       char                     *prmt_str,
       char                     *name,
       long                     max_len
       );
```

This alert contains a single prompt text string, an editable name text field and the standard OK and Cancel buttons. This is useful for asking the user for simple text input. The prmt_str parameter is the prompt text for the alert. The name parameter is the string shown in the edit field. This parameter contains the desired default or current value for the name string. When the dialog is dismissed, this parameter contains the string that was entered in the text field. The max_len parameter is the length of the name string (in bytes). The alert remains on the screen until the user selects one of the buttons in the alert. The function returns FZRT_STD_OK if the OK button is pressed or FZRT_STD_CANCEL if the Cancel button is pressed. The following as an example of a standard name alert used to change an object name for a given object (obj) of a project window (windex);.

```
long          rv;
char          name[256];

if(fz_objt_attr_get_objt_name (windex, obj, name) == FZRT_NOERR)
{      rv = fz_fuim_alrt_std_name (
              “New object name:”,
              name,
              256);

       if(rv == FZRT_STD_OK)
       {
              fz_objt_attr_set_objt_name(windex, obj, name);
       }
}
```

Standard error alert

```
fzrt_boolean fz_fuim_alrt_std_error(
       fzrt_error_td       err_id,
       long                where_id,
```

```
	char			*where_str
);
```

This alert is used for displaying error messages. This is used for posting error messages returned from **form•Z** API functions or errors in an extension that registered the error with the `fzrt_error_set` function. **form•Z** will post error messages for extensions that return errors from their call back functions, however, there are times where it may be desirable for an error alert to be displayed from an extension directly.

The alert contains a single prompt text string and the standard OK button. The `err_id` parameter is the error value returned from a **form•Z** API function or `fzrt_error_set` function call in an extension. The `where_id` parameter is a numeric indicator of where in the extension the error occurred. Each call to the `fz_fuim_alrt_std_error` function should have a unique numeric value in this parameter so that the location in the extension code where the error occurred can be identified. The `where_str` is an optional parameter that complements `where_id`. This string can be used to give additional details of where in the extension the error occurred.). The alert remains on the screen until the user selects the OK button in the alert.

```
err = fz_objt_attr_set_objt_name(windex, obj, name);

if(err != FZRT_NOERR)
{	fz_fuim_alrt_std_error(err,	1, "Attempting to change name");
}
```

Custom alerts

Custom alerts are constructed by initializing an alert pointer, then adding prompt text item(s) and button item(s). The alert is then displayed to the user and disposed when it is closed. The alert remains on the screen until the user selects one of the buttons in the alert.

Custom alert initialization

```
fzrt_error_td  fz_fuim_alrt_ptr_init (
	fz_fuim_alrt_ptr		*fuim_alrt,
	fz_fuim_alrt_flag_enum		flags,
	fz_fuim_alrt_icon_enum		alrt_icon,
	char				*alrt_title
);
```

This function creates the alert pointer. The alert pointer is a **form•Z** opaque data structure used to manage alerts. The pointer is returned in the `fuim_alrt` parameter. The `flags` parameter indicates optional control for the display of the alert. The default value for no options is `FZ_FUIM_ALRT_FLAG_NONE`. The value `FZ_FUIM_ALRT_FLAG_BVRT` can be used to indicate that the buttons in the alert should appear vertically stacked rather than the default horizontal layout. The `alrt_icon` parameter tells **form•Z** which standard icon should be shown in the alert. The valid values are `FZ_FUIM_ALRT_ICON_STOP`, `FZ_FUIM_ALRT_ICON_ASK` and `FZ_FUIM_ALRT_ICON_INFO`. The `alrt_title` parameter is the text for the title of the alert. This is shown in the title bar of the alert dialog. This parameter is optional.

Custom alert strings

```
fzrt_error_td fz_fuim_alrt_ptr_add_str(
	fz_fuim_alrt_ptr		fuim_alrt,
```

```
        long                    flags,
        char                    *str
        );
```

This function adds a string to the alert. The `fuim_alrt` parameter is the alert pointer created by the `fz_fuim_alrt_ptr_init` function. The `flags` parameter is currently not used and should always be set to 0. The str parameter is the text for the string that is to be shown in the alert.

Custom alert buttons

```
fzrt_error_td fz_fuim_alrt_ptr_add_button(
        fz_fuim_alrt_ptr                fuim_alrt,
        long                            button_id,
        fz_fuim_alrt_butn_opts_enum     button_opts,
        fz_fuim_alrt_button_enum        button_kind,
        char                            *str
        );
```

This function adds a button to the alert. The `fuim_alrt` parameter is the alert pointer created by the `fz_fuim_alrt_ptr_init` function. The `button_id` should be set to a unique numeric value for each button. This value is used to identify which button the user selects when the alert is displayed on the screen. The `button_opts` parameter indicates optional control for the button. The value `FZ_FUIM_ALRT_BUT_NONE` is used too indicates no options. The value `FZ_FUIM_ALRT_BUT_DEF` can be used to indicate that the button is the default button. The default button is the button that is selected if the return or enter key is pressed while the alert is displayed on the screen. The value `FZ_FUIM_ALRT_BUT_DEF_CANCEL` can be used to indicate that the button is the cancel button. The cancel button is the button that is selected if the escape (esc) key (or any user defined cancel key shortcut) is pressed while the alert is displayed on the screen. The `button_kind` parameter indicates what title should be used for the button. The following values are available:

`FZ_FUIM_ALRT_BUTTON_OK`: Button is named “OK”.
`FZ_FUIM_ALRT_BUTTON_CANCEL`: Button is named “Cancel”.
`FZ_FUIM_ALRT_BUTTON_YES`: Button is named “Yes”.
`FZ_FUIM_ALRT_BUTTON_NO`: Button is named “No”.
`FZ_FUIM_ALRT_BUTTON_QUIT`: Button is named “Quit”.
`FZ_FUIM_ALRT_BUTTON_CUSTOM`: The title is specified in the `str` parameter.

Custom alert display

```
long fz_fuim_alrt_driver (
        fz_fuim_alrt_ptr                fuim_alrt
        );
```

This function displays the alert on the screen. The `fuim_alrt` parameter is the alert pointer created by the `fz_fuim_alrt_ptr_init` function. The alert remains on the screen until the user selects one of the buttons in the alert. The value returned from this function is the ID of the user sleeted button. The ID is the value of the `button_id` parameter that was used to create the button with the `fz_fuim_alrt_ptr_add_button` function.

Custom alert disposal

```
void fz_fuim_alrt_ptr_finit(
```

```
        fz_fuim_alrt_ptr                    fuim_alrt
        );
```

This function disposes the alert pointer and all memory used by the alert.

The following example shows a custom alert that asks the user if they want to delete selected objects. Note that for clarity the strings in this example are shown directly rather than the preferred method of storing them in .fzr files as described in section 1.4.2.

```
fz_fuim_alrt_ptr            fuim_alrt;
long                        hit;

        /* initalize the alert */
        fz_fuim_alrt_ptr_init(&fuim_alrt, 0, FZ_FUIM_ALRT_ICON_STOP, NULL);

        /* add the message */
        fz_fuim_alrt_ptr_add_str(fuim_alrt, 0,
                "Are you sure you want to delete the selected objects?");

        /* add the "Delete" and "Keep" buttons */
        fz_fuim_alrt_ptr_add_button(fuim_alrt, 1, FZ_FUIM_ALRT_BUT_DEF,
                FZ_FUIM_ALRT_BUTTON_CUSTOM, "Delete");
        fz_fuim_alrt_ptr_add_button(fuim_alrt, 2, FZ_FUIM_ALRT_BUT_DEF_CANCEL,
                FZ_FUIM_ALRT_BUTTON_CUSTOM, "Keep");

        /* display the alert to the user */
        hit = fz_fuim_alrt_driver(fuim_alrt);

        /* dispose the alert */
        fz_fuim_alrt_ptr_finit(&fuim_alrt);

        /* handle the users choice */
        if(hit == 1)
        {
                /* Delete  objects here */
        }
```

2.6.2 Dialogs

Dialogs are invoked by calling a dialog driver function. The driver creates the window for the dialog and calls a FUIM **template function** provided by the plugin to create the content of the dialog. The driver displays the dialog on the screen and the user dismisses handles user interaction with the template until the dialog.

There are three dialog driver functions that work in identical fashion. The three dialog driver variants correspond to the three variants of template functions available as described in the next section. The driver that is used is based on the needs of the template function. By default the driver functions return `FZRT_STD_OK` if an OK button is pressed or `FZRT_STD_CANCEL` if a Cancel button is pressed to dismiss the dialog. The template function can customize the values that are returned by the driver. The tree driver functions are as follows.

```
short fz_fuim_dlog_drive(
        fz_fuim_setup_func          fuim_setup
        );

short fz_fuim_dlog_drive_data(
        fz_fuim_setup_data_func     fuim_setup_data,
```

```
        void                        *data
        );

short fz_fuim_dlog_drive_windex(
        long                        windex,
        fz_fuim_setup_windex_func   fuim_setup_windex,
        void                        *data
        );
```

The first is the basic driver function. The `fuim_setup` parameter is a function pointer for the template function. The second version of the function adds a pointer to plugin supplied data (`data`). The template manager passes on this pointer to the template function. The third variant adds the project window index (`windex`). This parameter is the project window index to be used for project references in the template function.

2.6.3 Template Function

The FUIM template function defines a template by calling **form•Z** API functions to create the interface elements, define relationships between items, and bind the data storage (variables) to the user interface elements. There are three variants of the template function. All three variants function in the same fashion, however they vary in the parameters that they receive.

```
typedef fzrt_error_td  (FZRT_SPEC *fz_fuim_tmpl_func)(
        fz_fuim_tmpl_ptr            tmpl_ptr
        );
```

This is the basic template function. The `tmpl_ptr` parameter is an opaque pointer that is created by **form•Z** and used to manage the template. The template pointer parameter is used as the first parameter to all FUIM API functions. This function should return `FZRT_NOERR` if the template is successfully created. Any other return value indicates that template creation failed.

```
typedef fzrt_error_td  (FZRT_SPEC *fz_fuim_tmpl_data_func)(
        fz_fuim_tmpl_ptr            tmpl_ptr,
        fzrt_ptr                    tmpl_data
        );
```

This is the same as basic template function with the addition of the `tmpl_data` parameter. This parameter is a generic pointer used as a reference to data that is needed by the template. This pointer must be supplied by the function that that is driving the template.

```
typedef fzrt_error_td (FZRT_SPEC *fz_fuim_tmpl_windex_func)(
        long                        windex,
        fz_fuim_tmpl_ptr            tmpl_ptr,
        fzrt_ptr                    tmpl_data
        );
```

This is the same as data template function with the addition of the `windex` parameter. This parameter is the project window index to be used for project references in the template function. This template function variant is used when operating on project or window level data where the windex is needed to access project or window data. The value for windex supplied by the function that that is driving the template.

The first function that should be called inside of a template function is `fz_fuim_tmpl_init`.

```
fzrt_error_td fz_fuim_tmpl_init(
        fz_fuim_tmpl_ptr    fuim_tmpl,
        char                *titl_str,
        short               tmpl_flags,
```

```
fzrt_UUID_td        uuid,
long                version
);
```

This function initializes the template definition. The `fuim_tmpl` parameter is the template pointer. The `titl_str` parameter is the name of the template. For dialogs, this is the title that appears in the title bar of the dialog window. This parameter is not used for palettes. The `tmpl_flags` parameter is currently unused and should always be 0. The `uuid` parameter is the ID of the template. This is an optional parameter. When a UUID is provided, the **form•Z** template manager stores information about the state of the template for reuse each time the template is used. This includes remembering which tab is active for tab elements and items that are collapsed in palettes. The version parameter complements the UUID and is only used when a UUID is provided. This number informs the **form•Z** template manager what version of the template is in use. This number should be set to zero for the first implementation of a template and then increased when changes are made to the implementation of the template (i.e. elements changed, removed or added). This version change informs the template manager that the template has changed and that it should no longer use the saved state from the previous implementation.

2.6.3.1 Element creation and variable association

Each interface element in the template is referred to as a template **item**. Items are referenced by their **ID** that is assigned by the plugin when the item is created. IDs must be unique within each template. All items except groups, dividers, and images have are said to have a **value**. The value can be a **specific** numeric value or a **range** of values depending on the interface element. Items that have values can associate a plugins **variable** with the item. When the user changes the interface element, the associated variable is updated to the defined value.

The next section describes the common aspects of template item creation. The following section describes how variables are associated with items. The remainder of the sections describes each type of element, the function that is used to create the item and what types of association are supported.

Item creation

There is a single function for creating an item of each type of interface element. All of the creation functions return the ID of the new item. If the item can not be created, the value `FZ_FUIM_NONE` is returned. All of the item creation functions start with `fz_fuim_new_` and contain the following common parameters:

```
fz_fuim_tmpl_ptr    fuim_tmpl
```

The `fuim_tmpl` parameter is the template pointer.

```
short               parent
```

The `parent` parameter is the ID of the parent item of the item being created. The value `FZ_FUIM_ROOT` should be used if the item is at the top of the template's hierarchy.

```
short               id,
```

The `id` parameter is the ID of item being created. This value must be unique within each template and be in the range of 0 to 32767. The template manager can create a unique ID automatically by specifying the value `FZ_FUIM_NONE` for this parameter. The generated ID is returned as the return value from the item creation function. Automatic generation should NOT be used when the `item_func` parameter is used as the item function needs to have predefined IDs to function properly.

```
long                flags
```

The `flags` parameter is a bit encoded parameter that specifies optional control for the item being created. These values should be combined using the bitwise or (|)operator (e.g. `FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_SMAL`). The following values are supported:

`FZ_FUIM_FLAG_NONE`: Indicates no flags.
`FZ_FUIM_FLAG_HORZ`: Indicates that the child items of the new item should have a horizontal layout. If this is not specified, they have the default vertical layout.
`FZ_FUIM_FLAG_BRDR`: Indicates that the item should be drawn with a boarder around it.
`FZ_FUIM_FLAG_INDT`: Indicates that the item's position should be indented from the position of its parent. The indentation moves the item towards the right if it is in a vertical layout and towards the bottom if it is in a vertical layout.
`FZ_FUIM_FLAG_GFLT`: Indicates that the sibling items of the new item should have a horizontal layout next to the new item.
`FZ_FUIM_FLAG_HTOP`: Items in a horizontal layout are by default center aligned. If this value is provided, all of the child items that are in a horizontal layout will be bottom aligned. Should not be used with `FZ_FUIM_FLAG_HBOT`.
`FZ_FUIM_FLAG_HBOT`: Items in a horizontal layout are by default center aligned. If this value is provided, all of the child items in a horizontal layout will be bottom aligned. Should not be used with `FZ_FUIM_FLAG_HTOP`.
`FZ_FUIM_FLAG_VCNT`: Items in a vertical layout are by default left aligned. If this value is provided, all of the child items in a vertical layout will be center aligned. Should not be used with `FZ_FUIM_FLAG_VRGT`.
`FZ_FUIM_FLAG_VRGT`: Items in a vertical layout are by default left aligned. If this value is provided, all of the child items in a vertical layout will be right aligned. Should not be used with `FZ_FUIM_FLAG_VCNT`.
`FZ_FUIM_FLAG_SMAL`: Indicates that the item should be shown in a reduced width.
`FZ_FUIM_FLAG_EQSZ`: Indicates that all of the child item should be shown made to be the same size. The size of the largest child is calculated and all child items are set to be the same size.
`FZ_FUIM_FLAG_JRGT`: Indicates that the new item should be right justified. If this is not set then the default left justification is used.
`FZ_FUIM_FLAG_DIMM`: Indicates that the item should be shown always dimmed and inactive.
`FZ_FUIM_FLAG_FRAM`: Indicates that a boarder should be drawn around all of the child items of the new item.
`FZ_FUIM_FLAG_PASS`: This is a special flag only used by text items. It indicates that the text is a password field and it should not show the text directly. When this option is selected, the text is shown with a "*" for each character in the string.

```
fz_fuim_item_func   item_func
```

The `item_func` parameter is an optional parameter for a function pointer to an item function. The template manager calls the item function at various times to either to retrieve information about the item or notify the item about an action related to the item.

The item function should return TRUE the action was handled and FALSE if it was not. When the function handles the action, then the template manager does not. The item function has the following prototype and parameters:

```
typedef short fz_fuim_item_func(
      fz_fuim_tmpl_ptr                  fuim_tmpl,
      long                              action,
      short                             item_id,
      void                              *item_data,
      fz_fuim_type_td                   *action_data
      );
```

`fz_fuim_tmpl_ptr    fuim_tmpl`

The `fuim_tmpl` parameter is the template pointer. The action parameter is the action that is being requested or sent to the item function. The following actions are currently supported.

`long                action`

`FZ_FUIM_ACTN_GET_ACTIV`: This action is sent by the template manager to find out if the item is active or not. This message is useful to make items dependent on each other, which are not descendants of each other in the template's hierarchy. An active item is indicated by the value of 0 in the `fuim_short` field of the `item_data` parameter (`item_data->fuim_short == 0`). An inactive item is indicated by the value of 255 in the `fuim_short` field of the `item_data` parameter(`item_data->fuim_short == 255`).
`FZ_FUIM_ACTN_BLESS`: This action is sent by the template manager for button items to find out if the item should be blessed as the default OK action for a dialog or the default cancel action for a dialog. The OK action is executed when the return or enter key is pressed in a dialog. The default cancel action is executed when the <esc> key or a cancel key shortcut is selected. By default the OK button and Cancel button perform these actions. A different button can be assigned these actions by returning `FZRT_STD_OK` or `FZRT_STD_CANCEL` in the `fuim_short` field of the `item_data` parameter.
`FZ_FUIM_ACTN_GET_VALUE`: This action is sent by the template manager to get the value of an item. The value should be returned in the `item_data` parameter. The field corresponding to the type of variable that is associated with the item should be used (i.e. if the associated variable is of type long, the `fuim_long` field should be used).
`FZ_FUIM_ACTN_PICT_SCALE`: This action is sent by the template manager for picture items to desired scale for the item. The desired scale value should be returned in the `fuim_float` field of the `item_data` parameter.
`FZ_FUIM_ACTN_HIT`: This action is sent to the item function when an item is hit (clicked on). This is most useful for button items when some action needs to be performed to handle the button click.
`FZ_FUIM_ACTN_SET_VALUE`: This action indicates that the item's value should be set to the provided value in the `data` parameter. The field corresponding to the type of variable that is associated with the item should be used (i.e. if the associated variable is of type long, the `fuim_long` field should be used).
`FZ_FUIM_ACTN_NEW_VALUE`: This action indicates that the item's value has changed.

`short               item_id`

The `item_id` parameter is the ID of item being processed by the item function.

```
void                *item_data
```

The `item_data` parameter is a pointer to item specific data that is defined when the item is created. This pointer is stored by the template manager and provided to the item function.

```
fz_fuim_type_td     *action_data
```

The `action_data` parameter is a pointer to a union of the basic types supported by the template manager. This parameter is used to pass data in and out the item function. The field that contains the data is dependent on the value of the action parameter as described above. The following is the definition of the union:

```
typedef union fz_fuim_type_td
{       fzrt_boolean        fuim_boolean;
        char                fuim_char;
        unsigned char       fuim_uchar;
        short               fuim_short;
        unsigned short      fuim_ushort;
        long                fuim_long;
        unsigned long       fuim_ulong;
        float               fuim_float;
        double              fuim_double;
} fz_fuim_type_td;
```

```
void          *item_data
```

The `item_data` parameter is only used when the `item_func` parameter is provided. This parameter is optional. When it is provided, the template manager passes it into the `item_data` parameter of the item function.

Most of the functions also contain a `titl_str` parameter. This string is the title of the item in the template. It is recommended that the strings be stored in .fzr files and loaded from this file so that they can be localized.

Variable association

Most FUIM items displayed to the user have some sort of input or value associated with them, which the user can change, usually within some range of valid values. This means that variables must be associated with these FUIM items. A variable's value can be a specific value or a range of values. Items that have values

Unary

Specific values

Specific values are used for interface elements that are binary. That is, they only have two states ; on (TRUE or 1) and off (FALSE or 0). These are check boxes, radio buttons, icons and custom items (depending on the implementation). There are three methods for defining the specific values: unary, binary and encoded. In the unary case the FALSE value is always 0 and the TRUE value is supplied by the plugin. In a binary case, both the FALSE value and the TRUE value are supplied. The encoded method compares the variable with a supplied bit mask. That is, the

FALSE value is occurs when all of the masked bits are off in the variable and TRUE is defined when all of the bits are on.

There are 21 functions that are used to associated a specific value; seven for each method

```
fz_fuim_item_unary_bool    fz_fuim_item_binary_bool   fz_fuim_item_encod_bool
fz_fuim_item_unary_char    fz_fuim_item_binary_char   fz_fuim_item_encod_char
fz_fuim_item_unary_uchar   fz_fuim_item_binary_uchar  fz_fuim_item_encod_uchar
fz_fuim_item_unary_short   fz_fuim_item_binary_short  fz_fuim_item_encod_short
fz_fuim_item_unary_ushort  fz_fuim_item_binary_ushort fz_fuim_item_encod_ushort
fz_fuim_item_unary_long    fz_fuim_item_binary_long   fz_fuim_item_encod_long
fz_fuim_item_unary_ulong   fz_fuim_item_binary_ulong  fz_fuim_item_encod_ulong
```

All of the functions for each method have the same parameters and work identically. Each variant is provided for the type of the variable that is being associated (long, short etc.). For example if the plugin variable is a short, then the function `fz_fuim_item_unary_short`, or `fz_fuim_item_binary_short`, or `fz_fuim_item_encod_short` is used.

```
typedef void  fz_fuim_item_unary_short(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               item_id,
       short               *data_ptr,
       short               true_value
       );
```

The `fuim_tmpl` parameter is the template pointer where the item is found. The `item_id` parameter is the ID of the item that is being associated. The `data_ptr` parameter is the pointer to the plugin variable that is being associated. The type for this variable matches the type specified in the function name. The `true_value` parameter is the value that the variable (`*data_ptr`) must have for the element to be in its TRUE state. That is when `*data_ptr == true_value`, the items value is TRUE and when `*data_ptr != true_value,` the item's value is FALSE.

```
typedef void  fz_fuim_item_binary_short(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               item_id,
       short               *data_ptr,
       short               true_value,
       short               false_value
       );
```

The `fuim_tmpl` parameter is the template pointer. The `item_id` parameter is the ID of the item that is being associated. The `data_ptr` parameter is the pointer to the plugin variable that is being associated. The type for this variable matches the type specified in the function name. The `true_value` parameter is the value that the variable (`*data_ptr`) must have for the element to be in its TRUE state. That is when `*data_ptr == true_value`, the items value is TRUE. The `false_value` parameter is the value that the variable (`*data_ptr`) must have for the element to be in its FALSE state. That is when `*data_ptr == false_value`, the item's value is FALSE.

```
typedef void fz_fuim_item_encod_short(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               item_id,
       short               *data_ptr,
       fzrt_boolean        true_value,
       short               bit_mask
       );
```

The fuim_tmpl parameter is the template pointer. The item_id parameter is the ID of the item that is being associated. The data_ptr parameter is the pointer to the plugin variable that is being associated. The type for this variable matches the type specified in the function name. The true_value parameter is the value (TRUE or FALSE) that the variable (*data_ptr) when masked with the bit mask (bit_mask) for the element to be in its TRUE state. That is when (*data_ptr & bit_mask) == true_value, the items value is true and when (*data_ptr & bit_mask)!= true_value, the item's value is FALSE. Note that the macro FZ_FUIM_BIT_TO_MASK is provided for turning bit values in into a bit mask.

Range values

Range association is used for interface elements that can represent more than a single specific value. These are menus, sliders, scroll bars, tabs, frames , text fields and custom items (depending on the implementation). There are nine functions that are used to associate a specific value to an item. Six of these items are used for integer values:

```
fz_fuim_item_range_char
fz_fuim_item_range_uchar
fz_fuim_item_range_short
fz_fuim_item_range_ushort
fz_fuim_item_range_long
fz_fuim_item_range_ulong
```

All of these functions have the same parameters and work identically. Each variant is provided for the type of the variable that is being associated. For example if the plugin variable is a short, then the function fz_fuim_item_range_short is used.

```
void fz_fuim_item_range_short(
        fz_fuim_tmpl_ptr            fuim_tmpl,
        short                       item_id,
        short                       *data_ptr,
        short                       min_value,
        short                       max_value,
        fz_fuim_format_int_enum     format,
        short                       flags
        );
```

The fuim_tmpl parameter is the template pointer. The item_id parameter is the ID of the item that is being associated. The data_ptr parameter is the pointer to the plugin variable that is being associated. The type for this variable matches the type specified in the function name. The min_value parameter is the minimum value for the range and max_value parameter is the maximum value. The type for these variables matches the type specified in the function name. The format parameter is used if the associated item contains a text string. The current values for this parameter are as follows:

FZ_FUIM_FORMAT_INT_DEFAULT: The value is displayed as a whole number in decimal notation
FZ_FUIM_FORMAT_INT_DATE_NATIVE: The value is displayed as a date (day, month and year) according to the current date format of the OS.
FZ_FUIM_FORMAT_INT_TIME_NATIVE: The value is displayed as a time of day (hours, minutes and seconds), according to the current date format of the OS.
FZ_FUIM_FORMAT_INT_DURN_HHMMSS: The value is displayed as a duration (hours, minutes and seconds), separated by colons.

The flags parameter can be used to add additional control as follows:

`FZ_FUIM_RANGE_NONE`: no flags (default).
`FZ_FUIM_RANGE_MIN`: Clamp input to the specified minimum value in text fields. This prevents a value less than the specified minimum value from being entered by the user. If the user enters a smaller value, it will be changed to the minimum.
`FZ_FUIM_RANGE_MIN_INCL`: The specified minimum value is inclusive. If this is not set it is exclusive.
`FZ_FUIM_RANGE_MAX`: Clamp input to the specified maximum value in text fields. This prevents a value greater than the specified maximum value from being entered by the user. If the user enters a larger value, it will be changed to the maximum.
`FZ_FUIM_RANGE_MAX_INCL`: The specified maximum value is inclusive. If this is not set it is exclusive.

There are two functions used for floating point values:

```
fz_fuim_item_range_float
fz_fuim_item_range_double
```

All of these functions have the same parameters and work identically. Each variant is provided for the type of the variable that is being associated. For example if the plugin variable is a double, then the function `fz_fuim_item_range_double` is used.

```
void fz_fuim_item_range_double(
       fz_fuim_tmpl_ptr           fuim_tmpl,
       short                      item_id,
       double                     *data_ptr,
       double                     min_value,
       double                     max_value,
       fz_fuim_format_float_enum  format,
       short                      flags
       );
```

All of the parameters are the same as the integer variations except for `format`. The `format` parameter is used if the associated item contains a text string. The following are currently supported:

`FZ_FUIM_FORMAT_FLOAT_DEFAULT`: The floating-point value is displayed as a fraction, with the whole and fractional part of the number separated by a decimal point.
`FZ_FUIM_FORMAT_FLOAT_DISTANCE`: floating point value is displayed as a distance value. The formatting is determined by the setting in the Working Units dialog. For example, when English units are selected the default linear distances are displayed with the feet and inch notation.
`FZ_FUIM_FORMAT_FLOAT_ANGLE`: The floating-point value is displayed as an angle. The variable's value is expected to be in radians. The display of an angle is shown in degrees in the text field.
`FZ_FUIM_FORMAT_FLOAT_PERCENT`: The floating-point value is displayed as a percentage value. That is, the variable's value is multiplied by 100 before it is displayed in the text field. This allows a value to be stored in a variable in a normalized range (0.0 to 1.0) but display it to the user as a percentage (0.0 to 100.0).
`FZ_FUIM_FORMAT_FLOAT_CURRENCY_NATIVE`: The floating point value is displayed as a currency, formatted according to the language setting of the operating system.
`FZ_FUIM_FORMAT_FLOAT_AREA_SQIN`: The floating point value is displayed as an area value, converted to and formatted as square inches. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
`FZ_FUIM_FORMAT_FLOAT_AREA_SQFT`: The floating point value is displayed as an area value, converted to and formatted as square feet. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.

FZ_FUIM_FORMAT_FLOAT_AREA_SQYD: The floating point value is displayed as an area value, converted to and formatted as square yards. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_AREA_ACRE: The floating point value is displayed as an area value, converted to and formatted as acres. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_AREA_SQMI: The floating point value is displayed as an area value, converted to and formatted as square miles. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_AREA_SQMM: The floating point value is displayed as an area value, converted to and formatted as square millimeters. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_AREA_SQCM: The floating point value is displayed as an area value, converted to and formatted as square centimeters. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_AREA_SQMT: The floating point value is displayed as an area value, converted to and formatted as square meters. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_AREA_SQKM: The floating point value is displayed as an area value, converted to and formatted as square kilometers. The floating point value is assumed to represent square inches in an english project or square centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_CUIN: The floating point value is displayed as a volumetric value, converted to and formatted as cubic inches. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_CUFT: The floating point value is displayed as a volumetric value, converted to and formatted as cubic feet. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_CUYD: The floating point value is displayed as a volumetric value, converted to and formatted as cubic yards. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_GAL: The floating point value is displayed as a volumetric value, converted to and formatted as gallons. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_CUMM: The floating point value is displayed as a volumetric value, converted to and formatted as cubic millimeters. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_CUCM: The floating point value is displayed as a volumetric value, converted to and formatted as cubic centimeters. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_VOLUME_LITR: The floating point value is displayed as a volumetric value, converted to and formatted as liters (cubic decimeter). The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.

FZ_FUIM_FORMAT_FLOAT_VOLUME_CUMT: The floating point value is displayed as a volumetric value, converted to and formatted as cubic meters. The floating point value is assumed to represent cubic inches in an english project or cubic centimeters in a metric project.
FZ_FUIM_FORMAT_FLOAT_WEIGHT_OZ: The floating point value is displayed as a weight value, converted to and formatted as ounces. The floating point value is assumed to represent grams.
FZ_FUIM_FORMAT_FLOAT_WEIGHT_LBS: The floating point value is displayed as a weight value, converted to and formatted as pounds (16 ounces). The floating point value is assumed to represent grams.
FZ_FUIM_FORMAT_FLOAT_WEIGHT_ETON: The floating point value is displayed as a weight value, converted to and formatted as english tons. The floating point value is assumed to represent grams.
FZ_FUIM_FORMAT_FLOAT_WEIGHT_G: The floating point value is displayed as a weight value, converted to and formatted as grams. The floating point value is assumed to represent grams.
FZ_FUIM_FORMAT_FLOAT_WEIGHT_KG: The floating point value is displayed as a weight value, converted to and formatted as kilo grams. The floating point value is assumed to represent grams.
FZ_FUIM_FORMAT_FLOAT_WEIGHT_MTON: The floating point value is displayed as a weight value, converted to and formatted as metric tons. The floating point value is assumed to represent grams.

There is one function for strings:

```
void fz_fuim_item_range_string(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               item_id,
       char                *data_ptr,
       short               min_value,
       short               max_value,
       short               flags
       );
```

The `fuim_tmpl` parameter is the template pointer. The `item_id` parameter is the ID of the item that is being associated. The `data_ptr` parameter is the pointer to the plugin string variable that is being associated. The `min_value` parameter is the minimum number of characters in the string and `max_value` is maximum number of characters in the string. The `flags` parameter is as in the integer range functions.

Check box

```
short fz_fuim_new_check(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       char                *titl_str,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

A check box is an interface element that can be in either an "on" (true/1) or "off" (false/0) state. Clicking on a check box changes its state from "on" to "off", or from "off" to "on". The title string is shown to the right of the check box graphic. Variables are associated with check box items using `fz_fuim_item_unary_*`, `fz_fuim_item_binary_*`, or `fz_fuim_item_encod_*` functions.

The following is an example of a check box with a short value associated with it such that the check is on when the variable is 2 and off when the variable is 1.

```
short  item;
short  my_variable = 2;

/* Create a check box item */
item = fz_fuim_new_check(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, "My Check Box", NULL, NULL);

/* Associate my_variable with the item, */
/* my_variable == 2 for check on, my_variable == 1 for off */
fz_fuim_item_binary_short(fuim_tmpl, item, &my_variable, 2, 1);
```

Radio button

```
short fz_fuim_new_radio(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       char                *titl_str,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

Radio buttons are like check boxes except that they are used in a set and are mutually exclusive in the set: when one is switched "on", all others in the set are switched "off". This function creates a single radio button. A set of radio buttons is defined by the creation of each button in the set and then associating them with the same variable (see next section on binding). The title string is shown to the right of the radio button graphic. Variables are associated with check box items using `fz_fuim_item_unary_*`, `fz_fuim_item_binary_*`, or `fz_fuim_item_encod_*` functions.

The following is an example of a three radio buttons with a short value associated with them such that the radio buttons are mapped to the values of 2, 3, and 7. That is when the first button is selected, the variable is set to 2, when the second is selected the variable is set to 3 and when the third is selected the variable is set to 7.

```
short  item;
short  my_variable = 2;

/* Create a radio button box item and variable with the item with a value of 2
*/
item = fz_fuim_new_radio(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, "My Radio 1", NULL, NULL);
fz_fuim_item_unary_short(fuim_tmpl, item,  &my_variable, 2);

/* Create a radio button box item and variable with the item with a value of 3
*/
item = fz_fuim_new_radio(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, "My Radio 2", NULL, NULL);
fz_fuim_item_unary_short(fuim_tmpl, item,  &my_variable, 3);

/* Create a radio button box item and variable with the item with a value of 7
*/
item = fz_fuim_new_radio(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
```

```
                    FZ_FUIM_FLAG_NONE, "My Radio 3", NULL, NULL);
fz_fuim_item_unary_short(fuim_tmpl, item,  &my_variable, 7);
```

Button

```
short fz_fuim_new_button(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       char                *titl_str,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

Buttons are interface items that perform an action when they arc clicked on. The action is handled in the `FZ_FUIM_ACTN_HIT` action of the `item_func` described in a following section. The title string is shown in graphics of the button. This item can not be associated with a variable as it does not change in value.

The following is an example of a button.

```
short  item;

/* Create a button item */
item = fz_fuim_new_button(fuim_tmpl, FZ_FUIM_ROOT, 100 /* item id */,
                    FZ_FUIM_FLAG_NONE, "My Button", my_button_func, NULL);
```

With the following item function to handle the click in the button.

```
short my_button_func (
       fz_fuim_tmpl_ptr                  fuim_tmpl,
       long                              action,
       short                             item_id,
       void                              *item_data,
       fz_fuim_type_td                   *action_data
       )
{
       short rv = FALSE;

       switch(item_id)
       {      case 100:
                     if(action == FZ_FUIM_ACTN_HIT)
                     {
                            /* Handle hit here */

                            rv = TRUE;
                     }
              break;
       }

       return(rv);
}
```

Static text

```
short fz_fuim_new_text_static(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
```

```
        char                *titl_str,
        fz_fuim_item_func   item_func,
        void                *item_data
        );
```

Static text items are single line strings that are used for information, labels, or titles for sub-groups in a template. The user can not change static text items. This item can not be associated with a variable as it does not change in value.

The following is an example of static text.

```
short  item;

/* Create static text item */
item = fz_fuim_new_text_static (fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, "My Static Text", NULL, NULL);
```

Editable text

```
short fz_fuim_new_text_edit(
        fz_fuim_tmpl_ptr    fuim_tmpl,
        short               parent,
        short               id,
        long                flags,
        char                *titl_str,
        fz_fuim_item_func   item_func,
        void                *item_data
        );
```

Editable text items are strings that can be changed by the user. They are used for numeric fields and string fields. If a numeric variable is associated with the edit text item, then the edit text will shown a numeric value and accept numeric input. If a character variable is associated with the edit text item, then the edit text will shown the string and accept character input. The title for the edit text is shown to the left with the editable area in a box to the right. Variables are associated with check box items using `fz_fuim_item_range_` functions.

The following is an example of editable text for a short variable with a range of 0 to 20.

```
short  item;
short  my_variable = 0;

/* Create editable text item */
item = fz_fuim_new_text_edit(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, NULL, NULL);
fz_fuim_item_range_short(fuim_tmpl, item, &my_variable, 0, 20,
                    FZ_FUIM_FORMAT_INT_DEFAULT,
                    FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
                    FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL );
```

The following is an example of editable text for a double variable which must be greater than zero.

```
short  item;
double my_variable = 1.0;

/* Create editable text item */
item = fz_fuim_new_text_edit(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, NULL, NULL);
fz_fuim_item_range_double(fuim_tmpl, item, &my_variable, 0.0, 0.0,
                    FZ_FUIM_FORMAT_FLOAT_DEFAULT,
```

```
                    FZ_FUIM_RANGE_MIN);
```

The following is an example of editable text for a string.

```
short  item;
char   my_string[256];

stcpy(my_string, “Initial String”);
/* Create editable text item */
item = fz_fuim_new_text_edit (fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, NULL, NULL);
fz_fuim_item_range_string(fuim_tmpl, item, my_string, 0, 256,
                    FZ_FUIM_RANGE_NONE);
```

Editable text can also be created with the function `fz_fuim_new_text_info`. This is a variant of the edit text item which includes additional control over the item’s dimensions.

```
short fz_fuim_new_text_edit_info(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       char                *titl_str,
       short               width,
       short               height,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

The width parameter is the desired width of the item in the template If this value is 0, then the default width is used. The height is the number of lines high the text should be. This actual size will vary with the user selected dialog or palette font size.

The following is an example of editable text for a double variable which is 75 pixels and one line high.

```
short  item;
double my_variable;

/* Create editable text item */
item = fz_fuim_new_text_edit_info (fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, “My Edit Text”, 72, 1, NULL, NULL);
fz_fuim_item_range_double(fuim_tmpl, item, &my_variable, 0.0, 0.0,
                    FZ_FUIM_FORMAT_FLOAT_DEFAULT,
                    FZ_FUIM_RANGE_MIN);
```

Note

```
short fz_fuim_new_note(
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       char                *titl_str,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

A note is like a static text item except that it supports multiple lines. Note are used for detailed information. The user can not change these items. This item can not be associated with a variable as it does not change in value. Variables are associated with check box items using `fz_fuim_item_range_*` functions.

```
/* Create note item */
item = fz_fuim_new_note(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
                    FZ_FUIM_FLAG_NONE, "My Note", NULL, NULL);
```

Menu

```
short fz_fuim_new_menu (
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       char                *titl_str,
       fzrt_menu_ptr       menu,
       fzrt_boolean        is_pop,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

A menu is a list of items form which items can be selected. A menu can be a regular menu or a pop-up menu. In regular menu, the menu has one active item. The active item is shown in the template and when the item is selected, the fill menu is displayed so that a new active item can be selected. A pop-up menu is shown in the template as a small triangle. When the triangle is selected, the menu is displayed and one of the items can be selected. As there is no active item, this type of menu is useful when the selection of the item performs an action (like loading preset values) or if the menu contains a series of on/off type of settings and the selection of an item toggles its state.

The `menu` parameter is a menu pointer for the menu. The menu loaded from and .fzr file using the `fzrt_fzr_get_menu` function. The menu can also be constructed by using `fzrt_new_menu` to create the menu and `fzrt_menu_append_item_text` to add items and `fzrt_menu_append_seperator` to add separators. If the `is_pop` parameter is set to TRUE then the menu is a pop-up menu and when it is set to FALSE, it is a regular menu.

Variables are associated with menu items using integer `fz_fuim_item_range_*` functions. The following is an example of menu variable with a range of 0 to 6. Menus are implicitly clamped at the range limits if one uses the `FZ_FUIM_RANGE_NONE` range flag. Otherwise, only the inclusive range flags are useful for menus (`FZ_FUIM_RANGE_MIN_INCL` and `FZ_FUIM_RANGE_MAX_INCL`).

```
short          item;
short          my_variable = 0;
fzrt_menu_ptr my_menu;

       /* create menu manager */
       my_menu = fzrt_menu_create("");
       fzrt_menu_append_item_text(my_menu, "Veggies");
       fzrt_menu_append_item_text(my_menu, "Meat");
       fzrt_menu_append_item_text(my_menu, "Dairy");
       fzrt_menu_append_separator(my_menu);
       fzrt_menu_append_item_text(my_menu, "Beer");
       fzrt_menu_append_item_text(my_menu, "Juice");
       fzrt_menu_append_item_text(my_menu, "Wine");

       /* create menu fuim item */
       item = fz_fuim_new_menu(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
              FZ_FUIM_FLAG_NONE, "My Edit Menu", my_menu, FALSE, NULL, NULL);
```

```
        /* associate a variable to drive which is current selection */
        fz_fuim_item_range_short(fuim_tmpl, item, &my_variable, 0, 6,
              FZ_FUIM_FORMAT_INT_DEFAULT, FZ_FUIM_RANGE_NONE);
```

Slider

```
short fz_fuim_new_slider(
        fz_fuim_tmpl_ptr    fuim_tmpl,
        short               parent,
        short               id,
        long                flags,
        char                *titl_str,
        fz_fuim_item_func   item_func,
        void                *item_data
        );
```

A slider is a graphic control useful for setting a value that has a specific range. The slider has an indicator that shows the current value of the slider. The user changes the value of the slider to the desired value by dragging the indicator. Variables are associated with slider items using either the integer or floating-point `fz_fuim_item_range_*` functions.

Icon

```
short fz_fuim_new_icon (
        fz_fuim_tmpl_ptr    fuim_tmpl,
        short               parent,
        short               id,
        long                flags,
        fzrt_floc_ptr       floc,
        long                hpos,
        long                vpos,
        fzrt_floc_ptr       floc_mask,
        long                hpos_mask,
        long                vpos_mask
        fz_fuim_item_func   item_func,
        void                *item_data
        );
```

Icons are like buttons except that they have a graphic image instead of a title. Like buttons they perform an action when they arc clicked on. The action is handled in the FZ_FUIM_ACTN_HIT action of the item_func described in a following section.

The icon image can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available.

The floc parameter should be filled with the file name and location of the file that contains the icon graphic. The hpos and vpos parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the plugin file. This makes it simple to find the file. The location of the plugin file can be retained during the FZPL_PLUGIN_INITIALIZE stage using the fsf->fzpl_plugin_file_get_floc function.

The floc_mask parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the floc parameter). The icon mask defines the transparent areas of the icon. The hpos_mask and vpos_mask parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided then the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of commands by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

Image

```
short fz_fuim_new_image (
      fz_fuim_tmpl_ptr    fuim_tmpl,
      short               parent,
      short               id,
      long                flags,
      fzrt_floc_ptr       floc,
      double              scale,
      fz_fuim_item_func   item_func,
      void                *item_data
      );
```

Images are static graphic elements. The `floc` parameter is the file location and name of the image file. The image file can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. The `scale` parameter is a scale factor that is applied to the image. A value of 1.0 indicates a scale of 100%. Other values scale the image up or down accordingly. This item can not be associated with a variable as it does not change in value.

Group

```
short fz_fuim_new_group (
      fz_fuim_tmpl_ptr    fuim_tmpl,
      short               parent,
      short               id,
      long                flags
      );
```

Groups are items that are used to organize items. This item can not be associated with a variable since it does not change in value. To associate items within the same group, the group id should be passed as the parent id to FUIM items created after the group. An example of a useful flag for a group is one that organizes its items vertically (default) or horizontally, or puts a border around the group. Groups can be organized hierarchically as well, having a group be a parent to many child groups and other items.

Tab

```
short fz_fuim_new_tab_group (
      fz_fuim_tmpl_ptr    fuim_tmpl,
      short               parent,
      short               id,
      long                flags,
      fz_fuim_item_func   item_func,
      void                *item_data
      );
```

A tab is used to organize information in a template into categories such that only one of the categories is shown at a given time. Each of the categories is represented by a title that is placed in a tab at the top of the interface element. The tab is a graphic that mimics the tab that would be found on a file folder. When a tab is clicked on, its contents are shown in the body of the tab interface element. This function simply creates the tab group. To construct the tab, the descendents of this item must be created in a certain fashion. Each child item of the tab item establishes an entry in the tab element. The children of the tab entries, are the contents of each tab. An integer variable should be associated with the tab group to determine which tab is actively viewable.

Frame

```
short fz_fuim_new_frame_group (
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags,
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

A frame functions like tab group except that does not have any graphics. That is, there are a number of categories of information in the frame that are all displayed in the same area of the template. The selection of the active category is driven by another interface element such as a menu or radio button. An integer variable should be associated with the tab group to determine which tab is actively viewable.

Divider

```
short fz_fuim_new_divider (
       fz_fuim_tmpl_ptr    fuim_tmpl,
       short               parent,
       short               id,
       long                flags
       fz_fuim_item_func   item_func,
       void                *item_data
       );
```

A divider is a graphic line drawn across the item. By default divider is drawn horizontally. If the value `FZ_FUIM_FLAG_HORZ` is set in the `flags` parameter, then the line is drawn vertically. This item can not be associated with a variable as it does not change in value.

Custom Items

```
short fz_fuim_new_custom(
       fz_fuim_tmpl_ptr           fuim_tmpl,
       short                      parent,
       short                      id,
       long                       flags,
       char                       *titl_str,
       fz_fuim_item_cust_func     cust_func,
       void                       *cust_data
       );
```

In addition to the normal parameters, a custom item may take a `cust_func` and `cust_data`. A custom item is an item whose appearance and behavior is defined by the plugin. The `cust_func` parameter replaces the standard item function. The template manager calls the custom function at various times to either get information about the custom item or to send notification of an action that needs to be handled by the custom item. Optionally, if needed, one may pass in a pointer to some data, which could then be accessed in the custom item function, through the `cust_data` parameter. The custom item function should return TRUE if the action was handled and FALSE if it was not. When the function handles the action, then the template manager does not. The item function has the following prototype and parameters:

```
typedef short fz_fuim_item_cust_func(
	fz_fuim_tmpl_ptr		fuim_tmpl,
	short				item_id,
	fz_fuim_actn_cust_enum		action,
	fz_type_td			*action_data,
	void				*func_data,
	void				**prvt_data
	);
```

The `fuim_tmpl` parameter is the template pointer. The `item_id` parameter is the id of the item for the current action. This is provided so that the same custom function can be used for multiple items in the same template and still be able to identify the item that is being processed. The `action` parameter is the action that is being requested or sent to the item function. The following actions are currently supported.

`FZ_FUIM_ACTN_CUST_INIT`: This action is sent when the item is created so that any necessary initialization can be performed.

`FZ_FUIM_ACTN_CUST_FINIT`: This action is sent when the item is destroyed so that any necessary de-allocation or cleanup can occur.

`FZ_FUIM_ACTN_CUST_GET_WIDTH_MIN`: This action is to retrieve the minimum width for the item. This should be returned in the `action_data` parameter as a short.

`FZ_FUIM_ACTN_CUST_GET_HEIGHT_MIN`: This action is to retrieve the minimum height for the item. This should be returned in the `action_data` parameter as a short.

`FZ_FUIM_ACTN_CUST_GET_WIDTH`: This action is to retrieve the preferred width for the item. If there is enough space for the preferred height, the value minimum width will be used. This should be returned in the `action_data` parameter as a short.

`FZ_FUIM_ACTN_CUST_GET_HEIGHT`: This action is to retrieve the preferred height for the item. If there is not enough space for the preferred height, the value minimum height will be used. This should be returned in the `action_data` parameter as a short.

`FZ_FUIM_ACTN_CUST_REDRAW`: This action indicates that the custom item should draw its graphic image. The function `fz_fuim_item_get_rect` should be called to retrieve the rectangle in the template in which the image should be drawn. The custom function should draw the item in an appropriate image if the item is inactive.

`FZ_FUIM_ACTN_CUST_CLICK`: This action indicates that a click occurred in the item. The point parameter in the `action_data` contains the cursor's coordinates.

`FZ_FUIM_ACTN_CUST_HILITE`: This action indicates that the item has become active or inactive. This is usually because of a parent item becoming active or inactive. The short parameter in the `action_data` is 0 if the item is active and 255 if it is inactive.

`FZ_FUIM_ACTN_CUST_REPOSITION`: This action indicates that the item has been moved to a new location. This message is always sent once right after initialization to indicate the initial position. Template items rarely move once the FUIM template manager positions them. The function `fz_fuim_item_get_rect` should be called to retrieve the rectangle in the template in which the item is located.

`FZ_FUIM_ACTN_CUST_MOUSE_MOVED_IN`: This action indicates that the mouse moved into the item. The cursor is now inside the item's rectangle.

FZ_FUIM_ACTN_CUST_MOUSE_MOVED_OUT: This action indicates that the mouse moved out of the item. The cursor is now outside the item's rectangle.

FZ_FUIM_ACTN_CUST_MOUSE_MOVED: This action indicates that the mouse moved while inside the items rectangle. The point parameter in the action_data contains the cursor's coordinates. Note that these are in global (screen) coordinates. The function fzrt_global_to_local should be called to convert the coordinate into the template's window coordinates. The function fz_fuim_item_get_rect can be called to retrieve the rectangle in the template in which the item is located.

FZ_FUIM_ACTN_CUST_KEY_DOWN: This action indicates that a key was pressed in the template. The key parameters are accessed by extracting a pointer to a fz_fuim_key_td variable from the pointer parameter of the action_data parameter.

FZ_FUIM_ACTN_CUST_MODF_KEY: This action indicates that a modifier was pressed in the template. The modifier keys are shift, control, option and command for the Macintosh and shift, alt and ctrl for Windows. The modifier keys state can be accessed using the function fzrt_get_keys.

FZ_FUIM_ACTN_CUST_NEW_VAL_INVAL: This action is used to find out if the item needs to be invalidated for redraw. This should be returned in the action_data parameter as a short.

FZ_FUIM_ACTN_CUST_TEXT_FOCUS_GET: This action is used to find out if the item has text focus. If the item has focus, then key strokes in the template will be handled by the item (FZ_FUIM_ACTN_CUST_KEY_DOWN). This should be returned in the action_data parameter as a short.

FZ_FUIM_ACTN_CUST_TEXT_FOCUS_SET: This action indicates that the item is given text focus or it is being taken away. Text focus is given to the item when the user selects the item and taken away when another item is selected. If the item has focus, then keys strokes in the template will be handled by the item (FZ_FUIM_ACTN_CUST_KEY_DOWN). This should be returned in the action_data parameter as a short.

The action_data parameter is an abstract data type (fz_type_td) used for exchange of data. Data is extracted from or stored into this parameter based on the action that is being handled by the custom function.

The func_data parameter is a storage place for any data that needs to be passed to the custom item function from the fz_fuim_new_custom function. This data can then be cast and used in this function.

The prvt_data parameter is a pointer to a pointer of privately allocated data for the custom item. This is usually allocated in the FZ_FUIM_ACTN_CUST_INIT action and released in the FZ_FUIM_ACTN_CUST_FINIT message. This is useful for data needed during the life of the custom item.

Combination items

There are a number of convenience functions that combine more than one FUIM item. Effectively, they create each of the component items, and align them in a horizontal group, and link them to the same variable. This means that when one of the items is updated the other item is updated as well. For example, a slider and edit field combo item has both a slider and an editable text field. If one were to edit the text field by supplying a new number, the slider would be updated with a new slider position and vice versa. The combination item functions are:

`fz_fuim_new_slid_edit_long` (slider with editable long field).
`fz_fuim_new_slid_edit_short` (slider with editable short field).
`fz_fuim_new_slid_edit_float` (slider with editable float field).
`fz_fuim_new_slid_edit_double` (slider with editable double field).
`fz_fuim_new_slid_edit_pcent_long` (slider with editable long field represented as a percentage).
`fz_fuim_new_slid_edit_pcent_short` (slider with editable short field represented as a percentage).
`fz_fuim_new_slid_edit_pcent_float` (slider with editable float field represented as a percentage).
`fz_fuim_new_slid_edit_pcent_double` (slider with editable double field represented as a percentage).

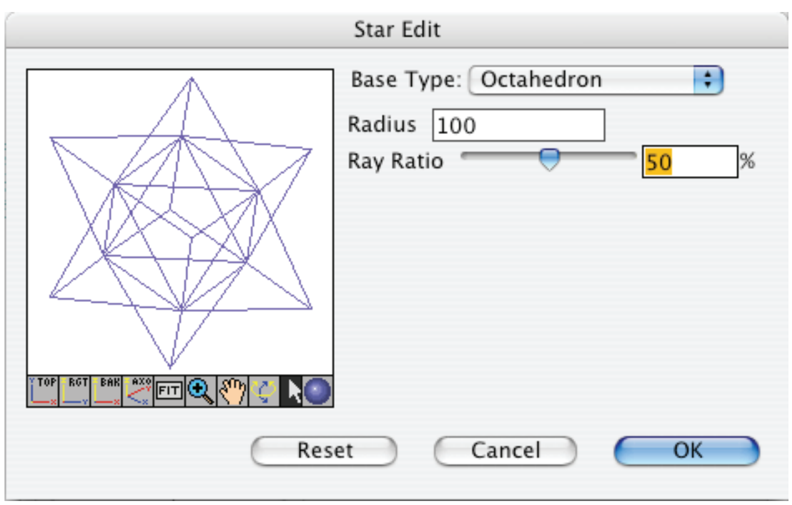
The following combination functions disable the use of their edit fields when they are turned off:

`fz_fuim_new_check_edit` (check box with an editable field – use a range function to associate a variable with edit field).
`fz_fuim_new_radio_text_edit` (radio button with an editable field – use a range function to associate a variable with edit field).
`fz_fuim_new_radio_text_static` (radio button with a text field – use a range function to associate a variable with text field).

2.6.3.2 Advanced template elements

Preview

A template can be set up to show a large square area with an object drawn in it. This is done frequently in object editing dialogs, such as the Star Edit dialog shown below. **form•Z** offers a number of API functions, which can assist a plugin in creating such an object preview. The example code shown in the following section is taken from the star tool plugin, which is available as sample source code in the **form•Z** SDK directory.



The Star Edit dialog with a preview window

Creating the preview item

The object preview template item is created with the following API function calls inside a template setup function:

```
...

enum
{
   STAR_OTYP_STACK_PVIEW_DATA = 0,
   STAR_OTYP_STACK_PVIEW_OPTS,

   STAR_OTYP_STACK_PVIEW_MAX
};

...

fzrt_error_td star_otyp_iface_tmpl(
   long                    windex,
   fz_fuim_tmpl_ptr        fuim_tmpl,
   fzrt_ptr                objt_ptr
   )
{
   fzrt_error_td           err = FZRT_NOERR;
   ...
   fz_objt_ptr             obj;
   star_otyp_parms_td      *star;
   fz_fuim_pview_opts_ptr  pview_opts;
   star_otyp_pview_data_td pview_data, ...
   ...

   obj = (fz_objt_ptr)objt_ptr;
   fz_objt_parm_get_data(windex,obj,(fzrt_ptr*)&star);
   ...
   if((err = fz_fuim_tmpl_init(fuim_tmpl, str, 0, STAR_OTYP_ID, 0)
      ) == FZRT_NOERR)
   {
      pview_data.src_obj = obj;
      pview_data.dst_obj = NULL;
      pview_data.src_windex = windex;
      pview_data.dst_windex = -1;
      pview_data.star_parms = *star;

      fz_fuim_tmpl_set_new_value_func(fuim_tmpl, star_otyp_fuim_newval, NULL);
      ...

      fz_fuim_pview_opts_init(&pview_opts,windex);
      fz_fuim_pview_opts_set_load_func(pview_opts, star_otyp_fuim_load_func);
      fz_fuim_pview_create(fuim_tmpl, g2, FZ_FUIM_NONE, pview_opts);
      fz_fuim_stack_put(fuim_tmpl, STAR_OTYP_STACK_PVIEW_DATA,
                                 sizeof(pview_data), &pview_data);
      ...

   }

   return(err);
}
```

`fz_fuim_pview_opts_init` allocates a preview options data structure and initializes it to default settings. Additional API functions can be called to change the default settings. In the example above, the API function, `fz_fuim_pview_opts_set_load_func` is necessary to load data, such as objects into the preview. It assigns a load function to the preview options. The load function is discussed in more detail below. Finally, the preview is created with a call to `fz_fuim_pview_create`. This creates all the necessary template items for the preview.

Loading objects into the preview window

The preview created with the `fz_fuim_pview_create` call in the template setup function will define a square window in the dialog, in which an object can be shown. For this purpose, **form•Z** creates a new project. The load function is designed to copy the object to be previewed from the original project to the new project. The API function call `fz_fuim_pview_opts_set_load_func` stores a load function implemented by the plugin with the preview options. This plugin defined load function is called right before the dialog is shown on the screen. To copy the object, the API function call `fz_objt_edit_copy_objt_to_windex` can be used. It copies an object from one project to another. The API call `fz_objt_add_objt_to_project` should be made right afterwards to properly add the object to the new project, as the copied object is initially tagged as temporary. The load function for the star tool is shown below.

```
fzrt_error_td    star_otyp_fuim_load_func(
    fz_fuim_tmpl_ptr fuim_tmpl,
    long src_windex,
    long dst_windex)
{
    fzrt_error_td           err=FZRT_NOERR;
    star_otyp_pview_data_td *pview_data;

    fz_fuim_stack_get_ptr(fuim_tmpl, STAR_OTYP_STACK_PVIEW_DATA,
                                   (void**)&pview_data);

    if(pview_data->src_obj != NULL)
    {
        pview_data->dst_windex = dst_windex;

        if((err = fz_objt_edit_copy_objt_to_windex(
            src_windex, pview_data->src_obj,
            dst_windex, TRUE, &pview_data->dst_obj)) == FZRT_NOERR)
        {
            err = fz_objt_add_objt_to_project(dst_windex,pview_data->dst_obj);
        }
    }

    return(err);
}
```

The star tool example also uses a data structure called `star_otyp_pview_data_td`. It is filled with information which is needed by the dialog while it is running. For example, the original object pointer is stored in this data structure, as well as the copied object. This data structure is also stored on the template stack.

The functions described above are sufficient to create the basic object preview. Additional functionality may be needed to enable user interaction with the object shown in the preview. In the star tool example, the user is able to edit the parameters of the star in the dialog. When a new value is entered, the preview is automatically updated. When the dialog is closed, the edited values are saved. The functions needed to enable these features are described in the next sections.

Updating the object preview

When the user changes a field in the dialog, which requires that the object preview is regenerated, the template should install a new value callback function. This needs to be done in the template setup function, as shown above by the star tool template function. The new value function is implemented as follows:

```
fzrt_boolean star_otyp_fuim_newval(fz_fuim_tmpl_ptr fuim_tmpl, void *data_ptr)
{
    star_otyp_pview_data_td *pview_data;
    star_otyp_parms_td      *parm_data;

    fz_fuim_stack_get_ptr(fuim_tmpl, STAR_OTYP_STACK_PVIEW_DATA,
                                (void**)&pview_data);

    /* ASSIGN THE EDITED OBJECT PARAMETERS TO THE PREVIEW OBJECT */
    fz_objt_parm_get_data(pview_data->dst_windex,
                        pview_data->dst_obj,(fzrt_ptr*)&parm_data);
    *parm_data = pview_data->star_parms;

    /* REGENERATE THE PREVIEW OBJECT */
    fz_objt_edit_parm_regen(pview_data->dst_windex,pview_data->dst_obj);

    return (TRUE);
}
```

The edit fields in the dialog were linked to a copy of the star's parameter data structure `pview_data->star_parms`. When **form•Z** calls the new value function, the star parameters are assigned to the parameter block of the object in the preview window:

```
        *parm_data = pview_data->star_parms;
```

The API function `fz_objt_edit_parm_regen` forces the star object to be rebuilt with the new parameters. This also causes the preview window to be redrawn.

Storing the edited object

When the user hits the cancel or OK button and the dialog is closed, the project created for the preview window is automatically deleted, including all the objects contained in it. For the Cancel action, this is appropriate, as no changes need to be kept. When OK is selected, the copy of the original object in the new project needs to be copied back to the original object. This is the inverse of the action taken by the load function, which copies the original object to the preview window's project. The star tools OK function shows how the preview object is copied back to the original object.

```
fzrt_boolean star_otyp_fuim_ok(fz_fuim_tmpl_ptr fuim_tmpl, void *data_ptr)
{
    star_otyp_pview_data_td       *pview_data;

    fz_fuim_stack_get_ptr(fuim_tmpl, STAR_OTYP_STACK_PVIEW_DATA,
            (void**)&pview_data);

    /* EXECUTE THE NEW VAL FUNCTION TO MAKE SURE THAT THE OBJECT IS UPTODATE */
    star_otyp_fuim_newval(fuim_tmpl, data_ptr);

    /* COPY THE PREVIEW OBJECT TO THE ORIGINAL OBJECT */
    fz_objt_edit_copy_objt_data_to_windex(pview_data->dst_windex,
            pview_data->dst_obj, pview_data->src_windex,
            pview_data->src_obj, TRUE);

    return(TRUE);
}
```

Note, that it is not necessary to explicitly delete the preview object that was copied from the original object. The project of the preview window is completely deleted when the dialog is closed, which includes the deletion of all objects in the project.

Clicking and tracking in the preview window

When the user needs to interact with the preview window, two callback function can be implemented, a click and a track function. The click function is set with the API function:

```
void   fz_fuim_pview_opts_set_click_func(
       fz_fuim_pview_opts_ptr     opts,
       fz_fuim_pview_click_func   click_func
       );
```

The click function itself needs to be declared as:

```
short  fz_fuim_pview_click_func(
       long                       windex,
       fz_fuim_pview_opts_ptr     opts,
       fzrt_point                 *click_pnt
       );
```

It is invoked by **form•Z** when the user clicks in the preview window. The screen position where the click occurred is passed in.

The track function is called when the mouse is moved in the preview window for an action other than the standard view editing commands. This allows a plugin to implement is own interactive editing operation, which is linked to the movement of the mouse. The track function is set with the API function:

```
void   fz_fuim_pview_opts_set_track_func(
       fz_fuim_pview_opts_ptr     opts,
       fz_fuim_pview_track_func   track_func
       );
```

The click function itself needs to be declared as:

```
short  fz_fuim_pview_track_func(
       long                              windex,
       fz_fuim_pview_opts_ptr            opts,
       fzrt_point                        *click_pnt,
       fzrt_cursor_ptr                   *curs_ptr
       );
```

As with the click function the screen location of the mouse is passed in. The track function is expected to return the cursor, which **form•Z** needs to draw while the tracking occurs.

Setting additional options in the preview window

The preview window can be configured to exhibit a number of different behaviors. These are set up with bit encoded flags in the API function call:

```
void   fz_fuim_pview_opts_set_flags(
       fz_fuim_pview_opts_ptr            opts,
       long                              flags
       );
```

This call needs to be made prior to the creation of the preview window custom item. The `flags` parameter determines the behavior of the window. The following options are available:

`FZ_FUIM_PVIEW_FLAG_2D_BIT`
When this bit is set, the window is created as a 2d window. The view is always set to a top view and cannot be changed to any other 3d view. This is useful for shown 2d profiles or curve like objects, that do not have a z dimension.

`FZ_FUIM_PVIEW_FLAG_NOPICK_BIT`
When this bit is set, the pick tool below the preview window is disabled. This should be done, when the preview window does not handle any picking by the user.

`FZ_FUIM_PVIEW_FLAG_NORNDRMENU_BIT`
When this bit is set, the preview window is does not offer any additional rendering modes. The only rendering mode available is Wireframe. This should be done, when showing abstract graphics in the window, instead of a solid or surface object.

`FZ_FUIM_PVIEW_FLAG_IRNDRMENU_BIT`
When this bit is set, only interactive rendering modes are offered by the rendering menu below the preview window.

`FZ_FUIM_PVIEW_FLAG_GRID_BIT`
When this bit is set, the project axis and reference plane are draw by **form•Z** in the preview window. By default the axis and the grid are not drawn.

`FZ_FUIM_PVIEW_FLAG_NOSIZE_BIT`
When this bit is set, the preview window is not resized to the maximum size available on the screen. By default, the preview window is sized according to the user's Preview Dialogs Size settings set in the Preferences dialog under the Dialogs section.

`FZ_FUIM_PVIEW_FLAG_NOPAN_BIT`
When this bit is set, the pan view tool below the preview window is disabled. By default, the user can pan the view in the preview window.

`FZ_FUIM_PVIEW_FLAG_NOGHOST_BIT`
When this bit is set, ghosted objects are not hidden in the preview window. By default, ghosted objects are not drawn.

List items

```
long fz_fuim_new_list(
      fz_fuim_tmpl_ptr        fuim_tmpl,
      long                    parent,
      long                    id,
      fz_fuim_list_type_enum  list_type,
      long                    list_flags,
      long                    width,
      long                    num_rows,
      long                    num_cols,
      long                    row_height,
      fz_fuim_list_ptr        *list_ptr
      );
```

A list is an interface element that groups items into a list.

A list is created slightly different from other interface elements in the parameters it takes. The item_func and item_data parameters of other items are not present when creating a list as the list itself is a custom item and handled internally. Callback functions are used anyplace that custom actions are required. See the individual callback functions below for more details. The flags parameter also is not present. A list will behave differently depending on the list type. The list type is set in the fz_fuim_new_list function and is of type fz_fuim_list_type_enum. The different types of lists and how they act are described below.

FZ_FUIM_LIST_TYPE_NONE: This type of list is for viewing data only. The items in the list are not clickable.

FZ_FUIM_LIST_TYPE_ONE: This type of list allows for one and only one item in the list to be picked at a time. It also requires that one item in the list be picked.

FZ_FUIM_LIST_TYPE_MULTI: This type of list allows multiple items in the list to be active at once. It also allows for no item no be picked. This handles mouse click input by toggling the picked status of the clicked item; clicking on a picked item will unpick it and vice versa.

FZ_FUIM_LIST_TYPE_SYSTEM: This type of list also allows for multiple items to be picked at once and allows for no item to be picked. It differs from the FZ_FUIM_LIST_TYPE_MULTI list in how it handles mouse clicks.

Shift-clicking will pick all items between the last picked item and the currently clicked item.

Command-clicking (on MacIntosh) and Control-clicking (on Windows) will toggle the picked status of the clicked item.

Clicking and dragging over a range will pick that range.

The flags parameter changes how the list looks and operates. It uses fz_fuim_list_enum for this, as described below.

FZ_FUIM_LIST_TITLE_BIT: Indicates that a title row is to be drawn at the top of the list. Regardless of the height of the item rows, the title row will always be the same height; namely just tall enough for the text to be drawn in. The default is for no title row to be drawn.

FZ_FUIM_LIST_DRAG_BIT: Indicates that dragging of row items is allowed. This only affects lists of type FZ_FUIM_LIST_TYPE_ONE and FZ_FUIM_LIST_TYPE_MULTI. If this bit is set, the fz_fuim_list_drag_func callback should be defined in order to handle the drag.

FZ_FUIM_LIST_DEFFONT2_BIT: Indicates the list is to use the alternate default font.

FZ_FUIM_LIST_BG_LIST_BIT: Indicates that the background of the list is to be drawn with the list theme. The default is for the background to be drawn with the theme of the background of the tab control if the list is in a tab control pane or with the theme of the background of the dialog otherwise. This bit only affects lists of type FZ_FUIM_LIST_TYPE_NONE. All other list types have their backgrounds drawn with the list theme.

FZ_FUIM_LIST_H_DIVIDE_BIT: Indicates that horizontal lines dividing rows are to be displayed. The default is for there to be no divisions between rows. Unlike vertical divisions, horizontal divisions are for visual separation only and can't be resized by click-dragging.

FZ_FUIM_LIST_AUTOSIZE_BIT: Indicates that the horizontal size of the list and the individual columns will be auto-sized to fit when the list is created. The individual columns will be set just wide enough to hold the longest string in the column at creation time. If a column title exists, that string will be included in the calculation. The width of the list will be the sum of each of the individual columns. The default is for the list to use the manually set values. If the horizontal size of the list has been set to any value greater than 0 (the width parameter in fz_fuim_new_list) the size will not be changed, but the individual columns will be resized to fit the list.

FZ_FUIM_LIST_NO_V_DIVIDE_BIT: Indicates that vertical lines dividing columns are not to be displayed. The default is for there to be divisions between columns. Note that if this bit is set, the user will be unable to be resize the columns by click-dragging.

FZ_FUIM_LIST_TITLE_SIZE_BIT: Indicates that the height of the title row is to be the same height as the other rows. The default is for the title row to be just tall enough for the text to be drawn in.

A list may contain 1 or more rows and columns.
When the list is created, the number of viewable rows and columns is set. If there are more row items than viewable rows, the list will enable it's vertical scroll bar to allow for viewing of all the items. The number of column items is set to the number of columns and can't change after the list is created; in other words, there is no horizontal scrolling.

The width parameter sets the horizontal width of the list in pixels. If the column widths are not specified (with fz_fuim_list_set_colm_width) each of the columns evenly divide the width of the list.
The num_rows parameter is the number of rows to be displayed in the list. If there is a title bar, it is not counted towards the number of rows.
The num_cols parameter is the number of columns to be displayed in the list.
The row_height parameter sets the height of each row in pixels. To set the row heights to be just large enough for the text items, pass -1 for row_height. All rows will be the same height. This doesn't affect the height of the title row (if it exists). The title row is always the same size.
The list_ptr parameter returns a pointer to the list structure. list_ptr should be saved to allow for any future calls on the list.

Additional functions for use with lists

Setting a column width

```
fzrt_error_td fz_fuim_list_set_col_width(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              col_indx,
      long              width
      );
```

fz_fuim_list_set_col_width is used to set the width of a column in pixels. The width is changed by the moving of the divider bar to the right of the column. If the selected width is smaller than the minimum, the width will be set to the minimum. If the selected width will make the column to the right smaller than it's minimum, the width will be set to the maximum it is allowed. If the column has a title, the minimum is the width of the title. Otherwise, the minimum is a set number of pixels. The column farthest to the right can't have it's width changed. (If that column's width needs to be changed, change the width on the column to it's left.) This function has no effect on lists with one column.

Getting a column width

```
fzrt_error_td fz_fuim_list_get_colm_width(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              col_indx,
      long              *width
      );
```

fz_fuim_list_get_colm_width is used to get the width of a column in pixels. col_indx is the index to the column and must be greater than or equal to 0 and less than the number of columns in the list. width is returned as the width of the specified column is pixels.

Setting a column title

```
fzrt_error_td fz_fuim_list_set_colm_title(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              col_indx,
      char              *title_str
      );
```

fz_fuim_list_set_colm_title is used to set the title of a column. col_indx is the index to the column and must be greater than or equal to 0 and less than the number of columns in the list. If title_str is passed as NULL or the empty string, the specified column title will be empty. By default, all column titles are empty.

Getting a column width

```
fzrt_error_td fz_fuim_list_get_colm_title(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              col_indx,
      char              *title_str,
      long              *max_len
      );
```

fz_fuim_list_get_colm_title is used to get the title of a column. col_indx is the index to the column and must be greater than or equal to 0 and less than the number of columns in the list. title_str must be preallocated to hold at least max_len number of characters.

Setting a column flags

```
fzrt_error_td fz_fuim_list_set_colm_flags(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              col_indx,
      long              flags
      );
```

Sets flags of a column. col_indx is the index to the column and must be greater than or equal to 0 and less than the number of columns in the list. If flags is passed as 0, no flags will be set and any previously set flags on the column will be cleared.

The flags parameter changes how the column looks and operates. It uses fz_fuim_list_colm_enum for this, as described below.

FZ_FUIM_LIST_COLM_NO_TITLE_BIT: Indicates that clicking in the title bar of the column isn't handled. Default is for the column of the title bar that was clicked to be inverted and for the click (or double-click) function called (if one is set).

FZ_FUIM_LIST_COLM_NO_DRAG_BIT: Indicates that the divider line between the column and the column to the right can't be dragged around. Note : Since there is no divider to the right of the far right column, this bit has no effect on that column.

Setting a column text justification

```
fzrt_error_td fz_fuim_list_set_colm_just(
      fz_fuim_tmpl_ptr              fuim_tmpl,
      fz_fuim_list_ptr              list_ptr,
```

```
	long				col_indx,
	fz_fuim_list_just_h_enum	just_h,
	fz_fuim_list_just_v_enum	just_v
	);
```

fz_fuim_list_set_colm_just is used to set the text justification of a column. col_indx is the index to the column and must be greater than or equal to 0 and less than the number of columns in the list. Pass FZ_FUIM_LIST_COLM_JUST_H_NO_CHANGE for just_h and FZ_FUIM_LIST_COLM_JUST_V_NO_CHANGE for just_v to not change the horizontal and vertical justification, respectively.

Horizontal justification can be set to any of the below.
FZ_FUIM_LIST_COLM_JUST_H_LEFT: Indicates that text is to be left justified.
FZ_FUIM_LIST_COLM_JUST_H_CENTER: Indicates that text is to be center justified.
FZ_FUIM_LIST_COLM_JUST_H_RIGHT: Indicates that text is to be right justified.
FZ_FUIM_LIST_COLM_JUST_H_NO_CHANGE: Indicates text justification is to remain how it is.

Vertical justification can be set to any of the below.
FZ_FUIM_LIST_COLM_JUST_V_TOP: Indicates that text is to be top justified.
FZ_FUIM_LIST_COLM_JUST_V_CENTER: Indicates that text is to be center justified.
FZ_FUIM_LIST_COLM_JUST_V_BOTTOM: Indicates that text is to be bottom justified.
FZ_FUIM_LIST_COLM_JUST_V_NO_CHANGE: Indicates text justification is to remain how it is.

Getting the number of row items

```
fzrt_error_td fz_fuim_list_get_nitems(
	fz_fuim_tmpl_ptr	fuim_tmpl,
	fz_fuim_list_ptr	list_ptr,
	long			*num_items
	);
```

fz_fuim_list_get_nitems is used to get the number of row items in a list.

Setting the auto-scroll

```
fzrt_error_td fz_fuim_list_set_auto_scroll(
	fz_fuim_tmpl_ptr	fuim_tmpl,
	fz_fuim_list_ptr	list_ptr,
	fzrt_boolean		auto_scroll
	);
```

fz_fuim_list_set_auto_scroll is used to set the auto-scroll status of a list. When auto-scroll is set, at least one picked item in the list is always viewable. If none of the current items in the list are picked, the list will auto-scroll to the first picked item. If there are no picked items in the list, nothing happens.

Getting the auto-scroll

```
fzrt_error_td fz_fuim_list_get_auto_scroll(
	fz_fuim_tmpl_ptr	fuim_tmpl,
	fz_fuim_list_ptr	list_ptr,
	fzrt_boolean			*auto_scroll
	);
```

fz_fuim_list_get_auto_scroll is used to get the auto-scroll status of a list.

Regenerating the list

```
fzrt_error_td fz_fuim_list_regen(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr
      );
```

fz_fuim_list_regen is used to regenerate a list. This is only useful if the fz_fuim_list_get_num_items_func and fz_fuim_list_get_picked_func callback functions have been set. The regen function retrieves the number of items in the list and re-checks each picked status.

Getting the picked status of an item

```
fzrt_error_td fz_fuim_list_get_picked_status(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              col_indx,
      fzrt_boolean            *picked
      );
```

fz_fuim_list_get_picked_status is used to retrieve the picked status of the specified row. This function is different from the get picked callback function. This function will retrieve the picked status of the item from the status stored in the list as opposed to the callback which requires an externally maintained pick status. row_indx is the index to the row item and will always be greater than or equal to 0 and less than the number of items in the list. col_indx is currently ignored. The picked status for the selected row will be passed back in picked.

Invalidating a list

```
fzrt_error_td fz_fuim_list_inval(
      fz_fuim_tmpl_ptr                          fuim_tmpl,
      fz_fuim_list_ptr                          list_ptr
      );
```

fz_fuim_list_inval is used to invalidate a list.

Callback functions for use with lists

Getting a string

```
fzrt_error_td fz_fuim_list_set_get_string_func(
      fz_fuim_tmpl_ptr              fuim_tmpl,
      fz_fuim_list_ptr              list_ptr,
      fz_fuim_list_get_string_func  func
      );
```

fz_fuim_list_set_get_string_func is used to set the "get string" callback function of a list. The "get string" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_get_string_func(
```

```
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              col_indx,
      char              *str,
      long              max_len,
      fzrt_boolean            *did
      );
```

This callback is designed to return the string in a specified row and column of a list. The draw item callback function is called before this. Both the draw item function and the get string function can be called for the same item. If both functions exist for an item, the draw item is handled first followed by the get string. In this case, the text will be drawn on top of the item. row_indx is the index to the row item and will always be greater than or equal to 0 and less than the number of items in the list. col_indx is the index to the column and will always be greater than or equal to 0 and less than the number of columns in the list. max_len is the maximum length of str and str must not exceed max_len. Strings in the title row are not handled by this callback but are instead set by calling fz_fuim_list_set_colm_title. This callback only needs to return the string for the specified item; it does not need to draw anything.

Drawing an item

```
fzrt_error_td fz_fuim_list_set_draw_item_func(
      fz_fuim_tmpl_ptr              fuim_tmpl,
      fz_fuim_list_ptr              list_ptr,
      fz_fuim_list_draw_item_func   func
      );
```

fz_fuim_list_set_draw_item_func is used to set the "draw item" callback function of a list.
The "draw item" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_draw_item_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              col_indx,
      fzrt_rect         *rect,
      fzrt_boolean            *did
      );
```

This callback is designed to draw the item in a specified row and column of a list. This is called before the get string callback function. Both the draw item function and the get string function can be called for the same item. If both functions exist for an item, the draw item is handled first followed by the get string. In this case, the text will be drawn on top of the item. rect is the rectangle, in screen coordinates, of where the item should be drawn. It is the responsibility of the draw function to only draw within the specified rectangle. If the item drawn is smaller than the rectangle, the rectangle where the item was drawn should be passed back through rect. This returned rect is used to invert the drawn item when the row is picked. row_indx is the index to the row item and will always be greater than or equal to 0 and less than the number of items in the list. col_indx is the index to the column and will always be greater than or equal to 0 and less than the number of columns in the list. Return whether an item was drawn in did.

Handling a single mouse click

```
fzrt_error_td fz_fuim_list_set_click_func(
```

```
      fz_fuim_tmpl_ptr                fuim_tmpl,
      fz_fuim_list_ptr                list_ptr,
      fz_fuim_list_click_func         func
      );
```

fz_fuim_list_set_click_func is used to set the "single-click" callback function of a list. The "single-click" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_click_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              col_indx,
      fzrt_boolean            picked
      );
```

This callback is designed to allow handling of single mouse clicks in the list. row_indx is the index to the row item where the click took place and will always be greater than or equal to -1 and less than the number of items in the list. If row_indx is -1, the click occurred in the title row. col_indx is the index to the column where the click took place and will always be greater than or equal to 0 and less than the number of columns in the list. In lists of type FZ_FUIM_LIST_TYPE_ONE, the picked parameter is always TRUE. In this list type, if a picked list is being maintained (ie. the picked status callback is defined), all other items in the list should be marked not-picked except the one that is being handled. In lists of type FZ_FUIM_LIST_TYPE_MULTI, the picked parameter will be TRUE to indicate the row was toggled on, and FALSE to indicate the row was toggled off. In lists of type FZ_FUIM_LIST_TYPE_SYSTEM, this callback function will be called whenever a row is picked or unpicked. For example, in shift-clicking a range, all rows in the range will be called with picked set to TRUE, and any rows that were previously picked but not in the new range will be called with picked set to FALSE. Note: Don't do any work in this callback that is of a time consuming nature since it is called on each click. This should mainly be used to update the state of other fuim items that are dependent on the item that is selected in the list.

Handling a double mouse click

```
fzrt_error_td fz_fuim_list_set_click_dbl_func(
      fz_fuim_tmpl_ptr                fuim_tmpl,
      fz_fuim_list_ptr                list_ptr,
      fz_fuim_list_click_dbl_func     func
      );
```

fz_fuim_list_set_click_dbl_func is used to set the "double-click" callback function of a list. The "double-click" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_click_dbl_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              col_indx
      );
```

This callback is designed to allow handling of double mouse clicks in the list. row_indx is the index to the row item and will always be greater than or equal to -1 and less than the number of items in the list. If row_indx is -1, the double-click occurred in the title row. col_indx is the index to the column and will always be greater than or equal to 0 and less than the number of columns in the list.

Handling a mouse drag

```
fzrt_error_td fz_fuim_list_set_drag_func(
      fz_fuim_tmpl_ptr              fuim_tmpl,
      fz_fuim_list_ptr              list_ptr,
      fz_fuim_list_drag_func        func
      );
```

fz_fuim_list_set_drag_func is used to set the "drag" callback function of a list.
The "drag" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_drag_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              row_dist,
      fzrt_boolean            *did
      );
```

This callback is designed to allow handling of mouse click drags in the list. row_indx is the index to the row item where the drag started and will always be greater than or equal to 0 and less than the number of items in the list. row_dist is the number of rows the item was dragged. Negative numbers indicate the item was dragged up. Positive numbers indicate the item was dragged down. This callback will not be called if an item is dragged to itself (ie. row_dist will never be 0). Return whether the move was succesfully handled in did. If did isn't set, the move is assumed to have been handled. This callback is only valid on lists of type FZ_FUIM_LIST_TYPE_ONE and FZ_FUIM_LIST_TYPE_MULTI. Drags in lists of type FZ_FUIM_LIST_TYPE_SYSTEM are handled with fz_fuim_list_click.

Getting the picked status of an item

```
fzrt_error_td fz_fuim_list_set_get_picked_func(
      fz_fuim_tmpl_ptr              fuim_tmpl,
      fz_fuim_list_ptr              list_ptr,
      fz_fuim_list_get_picked_func  func
      );
```

fz_fuim_list_set_get_picked_func is used to set the "get picked status" callback function of a list.
The "get picked status" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_get_picked_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              row_indx,
      long              col_indx,
      fzrt_boolean      *picked
      );
```

This callback is designed to return whether a selected item is picked in the list. This callback must be set in order for fz_fuim_list_regen to work. A structure should be maintained that has the picked status of each row. When this callback is called, the picked status for the selected row should be passed back in picked. row_indx is the index to the row item and will always be greater than or equal to 0 and less than the number of items in the list. Note : Currently, the get picked callback will always be called with 0 for col_indx.

Getting the number of items

```
fzrt_error_td fz_fuim_list_set_get_num_items_func(
      fz_fuim_tmpl_ptr                    fuim_tmpl,
      fz_fuim_list_ptr                    list_ptr,
      fz_fuim_list_get_num_items_func     func
      );
```

fz_fuim_list_set_get_num_items_func is used to set the "get number of items" callback function of a list.
The "get number of items" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_get_num_items_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_fuim_list_ptr  list_ptr,
      long              *num_items
      );
```

This callback is designed to return the number of items in the list. This callback must be set in order for fz_fuim_list_regen to work.

Custom initialization

```
fzrt_error_td fz_fuim_list_set_init_func(
      fz_fuim_tmpl_ptr        fuim_tmpl,
      fz_fuim_list_ptr        list_ptr,
      fz_fuim_list_init_func  func
      );
```

fz_fuim_list_set_init_func is used to set the "initialization" callback function of a list.
The "initialization" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_init_func(
      fz_fuim_tmpl_ptr        fuim_tmpl,
      fz_fuim_list_ptr  list_ptr
      );
```

This callback is designed to be called during the initialization of the list. It allows for any user defined data to be initialized as well as setting up the list at startup.

Custom finalization

```
fzrt_error_td fz_fuim_list_set_finit_func(
      fz_fuim_tmpl_ptr        fuim_tmpl,
      fz_fuim_list_ptr        list_ptr,
      fz_fuim_list_finit_func func
      );
```

fz_fuim_list_set_finit_func is used to set the "finalization" callback function of a list.
The "finalization" callback function itself needs to be declared as :

```
fzrt_error_td fz_fuim_list_finit_func(
      fz_fuim_tmpl_ptr  fuim_tmpl,
```

```
        fz_fuim_list_ptr  list_ptr
        );
```

This callback is designed to be called during the finalization of the list. It allows for any user defined data to be finalized.

Creating a list item

The following is an example of creating a list. It includes callback functions that would be used in a typical list. This list is a single pick list. It has 5 columns and 7 visible rows as well as a title row. There is a set number of items in the list, in this case 32. All the columns are resizable by click-dragging except the first two columns. This was done to force the second column to always be the same width. Dragging rows and auto-scrolling are enabled. All of the columns have their text centered both vertically and horizontally. When any of the column titles are clicked, except the second column, the list is sorted by the entries. This is something the list doesn't handle on it's own. It is in a function that is called when a title item is clicked. When an item is double-clicked, the system beeps.

```
#define NUM_COLS    5
#define NUM_ITEMS   32

typedef struct list_node_td
{       char                        name[NUM_COLS][256];
        fzrt_boolean        picked;
}  list_node_td;

typedef struct list_td
{       fz_fuim_list_ptr    list_ptr;
        list_node_td        nodes[NUM_ITEMS];
}  list_td;

list_td                         _list_data;

/*********************************************************************
**********************************************************************/
static void list_sort(fz_fuim_tmpl_ptr fuim_tmpl, fz_fuim_list_ptr list_ptr,
long col_indx)
{
        long                i, j, cmp;
        char                str1[256], str2[256];
        fzrt_key_state_enum test_shift;
        fzrt_boolean        inverse;
        list_node_td        temp_node;

        fzrt_util_funcs.fzrt_evnt_get_key_state(FZRT_VIRT_SHIFTKEY, &test_shift);
        if (test_shift == FZRT_KEY_PRESSED)     inverse = TRUE;
        else                                    inverse = FALSE;
        for (i = 0; i < NUM_ITEMS-1; i++)
        {       for (j = i+1; j < NUM_ITEMS; j++)
                {       strncpy(str1, _list_data.nodes[i].name[col_indx], 256);
                        strncpy(str2, _list_data.nodes[j].name[col_indx], 256);
                        cmp = strcmp(str1, str2);
                        if ((cmp < 0 && inverse == TRUE) ||
                                (cmp > 0 && inverse == FALSE))
                        {       temp_node = _list_data.nodes[i];
                                _list_data.nodes[i] = _list_data.nodes[j];
                                _list_data.nodes[j] = temp_node;
                        }
                }
        }
}
```

```
/*********************************************************************
**********************************************************************/
static void init_list(void)
{
	long			i, j, temp;

	for (i = 0; i < NUM_ITEMS; i++)
	{	for (j = 0; j < NUM_COLS; j++)
		{	if (j != 1)
			{	temp = rand() % NUM_ITEMS;
				sprintf(_list_data.nodes[i].name[j], "%02ld", temp);
			}
			else
			{	strcpy(_list_data.nodes[i].name[j], "");
			}
		}
		_list_data.nodes[i].picked = FALSE;
	}
	_list_data.nodes[NUM_ITEMS-1].picked = TRUE;

}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_get_picked(fz_fuim_tmpl_ptr fuim_tmpl,
	 fz_fuim_list_ptr list_ptr, long row_indx, long col_indx,
	 fzrt_boolean *picked)
{
	if (_list_data.nodes[row_indx].picked == TRUE) (*picked) = TRUE;
	else						(*picked) = FALSE;
	return(FZRT_NOERR);
}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_get_num_items(fz_fuim_tmpl_ptr fuim_tmpl,
		fz_fuim_list_ptr list_ptr, long *num_items)
{
	(*num_items) = NUM_ITEMS;
	return(FZRT_NOERR);
}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_click(fz_fuim_tmpl_ptr fuim_tmpl, fz_fuim_list_ptr
		list_ptr, long row_indx, long col_indx, fzrt_boolean picked)
{
	long			i;

	if (row_indx == -1)
	{	list_sort(fuim_tmpl, list_ptr, col_indx);
	}
	else
	{	for (i = 0; i < NUM_ITEMS; i++)
		{	_list_data.nodes[i].picked = FALSE;
		}
		_list_data.nodes[row_indx].picked = TRUE;
	}
	return(FZRT_NOERR);
}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_get_str(fz_fuim_tmpl_ptr fuim_tmpl,
```

```
               fz_fuim_list_ptr list_ptr, long row_indx, long col_indx,
               char *str, long max_len, fzrt_boolean *did)
{
       if (col_indx != 1)
       {      strncpy(str, _list_data.nodes[row_indx].name[col_indx], max_len);
              (*did) = TRUE;
       }
       else (*did) = FALSE;
       return(FZRT_NOERR);
}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_click_dbl(fz_fuim_tmpl_ptr fuim_tmpl,
               fz_fuim_list_ptr list_ptr, long row_indx, long col_indx)
{
       if (row_indx > -1)fzrt_sys_beep(0);
       return(FZRT_NOERR);
}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_draw_item(fz_fuim_tmpl_ptr fuim_tmpl,
               fz_fuim_list_ptr list_ptr, long row_indx, long col_indx,
               fzrt_rect *rect, fzrt_boolean *did)
{
       fzrt_rgb_color_td   black = {0, 0, 0};

       if (col_indx == 1)
       {      rect->right = rect->left + 50;
              rect->bottom = rect->top + 50;
              fzrt_rgb_fore_color(&black);
              fzrt_move_to(rect->left, rect->top);
              fzrt_line_to(rect->right, rect->bottom);
              fzrt_move_to(rect->left, rect->bottom);
              fzrt_line_to(rect->right, rect->top);
              (*did) = TRUE;
       }
       else (*did) = FALSE;
       return(FZRT_NOERR);
}

/*********************************************************************
**********************************************************************/
static fzrt_error_td list_drag(fz_fuim_tmpl_ptr fuim_tmpl,
               fz_fuim_list_ptr list_ptr, long from_row,
               long row_dist, fzrt_boolean *did)
{
       list_node_td        temp_node;
       long                i, to_row;

       to_row = from_row + row_dist;
       if (from_row < to_row)
       {      temp_node = _list_data.nodes[from_row];
              for (i = from_row; i < to_row; i++)
              {      _list_data.nodes[i] = _list_data.nodes[i + 1];
              }
              _list_data.nodes[to_row] = temp_node;
       }
       else
       {      temp_node = _list_data.nodes[from_row];
              for (i = from_row; i > to_row; i--)
              {      _list_data.nodes[i] = _list_data.nodes[i - 1];
              }
              _list_data.nodes[to_row] = temp_node;
```

```
	}
	(*did) = TRUE;
	return(FZRT_NOERR);
}

/*********************************************************************
*********************************************************************/
static fzrt_error_td fuim_rndr_net_server_setup(fz_fuim_tmpl_ptr fuim_tmpl)
{
	fzrt_error_td			err = FZRT_NOERR;
	long				i, flags;
	fz_fuim_list_just_h_enum	just_h = FZ_FUIM_LIST_COLM_JUST_H_CENTER;
	fz_fuim_list_just_v_enum	just_v = FZ_FUIM_LIST_COLM_JUST_V_CENTER;

	if (fz_fuim_tmpl_init(fuim_tmpl, "List Dialog", FZ_FUIM_FLAG_NONE,
			RNDR_NET_SERVER_SETUP_FUIM_ID, 0) == FZRT_NOERR)
	{	init_list();
		flags = 0;
		FZ_SETBIT(flags, FZ_FUIM_LIST_TITLE_BIT);
		FZ_SETBIT(flags, FZ_FUIM_LIST_DRAG_BIT);
		FZ_SETBIT(flags, FZ_FUIM_LIST_H_DIVIDE_BIT);
		i = fz_fuim_new_list(fuim_tmpl, FZ_FUIM_ROOT, FZ_FUIM_NONE,
			FZ_FUIM_LIST_TYPE_ONE,
			flags, 500, 7, NUM_COLS, 50, &_list_data.list_ptr);
		fz_fuim_list_set_colm_title(fuim_tmpl, _list_data.list_ptr, 0,
				"Column 1");
		fz_fuim_list_set_colm_title(fuim_tmpl, _list_data.list_ptr, 2,
				"Column 3");
		fz_fuim_list_set_colm_title(fuim_tmpl, _list_data.list_ptr, 3,
				"Column 4");
		fz_fuim_list_set_colm_title(fuim_tmpl, _list_data.list_ptr, 4,
				"Column 5");
		flags = 0;
		FZ_SETBIT(flags, FZ_FUIM_LIST_COLM_NO_DRAG_BIT);
		fz_fuim_list_set_colm_flags(fuim_tmpl, _list_data.list_ptr, 0,
				flags);
		FZ_SETBIT(flags, FZ_FUIM_LIST_COLM_NO_TITLE_BIT);
		fz_fuim_list_set_colm_flags(fuim_tmpl, _list_data.list_ptr, 1,
				flags);
		fz_fuim_list_set_col_width(fuim_tmpl, _list_data.list_ptr, 1, 50);
		for (i = 0; i < NUM_COLS; i++)
		{	fz_fuim_list_set_colm_just(fuim_tmpl, _list_data.list_ptr,
				i, &just_h, &just_v);
		}
		fz_fuim_list_set_get_string_func(fuim_tmpl, _list_data.list_ptr,
				list_get_str);
		fz_fuim_list_set_draw_item_func(fuim_tmpl, _list_data.list_ptr,
				list_draw_item);
		fz_fuim_list_set_click_func(fuim_tmpl, _list_data.list_ptr,
				list_click);
		fz_fuim_list_set_click_dbl_func(fuim_tmpl, _list_data.list_ptr,
				list_click_dbl);
		fz_fuim_list_set_drag_func(fuim_tmpl, _list_data.list_ptr,
				list_drag);
		fz_fuim_list_set_get_picked_func(fuim_tmpl, _list_data.list_ptr,
				list_get_picked);
		fz_fuim_list_set_get_num_items_func(fuim_tmpl,
				_list_data.list_ptr, list_get_num_items);
	}
	return(err);
}
```

2.6.4 Interface for time consuming tasks

Plugins that could potentially take a while to execute should implement the **wait cursor**, **key cancel**, and where possible a **progress bar**. These interface elements provide feedback to the user and allow the user to interrupt long or unintended tasks.

Wait cursor

The cursor should be changed to the wait cursor to indicate to the user when a task is being performed. On the Macintosh, this cursor is a spinning circle with alternating black and white quadrants. On Windows, the wait cursor is an animated hourglass. The function `fz_fuim_curs_wait` should be called to update the wait cursor during the processing of a task. This function takes a single parameter with the following three values:

> `FZ_FUIM_CURS_WAIT_START`: This value is used once at the start of the task. The cursor is changed to the wait cursor.
> `FZ_FUIM_CURS_WAIT_TURN`: This value is used during the processing of the task. The animated cursor is updated (turned). The function should be called with this value inside loops and other places where the flow of the extension will consume its time. Performance is not an issue with this value because the cursor is only updated every 1/4 second regardless of how frequently the function is called. Note that, if it is not called frequently enough, the cursor will appear jumpy.
> `FZ_FUIM_CURS_WAIT_END`: This value is used once at the end of a time consuming task. The cursor is changed back to the state it was in prior to the start of the task.

It is important to have exactly one start and end call so that the cursor display stays balanced. This allows for nesting of the wait cursor in a case where one time consuming extension invokes another time consuming extension.

Cancel

The user should be able to cancel any time consuming task. An extension can check to see if the user has pressed the key shortcut for cancel by calling the function `fz_fuim_key_cancel`. This function returns TRUE if the cancel key shortcut has been pressed and FALSE if it has not. Note that the user can program a variety of key combinations for the cancel key shortcut using the **Shortcuts** dialog, however, extensions do not need to make any adjustments for this as it is all handled by the one function.

Progress bar

A progress bar gives the user feedback on the progress of a task. A progress bar is a small window that displays graphic and optionally descriptive textual feedback on how far a task has progressed. A progress bar is divided into stages so that task sub-portions can be identified to the user. The progress bar is updated by the extension through the use of a variable in the extension that tracks the task's progress. Loop counters are often good indication of progress through a task as shown in the example at the end of this section.

form•Z offers normal and extended styles of the progress bar as shown below. The difference between them is that the extended has much larger areas for text. Both styles have two text areas referred to as the **info** and **detail** strings. The info string is usually used to display a title for the detail string. The detail string usually is used to give some information about the task progress. In the normal progress bar the info and detail strings are short and appear next to each other. This is the style of progress bar used throughout most of **form•Z**. In the extended style, the text fields are on top of each other and they are much larger. The space for the detail string supports multiple lines. This style of progress bar is used in **form•Z** during animation generation.

There are a number of functions in the FUIM for working with progress bars. They all start with `fz_fuim_prog_`. The basic required functions for implementing a progress bar are described here and in the example at the end of the section. The remainder of the function descriptions can be found in HTML API reference (chapter 5).

The function `fz_fuim_prog_init` is called once at the start of the task to initialize the progress bar.

```
fzrt_error_td fz_fuim_prog_init(
      long                             stages,
      fz_fuim_prog_kind_enum           kind,
      fzrt_boolean                     use_clock
      );
```

The `stages` parameter indicates how many stages the progress bar will have. There are two types of progress bars indicated by the `kind` parameter. The normal progress bar has a graphic progress indicator, a short information field and a short detail field. The expanded progress bar has a graphic progress indicator and a single line information field and a multi-line detail field. If the `use_clock` parameter is `TRUE`, then the graphic progress indicator is redrawn every 1/4 second (if there has been any progress since the last redraw). If this value is `FALSE`, then the progress bar is updated (redrawn) each time that the progress bar indicator changes. To avoid performance degradation from the progress bar, it is recommended that `TRUE` be used for this parameter.

The function `fz_fuim_prog_stage_init` is called to indicate the start of a task stage.

```
fzrt_error_td  fz_fuim_prog_stage_init(
      char          *name,
      long          min,
      long          max
      );
```

The `name` parameter is the title of the stage that is shown in the tittle bar of the progress bar window. The `min` and `max` parameters define the range of the progress indicator during the stage. That it is, the progress indicator will move from min to max during the stage with min indicating 0% completion and max indicating 100% completion.

The function `fz_fuim_prog_stage_set_current` is used during the processing of a stage to update the progress bar to indicate the current progress.

```
fzrt_error_td  fz_fuim_prog_stage_set_current(
      long          current
      );
```

The `current` parameter is the value of the progress indicator and must have a value between the min and max parameters used in the most recent `fz_fuim_prog_stage_init` function call.

The function `fz_fuim_prog_stage_set_strings` is used during the processing of a stage to update the info or detail strings in the progress window.

```
fzrt_error_td fz_fuim_prog_stage_set_strings(
      char          *prog_info,
      char          *prog_detail
      );
```

The `prog_info` parameter is the string for the info field of the progress window. If this string is not provided, the string is not changed. The `prog_detail` parameter is the string for the detail field of the progress window. If this string is not provided, the string is not changed.

The function `fz_fuim_prog_stage_finit` should be called to indicate the completion of a stage.

```
fzrt_error_td  fz_fuim_prog_stage_finit(
       void
       );
```

The function `fz_fuim_prog_finit` should be called to indicate the completion of the entire task. This function removes the progress bar window from the screen.

```
fzrt_error_td  fz_fuim_prog_finit(
       void
       );
```

The following example shows the implementation of the wait cursor, key cancel and multi-stage progress bar in two loops of a plugin.

```
fzrt_boolean  canceled= FALSE;
long          i;
char          str[256];
double        done;

/* start wait cursor */
fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_START);

/* initalize progress bar with 2 stages */
fz_fuim_prog_init(2, FZ_FUIM_PROG_KIND_NORMAL, TRUE);

/* start the first stage */
fz_fuim_prog_stage_init(“Loop 1”, 1, 100);
fz_fuim_prog_stage_set_strings(“Completed:”, “0 %”);
for(i=1;i<=100;i++)
{
       /* do task first stage processing here */

       /* check for key cancel short cut */
       if(fz_fuim_key_cancel())
       {      canceled = TRUE;
              break;
       }
       /* update the progress bar indicator */
       fz_fuim_prog_stage_set_current(i);

       /* update the progress bar detail text */
       done = i;
       sprintf(str, “%lf”, done);
       strcat(str, “ %”);
       fz_fuim_prog_stage_set_strings(NULL, str);

       /* update the wait cursor */
       fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_TURN);
}
/* complete the first stage */
fz_fuim_prog_stage_finit();
```

```
if(!canceled)
{
	/* start the second stage */
	fz_fuim_prog_stage_init(“Loop 2”, 1, 2000);
	fz_fuim_prog_stage_set_strings(“Completed:”, “0 %”);
	for(i=1;i<=2000;i++)
	{

		/* do second stage processing here */

		/* check for key cancel short cut */
		if(fz_fuim_key_cancel())
		{	canceled = TRUE;
			break;
		}
		/* update the progress bar indicator */
		fz_fuim_prog_stage_set_current(i);

		/* update the progress bar detail text */
		done = (i/2000.0) * 100.0;
		sprintf(str, “%lf”, done);
		strcat(str, “ %”);
		fz_fuim_prog_stage_set_strings(NULL, str);

		/* update the wait cursor */
		fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_TURN);

		/* complete the second stage */
		fz_fuim_prog_stage_finit();
	}
}

/* complete the progress bar */
fz_fuim_prog_finit();

/* complete the wait cursor */
fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_END);
```

2.7 Notification

The **form•Z** notification manager is used to notify plugins when certain events occur. The events include changes in **form•Z** project data like objects, lights and layers. Plugins can receive these notifications by implementing functions in the `fz_notf_cbak_fset`. This function set provides a variety of functions that **form•Z** calls when the respective event occurs. Care should be used when implementing these functions because notification functions are called throughout **form•Z** and a poor implementation can lead to performance issues or crashes. Likewise only necessary functions should be implemented, since even empty "shell" functions will cause some performance degradation.

Notification call back function set

Notifications are in the call back function set `fz_notf_cbak_fset`. A plugin which desires to receive notifications from **form•Z** must inform **form•Z** that it is a plugin that will implement notification call backs. This is done by the following function call, adding the notification function set to the plugin, and informing **form•Z** of another plugin function (`my_fill_notf_fset`) which **form•Z** will call to fill the notification function set with the plugin's specific call back functions. This call should be made after the plugin is registered (`fzpl_plugin_register`).

```
err = fzpl_glue->fzpl_plugin_add_fset(
      my_plugin_runtime_id,
      FZ_NOTF_CBAK_FSET_TYPE,
      FZ_NOTF_CBAK_FSET_VERSION,
      FZ_NOTF_CBAK_FSET_NAME,
      FZPL_TYPE_STRING(fz_notf_cbak_fset),
      sizeof (fz_notf_cbak_fset),
      my_fill_notf_fset,
      FALSE);
```

The `my_fill_notf_fset` is provided by the plugin developer and **form•Z** will call it to find out which notification call backs are implemented by the plugin. The `my_fill_notf_fset` receives a function set parameter `fset` to which it will assign the notification call back functions. All the notification call back functions are optional. When a notification occurs, **form•Z** will only call the functions provided by a plugin developer. The following shows the body of a `my_fill_notf_fset` function.

```
fzrt_error_td my_fill_notf_fset (
            const fzpl_fset_def_ptr fset_def,
            fzpl_fset_td * const fset )
{
      fzrt_error_td              err = FZRT_NOERR;
      fz_notf_cbak_fset          *notf_funcs;

      err = fzpl_glue->fzpl_fset_def_check ( fset_def,
            FZ_NOTF_CBAK_FSET_VERSION,
            FZPL_TYPE_STRING(fz_notf_cbak_fset),
            sizeof (fz_notf_cbak_fset),
            FZPL_VERSION_OP_NEWER );

      if ( err == FZRT_NOERR )
      {
            notf_funcs = (fz_notf_cbak_fset *)fset;

            notf_funcs->fz_notf_cbak_objt    = my_notf_objt;
            ...
      }
```

```
        return err;
}
```

Here the plugin developer supplies the `my_notf_objt` call back function, among other call backs, which must be implemented by the plugin. Each notification call back function is described next.

The system function (optional)

```
fzrt_error_td fz_notf_cbak_syst (
        fz_notf_syst_enum          syst_notf
        );
```

This function is called by **form•Z** when one of the actions specified by `fz_notf_syst_enum` occurs. This function is provided so that extensions can be notified when one of the actions occurs and the extension can make any extension specific adjustments in reaction to the action.

```
fzrt_error_td my_notf_syst(
        fz_notf_syst_enum          syst_notf
        )
{
        fzrt_error_td       err = FZRT_NOERR;


        /** Handle notification here **/

        return(err);
}
```

The project function (optional)

```
fzrt_error_td fz_notf_cbak_proj (
        long                       windex,
        fz_notf_proj_enum          proj_notf
        );
```

This function is called by **form•Z** when one of the actions specified by `fz_notf_proj_enum` occurs in the specified project. This function will be called for each project in which the action occurs. This function is provided so that extensions can be notified when one of the actions occurs and the extension can make any extension specific adjustments in reaction to the action.

```
fzrt_error_td my_notf_proj(
        long                       windex,
        fz_notf_proj_enum          proj_notf
        )
{
        fzrt_error_td       err = FZRT_NOERR;


        /** Handle project notification here **/

        return(err);
}
```

The window function (optional)

```
fzrt_error_td fz_notf_cbak_wind (
        long                       windex,
        fz_notf_wind_enum          wind_notf,
        fz_notf_proj_enum          proj_notf
```

```
        );
```

This function is called by **form•Z** when one of the actions specified by fz_notf_wind_enum occurs in the specified project. This function will be called for each window in which the action occurs. This function is provided so that extensions can be notified when one of the actions occurs and the extension can make any extension specific adjustments in reaction to the action.

This function is also called for each window in a project when a project notification happens (ie `fz_notf_cbak_proj` is called) . In this situation `wind_notf == FZ_NOTF_WIND_PROJ` and `proj_notf` is the value of the project level notification.

```
fzrt_error_td my_notf_wind(
        long                        windex,
        fz_notf_wind_enum           wind_notf,
        fz_notf_proj_enum           proj_notf
        )
{
        fzrt_error_td       err = FZRT_NOERR;


        /** Handle window notification here **/

        return(err);
}
```

The system units function (optional)

```
fzrt_error_td fz_notf_cbak_syst_units (
        fz_unit_type_enum           pref_units,
        fz_unit_scale_enum          pref_scale
        );
```

This function is called when the current unit type (English/Metric) or unit scale (large/medium/small/miniture) changes. This happens when the user changes the settings in the Working Units dialog , the function `fz_proj_units_set_parm_data` is called to change the settings or when the active window is changed to a project with different Working units settings. When this notification is received, all system level (global) dimensional values should be converted to a reasonable setting for the current settings.

It is recommended that function `fz_fuim_unit_convert` be used to get proper dimensional values (units and data scale) from default values for the specified `pref_units` and `pref_scale`. The `fz_fuim_unit_convert` function sets a double value to the current `pref_units` and `pref_scale` given an English and metric default unit values for a specified scale.

The following example establishes a default English value of 12.0 inches and a metric default value of 25 cm for the medium scale.

```
double my_syst_distance;

fzrt_error_td my_notf_syst_units (
        fz_unit_type_enum           pref_units,
        fz_unit_scale_enum          pref_scale
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        err = fz_fuim_unit_convert(12.0, 25.0, FZ_UNIT_SCAL_MEDIUM,
```

```
                        pref_units, pref_scale, &my_syst_distance);

        return(err);
}
```

The project units function (optional)

```
fzrt_error_td fz_notf_cbak_proj_units (
        long                        windex,
        fz_unit_type_enum           pref_units,
        fz_unit_scale_enum          pref_scale
        );
```

This function is called when the unit type (English/Metric) or unit scale (large/medium/small/miniature) for a project is changed. This happens when the user changes the settings in the Working Units dialog or the function `fz_proj_units_set_parm_data` is called to change the settings. When this notification is received, all project level dimensional values should be converted to a reasonable setting for the current settings.

It is recommended that function `fz_fuim_unit_convert` be used to get proper dimensional values (units and data scale) from default values for the specified `pref_units` and `pref_scale`. The `fz_fuim_unit_convert` function sets a double value to the current `pref_units` and `pref_scale` given English and metric default unit values for a specified scale.

The following example establishes a default English value of 12.0 inches and a metric default value of 25 cm for the medium scale.

```
double* my_distance;

fzrt_error_td my_notf_proj_units (
        long                        windex,
        fz_unit_type_enum           pref_units,
        fz_unit_scale_enum          pref_scale
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        err = fz_fuim_unit_convert(12.0, 25.0, FZ_UNIT_SCAL_MEDIUM,
                            pref_units, pref_scale, &my_distance[windex]);

        return(err);
}
```

The window units function (optional)

```
fzrt_error_td fz_notf_cbak_wind_units (
        long                        windex,
        fz_unit_type_enum           pref_units,
        fz_unit_scale_enum          pref_scale
        );
```

This function is called for each project window when the unit type (English/Metric) or unit scale (large/medium/small/miniature) for a project changes. This happens when the user changes the settings in the Working Units dialog or the function `fz_proj_units_set_parm_data` is called to change the settings. When this notification is received, all project level dimensional values should be converted to a reasonable setting for the current settings.

It is recommended that function fz_fuim_unit_convert be used to get proper dimensional values (units and data scale) from default values for the specified pref_units and pref_scale. The fz_fuim_unit_convert function sets a double value to the current pref_units and pref_scale given English and metric default unit values for a specified scale.

The following example establishes a default English value of 12.0 inches and a metric default value of 25 cm for the medium scale.

```
double* my_distance;

fzrt_error_td my_notf_wind_units (
       long                     windex,
       fz_unit_type_enum        pref_units,
       fz_unit_scale_enum       pref_scale
       )
{
       fzrt_error_td       err = FZRT_NOERR;

       err = fz_fuim_unit_convert(12.0, 25.0, FZ_UNIT_SCAL_MEDIUM,
                           pref_units, pref_scale, &my_distance[windex]);

       return(err);
}
```

The object function (optional)

```
fzrt_error_td fz_notf_cbak_objt (
       long                            windex,
       fz_notf_objt_enum               objt_notf,
       fz_objt_ptr                     objt
       );
```

This function is called to notify that an object has changed. The objt_notf parameter indicates what change occurred.

```
fzrt_error_td my_notf_objt (
       long                            windex,
       fz_notf_objt_enum               objt_notf,
       fz_objt_ptr                     objt
       )
{
       fzrt_error_td       err = FZRT_NOERR;


       /** Handle object notification here **/

       return(err);
}
```

The light function (optional)

```
fzrt_error_td fz_notf_cbak_lite (
       long                            windex,
       fz_notf_lite_enum               lite_notf,
       fz_lite_ptr                     lite
       );
```

This function is called to notify that a light has changed. The lite_notf parameter indicates what change occurred.

```
fzrt_error_td my_notf_lite(
	long				windex,
	fz_notf_lite_enum		lite_notf,
	fz_lite_ptr			lite
	)
{
	fzrt_error_td	err = FZRT_NOERR;


	/** Handle light notification here **/

	return(err);
}
```

The layer function (optional)

```
fzrt_error_td fz_notf_cbak_layr (
	long				windex,
	fz_notf_layr_enum		layr_notf,
	fz_layr_ptr			layr
	);
```

This function is called to notify that an layer has changed. The `layr_notf` parameter indicates what change occurred.

```
fzrt_error_td my_notf_layr(
	long				windex,
	fz_notf_layr_enum		layr_notf,
	fz_layr_ptr			layr
	)
{
	fzrt_error_td	err = FZRT_NOERR;


	/** Handle layer notification here **/

	return(err);
}
```

The view function (optional)

```
fzrt_error_td fz_notf_cbak_view (
	long				windex,
	fz_notf_view_enum		view_notf,
	fz_view_ptr			view
	);
```

This function is called to notify that a view has changed. The `view_notf` parameter indicates what change occurred.

```
fzrt_error_td my_notf_view(
	long				windex,
	fz_notf_view_enum		view_notf,
	fz_view_ptr			view
	)
{
	fzrt_error_td	err = FZRT_NOERR;


	/** Handle view notification here **/

	return(err);
```

```
}
```

The preference defaults function (optional)

```
fzrt_error_td fz_notf_cbak_pref_default (
	fz_unit_type_enum		pref_units,
	fz_unit_scale_enum		pref_scale
	);
```

The default function is called by **form•Z** called once at startup (after plugin registration) and when user resets the preferences to defaults in the preferences dialog. This function is provided so that plugins can establish default values for private data. All private data should be set to its default values and dimensional values should be set to the specified `pref_units` and `pref_scale`. It is recommended that function `fz_fuim_unit_convert` should be used to get proper dimensional values (units and data scale) from default values for the specified `pref_units` and `pref_scale`. The `fz_fuim_unit_convert` function sets a double value to the current `pref_units` and `pref_scale` given an English and metric default unit values for a specified scale.

```
double my_distance;

fzrt_error_td my_notf_pref_default (
	fz_unit_type_enum		pref_units,
	fz_unit_scale_enum		pref_scale
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	my_data->value1 = 0;
	my_data->value2 = 10;

	...

	err = fz_fuim_unit_convert(12.0, 25.0,
			FZ_UNIT_SCAL_MEDIUM, pref_units, pref_scale,
			&my_distance);

	...

	return(err);
}
```

The preference model type function (optional)

```
fzrt_error_td fz_notf_cbak_pref_model_type (
	fz_objt_model_type_enum	model_type
	);
```

The preference model type function is called by **form•Z** when the model type preference is changed. This function notifies the plugin to change its internal preference to facetted (`FZ_OBJT_MODEL_TYPE_FACT`) or smooth modeling (`FZ_OBJT_MODEL_TYPE_SMOD`) as indicated by the `model_type` parameter. This function is useful for tool plugins which support both facetted and smooth modeling.

```
fz_objt_model_type_enum my_model_type;

fzrt_error_td my_notf_pref_model_type (
	fz_objt_model_type_enum		model_type
	)
```

```
{
	fzrt_error_td		err = FZRT_NOERR;

	my_model_type = model_type;

	return(err);
}
```

2.8 Plugin Types (classes)

There are 10 types of plugins: **attributes**, **file translators**, **object types**, **renderers**, **commands**, **palettes**, RenderZone **shaders**, **tools**, **utilities** and **surface styles**. Plugins are organized into types based on the functionality they provide and how they implement it. Some types of plugins are flexible and can add functionality to various areas of **form•Z**. Other types of plugins add very specific functionality to a certain area of the program. The command and utility plugin types are examples of more flexible plugins while the RenderZone shader plugin type is very specific.

There is also a distinction between system and project level plugins. System plugins are not dependent on the active window or project, hence the call back functions for system plugins do not receive the active project window `windex` as a parameter. Project level plugins work on the active project window, and therefore do receive `windex` as a parameter.

2.8.1 Attributes

form•Z is equipped with a set of standard attributes, such as surface styles, layers, shadow casting or visibility. Attributes may be assigned to objects and/or faces. It is possible to create custom attributes in a plugin by registering a function set with a plugin class. Multiple attribute function sets may be installed with a single plugin. This allows a plugin to offer a suite of attribute types, which logically belong together in a single package. Attributes can be installed either as a stand alone plugin, or with a plugin of another type. For example, a plugin developer may create a new tool command plugin and add it to the Attributes tool palette. The tool may be used to select objects and assign a special custom attribute to them. The attribute would then be registered with the command plugin, not with an attribute plugin. If an attribute is registered alone, **form•Z** offers automatic mechanisms to add attributes to and remove them from objects and faces. This is described in more detail below.

Attribute plugin type and registration

An attribute plugin is identified with the plugin type `FZ_ATTR_EXTS_TYPE` and version of `FZ_ATTR_EXTS_VERSION`, and must implement the `fz_attr_cbak_fset` call back function set. The following code example shows the registration of an attribute plugin and an attribute callback function set. This is done from the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3.

```
fzrt_error_td my_attr_register_plugin ()
{
	fzrt_error_td		err = FZRT_NOERR;

	/* REGISTER THE ATTRIBUTE PLUGIN */
	err = fzpl_glue->fzpl_plugin_register(
		MY_ATTR_PLUGIN_UUID,
		MY_ATTR_PLUGIN_NAME,
		MY_ATTR_PLUGIN_VERSION,
		MY_ATTR_PLUGIN_VENDOR,
		MY_ATTR_PLUGIN_URL,
		FZ_ATTR_EXTS_TYPE,
		FZ_ATTR_EXTS_VERSION,
		NULL /*error string function*/, 0, NULL,
		&my_plugin_runtime_id);

	if ( err == FZRT_NOERR )
	{
		/* REGISTER THE ATTRIBUTE FUNCTION SET */
		err = fzpl_glue->fzpl_plugin_add_fset(
				my_plugin_runtime_id,
				FZ_ATTR_CBAK_FSET_TYPE,
				FZ_ATTR_CBAK_FSET_VERSION,
				FZ_ATTR_CBAK_FSET_NAME,
				FZPL_TYPE_STRING(fz_attr_cbak_fset),
				sizeof (fz_attr_cbak_fset),
				my_fill_attr_cbak_fset,
				FALSE);
	}

	return(err);
}
```

Attribute call back function set

Attribute plugins are implemented by the call back function set `fz_attr_cbak_fset`.

The plugin developer must pass a fill function to `fzpl_plugin_add_fset` which assigns the pointers of the functions which define the plugin's functionality to an instance of the `fz_attr_cbak_fset` callback function set. An example of the fill function for a sample attribute is shown below.

```
fzrt_error_td my_fill_attr_cbak_fset (
      const fzpl_fset_def_ptr fset_def,
      fzpl_fset_td * const fset )
{
   fzrt_error_td       err = FZRT_NOERR;
   fz_attr_cbak_fset   *attr_fset;

   err = fzpl_glue->fzpl_fset_def_check ( fset_def,
                  FZ_ATTR_CBAK_FSET_VERSION,
                  FZPL_TYPE_STRING(fz_attr_cbak_fset),
                  sizeof (fz_attr_cbak_fset),
                  FZPL_VERSION_OP_NEWER );

   if ( err == FZRT_NOERR )
   {
      attr_fset = (fz_attr_cbak_fset *)fset;

      /* ALL LEVELS CALLBACKS, REQUIRED */
      attr_fset->fz_attr_cbak_uuid         = my_attr_uuid;
      attr_fset->fz_attr_cbak_name         = my_attr_name;
      attr_fset->fz_attr_cbak_info         = my_attr_info;
      attr_fset->fz_attr_cbak_io           = my_attr_iost;

      /* ALL LEVELS CALLBACKS, OPTIONAL */
      attr_fset->fz_attr_cbak_deflt           = my_attr_deflt;
      attr_fset->fz_attr_cbak_finit        = my_attr_finit;
      attr_fset->fz_attr_cbak_copy         = my_attr_copy;
      attr_fset->fz_attr_cbak_tform           = my_attr_tform;
      attr_fset->fz_attr_cbak_are_equal    = my_attr_are_equal;
      attr_fset->fz_attr_cbak_iface_tmpl   = my_attr_iface_tmpl;

      /* OBJECT LEVEL CALLBACKS, OPTIONAL */
      attr_fset->fz_attr_cbak_objt_merge   = my_attr_objt_merge;

      /* FIELD INFORMATION FUNCTIONS, OPTIONAL */
      attr_fset->fz_attr_cbak_get_field_count = my_attr_get_field_count;
      attr_fset->fz_attr_cbak_get_field_info  = my_attr_get_field_info;
      attr_fset->fz_attr_cbak_get_field_data  = my_attr_get_field_data;
   }

   return err;
}
```

Of all the functions in the set, only four are required. They are:

```
fz_attr_cbak_name
fz_attr_cbak_uuid
fz_attr_cbak_info
fz_attr_cbak_io
```

All others are optional. Note, that there is no callback function to explicitly create an attribute. Depending on the use of the attribute, different mechanisms may be designed to assign attributes to entities. For example, a plugin developer may create a set of modeling tools, which create objects and also create the attribute when the modeling tool is executed. A modeling tool may also be entirely dedicated to assign and edit an attribute. This is currently the case in **form•Z** with the Texture Map Control tool. Attributes may be assigned automatically by **form•Z**, depending on the flags defined by the `fz_attr_cbak_info` callback function, which is described in more detail below.

The name function (required)

```
fzrt_error_td fz_attr_cbak_name (
       char          *name,
       long          max_len
       );
```

This function is called by **form•Z** to get the name of the attribute. This name shows up in the **form•Z** interface, whenever the content of the attribute is displayed. The name function must assign a string to the function's name argument. The length of the string assigned cannot exceed `max_len` characters. It is recommended that the attribute name be stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example below, this step is omitted for the purpose of simplicity.

```
fzrt_error_td my_attr_name (
       char          *name,
       long          max_len
       )
{
       strncpy(name,"Some attribute",max_len);
       return(FZRT_NOERR);
}
```

The uuid function (required)

```
fzrt_error_td        fz_attr_cbak_uuid (
       fzrt_UUID_td               uuid
       );
```

This function is called by **form•Z** to get the uuid of the attribute. This unique id is used by **form•Z** to distinguish the attribute from other attributes. For example, when a **form•Z** project file is written to disk, any attributes of this type are saved as well and identified with this uuid. When the project file is later opened again, **form•Z** will connect the loaded attribute data with the installed attribute plugin. If the plugin that created the attribute is not installed, the attribute is automatically deleted. The uuid function needs to assign this unique identifier string to the function's uuid argument. An example is shown below.

```
#define MY_ATTR_UUID       \
"\x2d\xa8\x6d\xe1\xdb\xd3\x40\xc4\xa7\xb3\xd9\xe3\xd2\x73\x69\x75"

fzrt_error_td        my_attr_uuid (
       fzrt_UUID_td uuid
       )
{
       fzrt_UUID_copy(MY_ATTR_UUID, uuid);
       return(FZRT_NOERR);
}
```

The info function (required)

```
fzrt_error_td fz_attr_cbak_info (
       long          *size,
```

```
    long          *level_flags,
    long          *flags
    );
```

The info function is called by **form•Z** to retrieve basic information about the attribute. Three separate pieces of information must be supplied: size, levels and flags.
form•Z manages the storage of each instance of an attribute. In order to do so, **form•Z** needs to know, what the data size (in # of bytes) of the attribute content is. The size argument must be set to the number of bytes that the attribute data storage requires. In most cases, a plugin developer will create a structure with fields which describe the attribute content. The size returned to **form•Z** via this callback can be acquired with a `sizeof(structure_type)` call.

Attributes may exist on a number of different levels. They are object and face. The level_flags argument must be set to the levels by which the attribute is used. An attribute may exist on more than one level. The bit defines in `fz_attr_level_enum` should be used to set the proper bits in the `level_flags` argument with the `FZ_SETBIT` macro.

The flags argument tells **form•Z** basic information about the attribute, for example, whether the attribute is assigned automatically to all new objects or not. The flags argument should be set with the bit encoded flags defined in the enum `fz_attr_flags_enum`. The following attribute behavior can be achieved by setting the respective bit flag:

`FZ_ATTR_FLAGS_ADD_OBJ_ALWAYS`

When this flag is set, the attribute is always added to a new object. When this is done, the `fz_attr_cbak_deflt` callback function is invoked to set the default parameters. This option should be chosen with care. In general it is better to not automatically assign an attribute to new objects. The plugin code that deals with using an attribute should assume default values when an attribute is not present with an object. This will save storage space, as attributes tend to occupy significant amounts of memory. Only if the attribute is a simple marker, or a reference to other data and it matters, when the object is created, should this flag be used. For example, if Surface Styles were implemented as a plugin attribute, the developer would create the actual surface style table and a simple object level attribute, which is a reference to a surface style in the palette. When a new object is created, the tag of the currently active surface style would be assigned as the attribute to the object.

`FZ_ATTR_FLAGS_SHOW_QUERY`

When this flags is set, an entry in the Additional Attributes list with the attributes name is shown in the respective Query Attributes dialog. The entry is only shown, if the queried entity has an attribute of the given type assigned to it. When double clicking on the list entry, the attribute's dialog, which is set up by the `fz_attr_cbak_iface_tmpl` callback function, is invoked.

`FZ_ATTR_FLAGS_SHOW_QUERY_ALWAYS`

When this flags is set, an entry in the Additional Attributes list with the attributes name is always shown in the respective Query Attributes dialog, regardless of whether the attribute is assigned to the entity or not. When double clicking on the list entry, the attribute's dialog, which is set up by the `fz_attr_cbak_iface_tmpl` callback function, is invoked. If the attribute does not exist with the entity, **form•Z** will automatically create the attribute. This will invoke the `fz_attr_cbak_deflt` callback function.

`FZ_ATTR_FLAGS_TEMPORARY`

When this flag is set, the attribute is only maintained during the runtime session of **form•Z**. The attribute content is not read to or written from file. When combined with the FZ_ATTR_FLAGS_SHOW_QUERY and FZ_ATTR_FLAGS_SHOW_QUERY_ALWAYS flags not set, it allows a plugin to create an attribute which is invisible to the user.

An example of an info function for a sample attribute is shown below.

```
fzrt_error_td my_attr_info (
        long            *size,
        long            *level_flags,
        long            *flags
        )
{
        *size = sizeof(my_attr_td);

        *level_flags = 0;
        FZ_SETBIT(*level_flags,FZ_ATTR_LEVEL_OBJT);
        FZ_SETBIT(*level_flags,FZ_ATTR_LEVEL_FACE);

        *flags = 0;
        FZ_SETBIT(*flags, FZ_ATTR_FLAGS_SHOW_QUERY);

        return(FZRT_NOERR);
}
```

The io stream function (required)

```
fzrt_error_td           fz_attr_cbak_io (
        long                            windex,
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum             dir,
        fzpl_vers_td * const            version,
        unsigned long                   size,
        void                            *data
        );
```

form•Z calls this function to write an attribute to and read it from file. It is expected from the plugin to keep track of version changes of the attribute. For example, in its first release, an attribute may consist of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, the plugin developer adds a fifth long integer value to increase the size to 20 bytes. When writing this new attribute, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the attribute, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value. Likewise, it is possible, that an older version of the plugin will be asked to read a newer version of the attribute. This may be the case when backsaving a **form•Z** project file to an older version and then reading that file with an older version of **form•Z** that contains the older version of the attribute plugin. In this case, the plugin may choose to read the data, i.e. the first 16 bytes of version 0. For safety, it may also choose to skip any attribute data that is written with a version that is newer than the one it is currently set to. An example of the attribute io steam function is shown below. Note, that **form•Z** will allocate the basic storage for the attribute when reading. That is, the data pointer passed in is allocated to the size defined by the attribute through the fz_attr_cbak_info callback function.

```
fzrt_error_td           my_attr_iost (
```

```
        long                        windex,
        fz_iost_ptr                 iost,
        fz_iost_dir_td_enum         dir,
        fzpl_vers_td * const        version,
        unsigned long               size,
        void                        *data
        )
{
        my_attr_td          *my_attr;
        fzrt_error_td rv = FZRT_NOERR;

        my_attr = (my_attr_td*)data;

        if ( dir == FZ_IOST_WRITE ) *version = 1;

        if((rv = fz_iost_long(iost,&my_attr->value1,4)) == FZRT_NOERR )
        {
                if ( *version == 1 )
                {
                        rv = fz_iost_one_long(iost,&my_attr->value5);
                }
                else
                {       if ( dir == FZ_IOST_READ ) my_attr->value5 = 0;
                }
        }

        return(rv);
}
```

The defaults function (optional)

```
fzrt_error_td fz_attr_cbak_deflt (
        long                windex,
        void                *data
        );
```

When an attribute is created automatically by **form•Z**, the default values of the attribute's content need to be assigned. Such an automatic creation may occur, when the `FZ_ATTR_FLAGS_ADD_OBJ_ALWAYS` or `FZ_ATTR_FLAGS_SHOW_QUERY_ALWAYS` flags are set in the `fz_attr_cbak_info` callback function. The defaults function is called by **form•Z** anytime this occurs, and is expected to fill in default values. An example of the defaults function for a sample attribute is shown below:

```
fzrt_error_td my_attr_deflt (
        long                windex,
        void                *data
        )
{

        my_attr_td          *my_attr;
        fzrt_error_td rv = FZRT_NOERR;

        my_attr = (my_attr_td*)data;

        my_attr->n_array = 0;
        my_attr->array = NULL;
        my_attr->value1 = 10;
        my_attr->value2 = 20;
        ...
```

```
        return(rv);
}
```

The finit function (optional)

```
fzrt_error_td fz_attr_cbak_finit (
        long                windex,
        void                *data
        );
```

When an attribute is deleted, it may be necessary to also free memory allocated inside the attribute, or it may be necessary to perform operations which must be executed when the attribute ceases to exist. The finit function is expected to perform these tasks. Note, that the basic attribute storage is managed by **form•Z**. It is not necessary for the plugin to deallocate the number of bytes which are defined through the `fz_attr_cbak_info` callback function. Assuming that the attribute has an array that was dynamically allocated some time during the attribute's existence, the finit function for a sample attribute may be written as follows:

```
fzrt_error_td my_attr_finit (
        long                windex,
        void                *data
        )
{
        my_attr_td          *my_attr;

        my_attr = (my_attr_td*)data;

        if (my_attr->array != NULL )
        {       fz_mem_zone_free(my_attr_zone_ptr,(fzrt_ptr*)&my_attr->array);
        }

        return(FZRT_NOERR);
}
```

Note, that the above function uses the API call `fz_mem_zone_free` to decallocate the dynamic memory. The first argument to this function is a memory zone. If a plugin uses dynamic memory which persists past the execution of a single function, it should create its own memory zone on a per project basis. Memory zones are discussed in more detail in section 1.4.4.

The copy function (optional)

```
fzrt_error_td fz_attr_cbak_copy (
        long                src_windex,
        void                *src_data,
        long                dst_windex,
        void                *dst_data
        );
```

When an entity which contains an attribute is copied, the attribute content must be copied as well. **form•Z** allocates the basic storage, as indicated by the `fz_attr_cbak_info` function with the destination entity. The copy function is then expected to copy the content of an attribute from the source to a destination storage. If this function is not implemented by the plugin, **form•Z** automatically copies each byte of the attribute content from the source to the destination. If this function is defined by the plugin, **form•Z** still allocates the destination storage to the # of bytes as before, but the copy callback function is now expected to copy the data from the source to the destination. For example, this is necessary, when the attribute contains dynamically allocated arrays. In this case, the copy function is responsible to allocate the array in the destination and copy the array from the source. The copy function of a sample attribute with a dynamic array is shown below.

```
fzrt_error_td my_attr_copy (
       long              src_windex,
       void              *src_data,
       long              dst_windex,
       void              *dst_data
       )
{
       my_attr_td        *src_my_attr,*dst_my_attr;
       fzrt_error_td err = FZRT_NOERR;

       src_my_attr = (my_attr_td*) src_data;
       dst_my_attr = (my_attr_td*) dst_data;

       if (src_my_attr->n_array > 0 )
       {
              if((err = fz_mem_zone_alloc(
                            my_attr_zone_ptr,
                            sizeof(long) * src_my_attr->n_array,
                            FALSE,
                            (fzrt_ptr*)dst_my_attr->array)
                 ) == FZRT_NOERR )
              {

                     fzrt_block_move(src_my_attr->array,
                                   dst_my_attr->array,
                                   sizeof(long) * src_my_attr->n_array);
              }
       }
       else
       {
              dst_my_attr->array = NULL;
       }
       dst_my_attr->n_array = src_my_attr->n_array;

       /* COPY REMAINING FIELDS */
       dst_my_attr->value1 = src_my_attr->value1;
       dst_my_attr->value2 = src_my_attr->value2;
       /* … ETC */

       return(err);

}
```

The compare function (optional)

```
fzrt_error_td fz_attr_cbak_are_equal (
       void              *data1,
       void              *data2,
       fzrt_boolean      *are_equal
       );
```

For certain operations in **form•Z**, it is necessary to determine, whether two attributes are equal in their content. The compare callback function is expected to perform this task. If this function is not implemented by the plugin, **form•Z** automatically determines whether the two attributes are equal, by comparing each byte in the attributes. The number of bytes compared is the same as the # of bytes returned by the `fz_attr_cbak_info` function. The compare function should be implemented when a straight byte comparison will not yield the proper result. This is the case, for example, when the attribute contains dynamically allocated arrays. The compare function of a sample attribute with a dynamic array is shown below.

```
fzrt_error_td my_attr_are_equal (
        void                *data1,
        void                *data2,
        fzrt_boolean        *are_equal
        )
{
        my_attr_td          *my_attr1,*my_attr2;
        fzrt_error_td       err = FZRT_NOERR;
        long                i;

        *are_equal = TRUE;

        my_attr1 = (my_attr_td*) data1;
        my_attr2 = (my_attr_td*) data2;

        /* COMPARE ARRAY SIZE */
        if (my_attr1->n_array == my_attr2->n_array )
        {
                /* COMPARE ARRAY CONTENT */
                for(i = 0; i < my_attr1->n_array; i++)
                {
                        if (my_attr1->array[i] != my_attr2->array[i] )
                        {
                                *are_equal = FALSE;
                                break;
                        }
                }

                if (*are_equal == TRUE)
                {
                        /* COMPARE REMAINING FIELDS */
                        if (my_attr1->value1 != my_attr2-> value1 ||
                            my_attr1->value2 != my_attr2-> value2 )
                        {
                                *are_equal = FALSE;
                        }
                }
        }
        else
        {
                *are_equal = FALSE;
        }

        return(err);

}
```

The dialog function (optional)

```
long            fz_attr_cbak_iface_tmpl(
        long                windex,
        fz_fuim_tmpl_ptr    fuim_tmpl,
        fzrt_ptr            fuim_data
        );
```

The dialog template function is expected to create the dialog items, with which the content of the attribute is displayed. If this function is implemented, double clicking on the attribute's entry in the Additional Attributes list in one of the Query Attributes dialog, invokes the attribute dialog. It is quite possible to also create "invisible" attributes, which are never accessible to a user. They may be designed to hold temporary data or may serve a purpose other than presenting information to a user. For these types of attributes, the dialog template function should not be implemented. A third use of an attribute may be, where the plugin creates one or more modeling tools, which allow the user to assign and edit a complex attribute. This is the case in **form•Z** with the Texture Map Control or Decals tools. For these attributes, the dialog template function may or may not be implemented. One of the modeling tools may be assigned to provide a dialog interface to deal with displaying and editing the content of the attribute. The dialog template function of a sample attribute is shown below.

```
fzrt_error_td       my_attr_dlog_tmpl (
       long                windex,
       fz_fuim_tmpl_ptr    fuim_tmpl,
       fzrt_ptr            fuim_data
       )
{
       my_attr_td          *my_attr;
       short               g1;
       fzrt_error_td       rv = FZRT_NOERR;

       my_attr = (my_attr_td*) fuim_data;

       rv = fz_fuim_tmpl_init(fuim_tmpl,"Some Attribute Options",
              FZ_FUIM_FLAG_NONE, MY_ATTR_TMPL_ID, 10);
       if (rv == FZRT_NOERR)
       {
              fz_fuim_new_text_static_edit(fuim_tmpl,
                     FZ_FUIM_ROOT,FZ_FUIM_NONE,"Value 1",
                     FZ_FUIM_NONE,FZ_FUIM_FLAG_NONE, NULL,NULL,&g1);

              fz_fuim_item_range_long(fuim_tmpl, g1, &my_attr->value1,
                     0,0,FZ_FUIM_FORMAT_INT_DEFAULT,FZ_FUIM_RANGE_NONE);

              fz_fuim_new_text_static_edit(fuim_tmpl,
                     FZ_FUIM_ROOT,FZ_FUIM_NONE,"Value 2",
                     FZ_FUIM_NONE,FZ_FUIM_FLAG_NONE, NULL,
                     NULL,&g1);

              fz_fuim_item_range_long(fuim_tmpl, g1, &my_attr->value2,
                     0,0,FZ_FUIM_FORMAT_INT_DEFAULT,FZ_FUIM_RANGE_NONE);
       }

       return(rv);
}
```

The transform function (optional)

```
fzrt_error_td fz_attr_cbak_tform (
       long                        windex,
       void                        *data,
       fz_mat4x4_td                *mat
       );
```

An attribute may contain data fields, which define dimensions, such as a length or a radius, or which define locations, such as an origin. These data fields may be subject to a transformation, that is performed on the entity which contains the attribute. The transform callback function, if defined, is invoked by **form•Z** when a transformation on the owning entity is performed. The callback is expected to adjust linear dimensions according to the scale contained in the transformation matrix and apply the matrix to 3d locations. The scale portion of a matrix can be extracted with the math API call `fz_math_4x4_mat_to_trl_scl_rot`. An example of a transform function of a sample attribute is shown below. It scales a radius field by the average scale of the matrix and transforms a xyz origin point by the matrix.

```
fzrt_error_td my_attr_tform (
        long                        windex,
        void                        *data,
        fz_mat4x4_td                *mat
        )
{
        fz_xyz_td           scl;
        my_attr_td          *my_attr;

        my_attr = (my_attr_td*) data;

        fz_math_4x4_mat_to_trl_scl_rot(mat,NULL,&scl,NULL);
        my_attr->radius *= (scl.x + scl.y + scl.z) / 3.0;
        fz_math_4x4_multiply_mat_xyz(mat,&my_attr->origin);

        return(FZRT_NOERR);
}
```

The object merge function (optional)

```
fzrt_error_td       fz_attr_cbak_objt_merge (
        long                              src_windex,
        fz_objt_ptr                       src_obj,
        long                              src_indx,
        long                              dst_windex,
        fz_objt_ptr                       dst_obj,
        long                              dst_indx,
        fz_objt_topo_level_enum           topo_level
        );
```

This function is called whenever part (or all) of an object was appended to or merged with another object. This is the case, for example, after a boolean or join volumes modeling operation. This callback function gives the plugin the opportunity to make adjustments to the attributes of appended faces. It may be called for any of the topological levels which contain attribute data. When this function is called, `dst_obj` already contains the merged data, including any copied attributes. The `dst_indx` parameter contains the face index of the entity which was merged, or it contains -1 if the topological level is the object level. `src_obj` is the original object which was merged and `src_indx` is the index of the original face, or -1 if the topological level is the object level. The `topo_level` parameter indicates for which topological level this function is called. Inside of the merge function, it may be necessary to retrieve the attribute of the source or destination or both. This can be done with the API calls `fz_objt_attr_get_objt_cust`, `fz_objt_attr_get_face_cust`, etc. After modifications were made to the attributes, the data can be assigned back to the respective entity with `fz_objt_attr_set_objt_cust` or `fz_objt_attr_set_face_cust`. An example of a merge function is shown below.

```
fzrt_error_td       my_attr_objt_merge (
        long                              src_windex,
        fz_objt_ptr                       src_obj,
        long                              src_indx,
        long                              dst_windex,
        fz_objt_ptr                       dst_obj,
        long                              dst_indx,
        fz_objt_topo_level_enum           topo_level
```

```
	)
{
	my_attr_td		src_my_attr,dst_my_attr;
	fzrt_error_td		rv = FZRT_NOERR;

	if (topo_level == FZ_OBJT_TOPO_LEVEL_FACE)
	{
		/* GET THE SOURCE ATTRIBUTE */
		rv = fz_objt_attr_get_face_cust(src_windex,
						src_obj,
						src_indx,
						MY_ATTR_UUID,
						NULL,
						&src_my_attr);

		if ( rv == FZRT_NOERR )
		{
			/* GET THE DESTINATION ATTRIBUTE */
			rv = fz_objt_attr_get_face_cust(dst_windex,
						dst_obj,
						dst_indx,
						MY_ATTR_UUID,
						NULL,
						&dst_my_attr);
		}

		/* NOW MAKE ADJUSTMENTS TO THE DESTINATION ATTRIBUTE */
		/* BASED ON INFORMATION FROM THE SOURCE AND DESTINATION */
		/* ATTRIBUTES */

		...

		if ( rv == FZRT_NOERR )
		{
			/* SAVE THE DESTINATION ATTRIBUTE BACK */
			rv = fz_objt_attr_set_face_cust(dst_windex,
						dst_obj,
						dst_indx,
						MY_ATTR_UUID,
						&dst_my_attr);
		}

	}

	return(rv);

}
```

The get field count function (optional)

```
fzrt_error_td  fz_attr_cbak_get_field_count (
	long		*num_fields
	);
```

This function is called by formZ to determine how many fields of this attribute need to be shown to the user. This is, for example, done in the Attributes Manager dialog. Note, that this does not necessarily have to be all the fields of the attribute, just the ones that need to be shown to the user in the context of attribute and information management.

```
fzrt_error_td  my_attr_get_field_count (
```

```
	long		*num_fields
	)
{
	*num_fields = 4;
	return(FZRT_NOERR);
}
```

The get field info function (optional)

```
fzrt_error_td  fz_attr_cbak_get_field_info (
	long			field_indx,
	char			*name,
	long			max_name_len,
	fz_attr_field_type_enum	*field_type,
	fz_fuim_format_float_enum	*unit_fmt_flt,
	fz_fuim_format_int_enum	*unit_fmt_int,
	fz_type_td		*def_value,
	fz_type_td		*min_value,
	fz_type_td		*max_value,
	fzrt_UUID_td		vlist_uuid,
	long			*flags
	);
```

This function is called by formZ to retrieve information about a particular attribute field. The information consists of field name, data type, default value, field format, minimum and maximum range. This information is retrieved, for example, in the Attributes Manager dialog. The index passed in needs to be interpreted by the plugin to address the proper field in the attribute. The index ranges between 0 and the value returned by `fz_attr_cbak_get_field_count`. Note, that it is not necessary to expose all fields of an attribute to a user in this fashion, just the ones that needs to be seen in the context of attribute and information management. The name returned by this function is also used to determine the proper reference in an expression in the Information Management dialog.

```
fzrt_error_td  my_attr_get_field_info (
	long			field_indx,
	char			*name,
	long			max_name_len,
	fz_attr_field_type_enum	*field_type,
	fz_fuim_format_float_enum	*unit_fmt_flt,
	fz_fuim_format_int_enum	*unit_fmt_int,
	fz_type_td		*def_value,
	fz_type_td		*min_value,
	fz_type_td		*max_value,
	fzrt_UUID_td		vlist_uuid,
	long			*flags
	)
{
	fz_string_td  str,str2;
	double        dval;

	*flags = 0;
	switch(field_indx)
	{
		case 0 :
			fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR1, str);
			*field_type = FZ_ATTR_FIELD_TYPE_STNG;
			fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR2, 
str2);
			fz_type_set_string(str2,def_value);
		break;

		case 1 :
```

```
                    fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR3, str);
                    *field_type = FZ_ATTR_FIELD_TYPE_STNG;
                    fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR4,
str2);
                    fz_type_set_string(str2,def_value);
              break;

              case 2 :
                    fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR5, str);
                    *field_type = FZ_ATTR_FIELD_TYPE_STNG;
                    fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR6,
str2);
                    fz_type_set_string(str2,def_value);
              break;

              case 3 :
                    fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS, MY_STR7, str);
                    *field_type = FZ_ATTR_FIELD_TYPE_CRCY;

                    FZ_SETBIT(*flags,FZ_FUIM_RANGE_MIN_BIT);
                    dval = 1495.0;      fz_type_set_double(&dval,def_value);
                    dval = 0.0;         fz_type_set_double(&dval,min_value);
                    dval = 0.0;         fz_type_set_double(&dval,max_value);
              break;
       }

       strncpy(name,str,max_name_len);

       return(FZRT_NOERR);
}
```

The get field data function (optional)

```
fzrt_error_td  fz_attr_cbak_get_field_data (
       long          windex,
       fz_objt_ptr   obj,
       void          *data,
       long          field_indx,
       fz_type_td    *value
       );
```

This function is called by formZ to retrieve information about a particular attribute field. The information consists of field name, data type, default value, field format, minimum and maximum range. This information is retrieved, for example, in the Attributes Manager dialog. The index passed in needs to be interpreted by the plugin to address the proper field in the attribute. The index ranges between 0 and the value returned by `fz_attr_cbak_get_field_count`. Note, that it is not necessary to expose all fields of an attribute to a user in this fashion, just the ones that needs to be seen in the context of attribute and information management. The name returned by this function is also used to determine the proper reference in an expression in the Information Management dialog.

```
fzrt_error_td  my_attr_get_field_data (
       long          windex,
       fz_objt_ptr   obj,
       void          *data,
       long          field_indx,
       fz_type_td    *value
       )
{
       fz_string_td  str;
       double        dval;
```

```
	my_attr_td	*my_attr;


	my_attr = (my_attr_td*) data;

	switch(field_indx)
	{
		case 0 :
			fz_type_set_string(my_attr->str1,value);
		break;

		case 1 :
			fz_type_set_string(my_attr->str2,value);
		break;

		case 2 :
			switch(my_attr->value5)
			{
				case 0 :
					fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS,
						MY_STR8, str);
				break;

				case 1 :
					fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS,
						MY_STR9, str);
				break;

				case 2 :
					fzrt_fzr_get_string(my_rsrc_ref, MY_STRINGS,
						MY_STR10, str);
				break;
			}
			fz_type_set_string(str,value);
		break;

		case 3 :
			switch(my_attr->value5)
			{
				case 0 : dval = my_attr->dval1;
				break;
				case 1 : dval = my_attr->dval2;
				break;
				case 2 : dval = my_attr->dval3;
				break;
			}
			fz_type_set_double(&dval,value);
		break;

	}

	return(FZRT_NOERR);
}
```

2.8.2 Command Plugins

A command in **form•Z** is an action that is invoked from a menu item, icon in the command palette or a key shortcut. Command plugins are extensions that complement the **form•Z** commands and behave consistent with the **form•Z** commands. Command plugins are available in ***system*** and ***project*** levels. A system command is global in nature and does not require a project window index. These are typically utility actions for which it is desirable to have access to the utility in the **form•Z** interface. A project command requires a project or window index and are expected to operate on project information for provided project. Project commands are unavailable when there is no open project window.

Commands are described as **states** and **actions**. A state reflects a setting that has a specific set of selectable values (states) and a single current setting (or active state). For example, the **Show Grid** item in the **Windows** menu is a **form•Z** command that reflects the state of the grid display (on or off). When this item is selected, the state is changed and the check mark in the menu is updated to reflect the current state.

An action command is a command that performs a task when it is selected. The task is linear in nature in that **form•Z** waits for the task to be completed before anything else can be done. An action command is very flexible as virtually any **form•Z** API function can be called during the execution of the task.

There is no explicit distinction between actions and states in the **form•Z** call back functions. For a command to function properly as a state, it should implement the active function described below. This tells **form•Z** that the command in its active state and that the check mark should be drawn in the menu or the icon drawn active in the command palette.

The Samples directory in the **form•Z** SDK folder contains a folder named Commands that contains an example of a command plugin named my_view_command. This example creates a project command plugin with separate commands for selecting each of the standard view types. This sample can be very valuable as both starting points for development as well as examples of how the functions work.

Command plugin type and registration

Command plugins are registered with the plugin type identifier `FZ_CMND_EXTS_TYPE` and version of `FZ_CMND_EXTS_VERSION`. System command plugins must implement the function set `fz_cmnd_cbak_syst_fset` and project command plugins must implement the function set `fz_cmnd_cbak_proj_fset`.

The following example shows the registration of a command plugin and the binding of a system command and project command function sets to the plugin. This registration is performed in the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3. Note that the normal usage is to register a system palette or a project palette (not both). Command plugins may also provide the `fz_notf_cbak_fset` function set to be notified when changes occur within **form•Z**.

```
fzrt_error_td my_cmnd_register_plugins()
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_plugin_name[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_plugin_name)
	   ) == FZRT_NOERR)
```

```
	{
		/* register the plugin as a command plugin  */
		err = fzpl_glue->fzpl_plugin_register(
			MY_PLUGIN_UUID,
			my_plugin_name,
			MY_PLUGIN_VERSION,
			MY_PLUGIN_VENDOR,
			MY_PLUGIN_URL,
			FZ_CMND_EXTS_TYPE,
			FZ_CMND_EXTS_VERSION,
			my_plugin_error_string_func,
			0,
			NULL,
			&my_plugin_runtime_id);

		/*** add a system command callback function set ***/
		if ( err == FZRT_NOERR )
		{
			err = fzpl_glue->fzpl_plugin_add_fset(
				my_plugin_runtime_id,
				FZ_CMND_CBAK_SYST_FSET_TYPE,
				FZ_CMND_CBAK_SYST_FSET_VERSION,
				FZ_CMND_CBAK_SYST_FSET_NAME,
				FZPL_TYPE_STRING(fz_cmnd_cbak_syst_fset),
				sizeof (fz_cmnd_cbak_syst_fset),
				my_fill_cmnd_cbak_syst_fset,
				FALSE);
		}
		/*** add a project command callback function set ***/
		if ( err == FZRT_NOERR )
		{
			err = fzpl_glue->fzpl_plugin_add_fset(
				my_plugin_runtime_id,
				FZ_CMND_CBAK_PROJ_FSET_TYPE,
				FZ_CMND_CBAK_PROJ_FSET_VERSION,
				FZ_CMND_CBAK_PROJ_FSET_NAME,
				FZPL_TYPE_STRING(fz_cmnd_cbak_proj_fset),
				sizeof (fz_cmnd_cbak_proj_fset),
				my_fill_cmnd_cbak_proj_fset,
				FALSE);
		}
	}
	return (err);
}
```

2.8.2.1 System Command

System command plugins are implemented by the plugin by providing the call back function set `fz_cmnd_cbak_syst_fset`. There are 13 functions in this function set. The following example shows the assignment of the plugin's own functions into the call back function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above. Note that some of these functions are optional hence a plugin would rarely implement all functions.

```
fzrt_error_td my_fill_cmnd_cbak_syst_fset (
		const fzpl_fset_def_ptr fset_def,
		fzpl_fset_td * const fset )
{
	fzrt_error_td			err = FZRT_NOERR;
	fz_cmnd_cbak_syst_fset		*cmnd_syst;

	err = fzpl_glue->fzpl_fset_def_check ( fset_def,
		FZ_CMND_CBAK_SYST_FSET_VERSION,
		FZPL_TYPE_STRING(fz_cmnd_cbak_syst_fset),
```

```
		sizeof ( fz_cmnd_cbak_syst_fset ),
		FZPL_VERSION_OP_NEWER );

	if ( err == FZRT_NOERR )
	{
		cmnd_syst = (fz_cmnd_cbak_syst_fset *)fset;

		cmnd_syst->fz_cmnd_cbak_syst_init		= my_cmnd_syst_init;
		cmnd_syst->fz_cmnd_cbak_syst_finit		= my_cmnd_syst_finit;
		cmnd_syst->fz_cmnd_cbak_syst_name		= my_cmnd_syst_name;

		cmnd_syst->fz_cmnd_cbak_syst_uuid		= my_cmnd_syst_uuid;
		cmnd_syst->fz_cmnd_cbak_syst_help		= my_cmnd_syst_help;

		cmnd_syst->fz_cmnd_cbak_syst_avail		= my_cmnd_syst_avail;
		cmnd_syst->fz_cmnd_cbak_syst_select		= my_cmnd_syst_select;
		cmnd_syst->fz_cmnd_cbak_syst_active		= my_cmnd_syst_active;

		cmnd_syst->fz_cmnd_cbak_syst_menu		= my_cmnd_syst_menu;
		cmnd_syst->fz_cmnd_cbak_syst_icon_menu		= my_cmnd_syst_icon_menu;
		cmnd_syst->fz_cmnd_cbak_syst_icon_menu_adjacent =
						my_cmnd_syst_icon_menu_adjacent;
		cmnd_syst->fz_cmnd_cbak_syst_icon_rsrc		= my_cmnd_syst_icon_rsrc;
		cmnd_syst->fz_cmnd_cbak_syst_icon_file		= my_cmnd_syst_icon_file;

		cmnd_syst->fz_cmnd_cbak_syst_pref_io		= my_cmnd_syst_pref_io;
	}

	return err;
}
```

The initialization function (optional)

```
fzrt_error_td fz_cmnd_cbak_syst_init(
	void
	);
```

This function is called by **form•Z** once when the plugin is successfully loaded and registered. The initialization function is where the plugin should initialize any data that may be needed by the other functions in the function set.

```
fzrt_error_td my_cmnd_syst_init(
	void
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/** Do initialization here **/

	return(err);
}
```

The finalization function (optional)

```
fzrt_error_td fz_cmnd_cbak_syst_finit(
	void
	);
```

This function is called by **form•Z** once when the plugin is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to free any memory allocated in the initialization function or during the life of the command.

```
fzrt_error_td my_cmnd_syst_finit(
	void
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/** Free any initialized data here **/

	return(err);
}
```

The name function (recommended)

```
fzrt_error_td fz_cmnd_cbak_syst_name(
	char		*name,
	long		max_len
	);
```

This function is called by **form•Z** to get the name of the command. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the command name string is stored in a .fzr file so that it is localizable. This function is recommended for all command plugins. If this function is not provided , the name of the plugin is used.

```
fzrt_error_td my_cmnd_syst_name(
	char		*name,
	long		max_len
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_str)) == 
FZRT_NOERR)
	{
		/* copy the string to the name parameter */
		strncpy(name,  my_str, max_len);
	}

	return(err);
}
```

The uuid function (recommended)

```
fzrt_error_td fz_cmnd_cbak_syst_uuid
	fzrt_UUID_td  uuid
	);
```

This function is called by **form•Z** to get the UUID of the command. This unique id is used by **form•Z** to distinguish the command from other commands. This function is recommended for all command plugins. If a UUID is not provided, one will be generated internally by **form•Z**. In this situation the UUID will not be the same each time **form•Z** is run and hence persistent information will not be retained. This includes any preference information provided by a supplied `fz_cmnd_cbak_syst_pref_io` function or any user customization like key shortcuts and tool icon layout.

```
#define MY_SYST_UUID 
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"
```

```
fzrt_error_td  my_cmnd_syst_uuid(
       fzrt_UUID_td  uuid
       )
{
       fzrt_error_td        err = FZRT_NOERR;

       /* copy constant UUID to into the uuid parameter */
       fzrt_UUID_copy(MY_SYST_UUID, uuid);

       return(err);
}
```

The help function (recommended)

```
fzrt_error_td fz_cmnd_cbak_syst_help(
       char          *help,
       long          max_len
       );
```

This function is called by **form•Z** to display a help string that describes the detail of what the command does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` bytes of data. It is recommended that the command name is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 512 bytes (characters).

```
fzrt_error_td my_cmnd_syst_help(
       char          *help,
       long          max_len
       )

{
       fzrt_error_td        err = FZRT_NOERR;
       char                 my_str[512];

       /* Get the help string from the plugin's resource file */
       if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) ==
FZRT_NOERR)
       {
              /* copy the string to the help parameter */
              strncpy(help,  my_str, max_len);
       }
       return(err);
}
```

The available function (recommended)

```
fzrt_error_td fz_cmnd_cbak_syst_avail(
       long                 *rv
       );
```

This function is called by **form•Z** at various times to see if the command is available. This is useful if the command is dependent on certain conditions and it is desirable to restrict its use when the conditions are not currently satisfied. If the command is not available, then it is shown as inactive (dimmed) in the **form•Z** interface (menu, icon or palette). Key shortcuts are also disabled for the command when it is not available. If this function is not provided then the command is always available.

Availability is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is available, a value of 0 indicates that the command is unavailable.

```
fzrt_error_td my_cmnd_syst_avail(
	long			*rv
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/* return 1 for available, 0 for not available */
	*rv = 1;

	return(err);
}
```

The active function (Optional)

```
fzrt_error_td cmnd_cbak_syst_active(
	long			*rv
	);
```

This function is called by **form•Z** at various times to see if the command is active. This function is needed to implement a state command where the interface element indicates the current state. This If the command is active, then it is shown selected in the **form•Z** interface. Active commands in a menu are indicated with a check mark in front of the command name. Active commands in command palettes are indicated with a highlighted icon.

Activity is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is active, a value of 0 indicates that the command is inactive. The following example shows the active function for a state command.

```
fzrt_error_td my_cmnd_syst_active(
	long			*rv
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/*** check if state is active ***/
	if(my_command->value1 == 1)	*rv = 1;
	else				*rv = 0;

	return(err);
}
```

The select function (required)

```
fzrt_error_td fz_cmnd_cbak_syst_select(
	void
	);
```

This function is called by **form•Z** when an action or state command is selected from the interface (menu, icon or palette) or when a key shortcut for the command is invoked. The select function is where the real execution for the command takes place. For action commands the desired action should be performed in this function. For state commands, the state should be changed and the appropriate actions should be taken. After the select function is executed, **form•Z** will call the active function to check for active states.

Action command example:

```
fzrt_error_td my_cmnd_syst_select(
```

```
        void
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        /*** perform  command action here ***/

        return(err);
}
```

State command example:

```
fzrt_error_td my_cmnd_syst_select(
        void
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        /*** toggle state ***/
        my_command->value1  = !my_command->value1;

        return(err);
}
```

The menu function (Optional)

```
fzrt_error_td fz_cmnd_cbak_syst_menu (
        fz_fuim_menu_ptr                menu_ptr,
        const fzrt_UUID_td              extensions_uuid,
        fzrt_UUID_td                    group_uuid,
        long                            *position
        );
```

This function is called by **form•Z** to add the command to the Extensions menu. System commands are grouped at the top of the extensions menu. The presence of this function places the command in the menu. If this function is not provided, then the command does not appear in the menu. Assigning values to the parameters of the function provides control over the placement of items in the menu. The name that appears in the menu is the name returned in the `fz_cmnd_cbak_syst_name` function.

A group of items can be placed into a pop-out heiractal menu rather than in the extensions menu itself. Calling the function `fz_fuim_exts_menu` creates a pop-out menu in the extensions menu. The `menu_ptr` and `extensions_uuid` parameters provided to the `fz_cmnd_cbak_syst_menu` function are used in the creation of the pop-out menu. The UUID of the new menu should be assigned to the `group_uuid` parameter. The pop-out menu should be created in each `fz_cmnd_cbak_syst_menu` call back function for the group so that if the grouped items are actually in separate plugins, and the user has disabled one of the plugins, the menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists that would occur if all the plugins are enabled.

form•Z will group together all commands in the extensions menu that have the same `group_uuid`. That is, all `fz_cmnd_cbak_syst_menu` implemented functions that return the same `group_uuid` parameter are placed together in the extensions menu in a group separated from other items by a menu separator. The `position` parameter specifies the order of the items. The items in the group are sorted from lowest to highest position. If `position` is set to 0, the items are placed in alphabetic order.

The following is an example of a menu function with a pop-out menu.

```
#define MY_GRUP_UUID
"\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

fzrt_error_td my_cmnd_syst_menu (
	fz_fuim_menu_ptr			menu_ptr,
	const fzrt_UUID_td			extensions_uuid,
	fzrt_UUID_td				group_uuid,
	long					*position
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)
	   ) == FZRT_NOERR)
	{
		/* create the menu group */
		err = fz_fuim_exts_menu(menu_ptr, extensions_uuid, my_str,
					MY_GRUP_UUID);

		if(err == FZRT_NOERR)
		{
			fzrt_UUID_copy(MY_GRUP_UUID, group_uuid);
			*position = 1;
		}
	}
	return(err);
}
```

Nested menus can be created up to 3 levels of hierarchy by passing the uuid of another pop-out menu to the `fz_fuim_exts_menu` function. The following is an example of a nested pop-out menu.

```
#define MY_GRUP_UUID_NEST "\x24\xf6\x35\x41\x6b\xab\x7f\xb4\xa5\x6a\xd5\xaa\x65\x36\xfb\xe0"

fzrt_error_td my_cmnd_syst_menu (
	fz_fuim_menu_ptr		menu_ptr,
	const fzrt_UUID_td		extensions_uuid,
	fzrt_UUID_td			group_uuid,
	long				*position
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)
	    ) == FZRT_NOERR)
	{
		/* create the menu group */
		if((err = fz_fuim_exts_menu(menu_ptr, extensions_uuid, my_str,
			 MY_GRUP_UUID)) == FZRT_NOERR)
		{

			/* Get title string from the resource file */
			err = fzrt_fzr_get_string(my_rfzr_refid, 1, 3, my_str);

			if(err == FZRT_NOERR)
			{
```

```
                        /* create the nested menu group */
                        err = fz_fuim_exts_menu(menu_ptr, MY_GRUP_UUID,
                                    my_str, MY_GRUP_UUID_NEST);

                        if(err == FZRT_NOERR)
                        {       fzrt_UUID_copy(MY_GRUP_UUID_NEST, group_uuid);
                                *position = 1;
                        }
                }
            }
        }
        return(err);
}
```

By default menu items are enabled. The `fz_mnd_cbak_syst_avail` function can be used to disable the command and make its menu item shown dimmed. Menu items for state commands are shown with a check mark when the `fz_cmnd_cbak_syst_active` function indicates that the state for the command is active.

The icon menu function (Optional, mutually exclusive with icon menu adjacent function)

```
fzrt_error_td fz_cmnd_cbak_syst_icon_menu (
        const fzrt_UUID_td              icon_menu_uuid,
        fzrt_UUID_td                    group_uuid,
        fz_fuim_icon_group_enum         *group_pos,
        long                            *group_row,
        long                            *group_col
        );
```

This function is called by **form•Z** to add the command to the system command icon menu palette. The presence of this function places the command in the palette. If no other parameters are set then the command will get added to a group of icons at the bottom (end) of the icon menu. Note that this only adds the position to the icon palette. The function `fz_cmnd_cbak_syst_icon_rsrc` or `fz_cmnd_cbak_syst_icon_file` must be provided to add custom graphics for the icon. If one of these is not provided, **form•Z** uses a generic plugin icon graphic.

The `group_uuid` parameter is assigned to all commands that should be grouped together. That is, all `fz_cmnd_cbak_syst_icon_menu` implemented functions that return the same `group_uuid` parameter are placed together in the system icon menu in the same group (pop-out tool menu). This group is added to the bottom (end) of the menu. The placement of the item in the group is controlled by the `group_pos` parameter. A value of `FZ_FUIM_ICON_GROUP_START` places the item at the start of the group and a value of `FZ_FUIM_ICON_GROUP_END` places it at the end of the group. Note that these may not always yield constant results because plugin load order can vary hence multiple uses of `FZ_FUIM_ICON_GROUP_END` my note build the icon palette in the expected order. When `FZ_FUIM_ICON_GROUP_CUSTOM` is selected, then the `group_row` and `group_col` parameters specify the position of the item in the tool menu group.

```
#define MY_GRUP_UUID
"\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

fzrt_error_td my_cmnd_syst_icon_menu (
        const fzrt_UUID_td              icon_menu_uuid,
        fzrt_UUID_td                    group_uuid,
        fz_fuim_icon_group_enum         *group_pos,
        long                            *group_row,
        long                            *group_col
        )
```

```
{
	fzrt_error_td		err = FZRT_NOERR;

	fzrt_UUID_copy(MY_GRUP_UUID, group_uuid);
	*group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
	*group_row = 1;
	*group_col = 1;

	return(err);
}
```

The function `fz_fz_fuim_exts_icon_group` can be called to better control the group containing the set of commands. This adds the ability to name the group and insert the pop-out menu group in the existing menu groups. The icon pop-out menu can be created in each `fz_cmnd_cbak_syst_icon_menu` so that if the grouped items are actually in separate plugins, and the user has disabled one of the plugins, the icon menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists that would occur if all the plugins are enabled. The following is an example of a pop-out menu.

```
fzrt_error_td my_cmnd_syst_icon_menu (
	const fzrt_UUID_td			icon_menu_uuid,
	fzrt_UUID_td				group_uuid,
	fz_fuim_icon_group_enum		*group_pos,
	long					*group_row,
	long					*group_col
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	err = fz_fuim_exts_icon_group(icon_menu_uuid,
		"My  Group", MY_GRUP_UUID,
		FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE,
		FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE);

	if(err == FZRT_NOERR)
	{	fzrt_UUID_copy(MY_GRUP_UUID, group_uuid);
		*group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
		*group_row = 1;
		*group_col = 1;
	}
	return(err);
}
```

The icon menu adjacent function (Optional, mutually exclusive with icon menu function)

```
fzrt_error_td fz_cmnd_cbak_syst_icon_menu_adjacent(
	const fzrt_UUID_td			icon_menu_uuid,
	fzrt_UUID_td				adjacent_uuid,
	fz_fuim_icon_adjacent_enum	*where
	);
```

This function is called by **form•Z** to add the command to the command icon menu palette. It serves the same purpose as the `fz_cmnd_cbak_syst_icon_menu` function, however it specifies the location of the icon item quite differently. The location is identified by referencing another command in the icon menu. The `adjacent_uuid` parameter is the UUID of the command to which the icon should be added adjacent. The `where` parameter specifies to which side of the adjacent icon the icon should be added. The available options are `FZ_FUIM_ICON_ADJACENT_TOP, FZ_FUIM_ICON_ADJACENT_BOTTOM,`

`FZ_FUIM_ICON_ADJACENT_LEFT, FZ_FUIM_ICON_ADJACENT_RIGHT`. The default action is specified by `FZ_FUIM_ICON_ADJACENT_DEFAULT` which currently is the same as `FZ_FUIM_ICON_ADJACENT_RIGHT`. New pop-out groups can not be created with this function. The following example ads the icon to the right of the **form•Z** save command.

```
fzrt_error_td my_cmnd_syst_icon_menu_adjacent (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			adjacent_uuid,
	fz_fuim_icon_adjacent_enum	*where
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	fzrt_UUID_copy(CMND_SAVE, adjacent_uuid);
	*where = FZ_FUIM_ICON_ADJACENT_RIGHT;

	return(err);
}
```

The icon file function (Optional, mutually exclusive with icon resource function)

```
fzrt_error_td fz_cmnd_cbak_syst_icon_file (
	fz_fuim_icon_enum		which,
	fzrt_floc_ptr			floc,
	long				*hpos,
	long				*vpos,
	fzrt_floc_ptr			floc_mask,
	long				*hpos_mask,
	long				*vpos_mask
	);
```

This function is called by **form•Z** to get an icon for the command from an image file. The icon image can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. **form•Z** will request an icon when the command is displayed in a tool menu using `fz_cmnd_cbak_syst_icon_menu` or `fz_cmnd_cbak_syst_icon_menu_adjacent`.

form•Z supports 3 styles of icon display. Recall that these are selectable by the user from the Icon Style menu in the Icons Customization dialog. The first two options (White and Gray) are generated from a black and white source graphic with different treatments at drawing time. The third option is generated from a color source graphic. The first two options are older icon styles that are provided for backward compatibility. The color icons became the default with v 4.0. Note that if an icon of one type or the other (or both) is not provided, then **form•Z** uses a generic plugin icon graphic.

The `which` parameter indicates the type of source graphic icon that is needed by **form•Z**. For each type of icon source (black and white and color), there are two possible sizes. The full size icon is the size that is used in the main tool palettes and tear off tool palettes. The black and white source full size is 30 x 30 pixels and indicated by `FZ_FUIM_ICON_MONOC`. The color source is 32 x 32 pixels and indicated by `FZ_FUIM_ICON_COLOR`. The alternate size is the smaller size used for window icons that are drawn in the lower margin of the window. The alternate size for both black and white and color sources is 20 x 16 pixels and indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively.

The `floc` parameter should be filled with the file name and location of the file that contains the icon graphic. The `hpos` and `vpos` parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the plugin file. This makes it simple to find the file. The location of the plugin file can be retained during the `FZPL_PLUGIN_INITIALIZE` stage using the `fzpl_glue->fzpl_plugin_file_get_floc` function.

The `floc_mask` parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the `floc` parameter). The icon mask defines the transparent areas of the icon. The `hpos_mask` and `vpos_mask` parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided than the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of commands by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

```
fzrt_error_td my_cmnd_syst_icon_file (
	fz_fuim_icon_enum		which,
	fzrt_floc_ptr			floc,
	long				*hpos,
	long				*vpos,
	fzrt_floc_ptr			floc_mask,
	long				*hpos_mask,
	long				*vpos_mask
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	switch(which)
	{
		case FZ_FUIM_ICON_MONOC:
			err = fzrt_file_floc_copy(my_plugin_floc,floc);
			if(err == FZRT_NOERR)
			{
				err = fzrt_file_floc_set_name(floc,
						"my_icon_bw.tif");
				*hpos = 0;
				*vpos = 0;
			}
		break;
		case FZ_FUIM_ICON_COLOR:
			err = fzrt_file_floc_copy(my_plugin_floc,floc);
			if(err == FZRT_NOERR)
			{
				err = fzrt_file_floc_set_name(floc,
						"my_icon_col.tif");
				*hpos = 0;
				*vpos = 0;
			}
		break;
	}
	return(err);
}
```

The icon resource function (Optional, mutually exclusive with icon file function)

```
fzrt_error_td fz_cmnd_cbak_syst_icon_rsrc (
	fz_fuim_icon_enum		which,
	fzrt_icon_ptr			*icon
	);
```

This function is called by **form•Z** to load an icon for the command from a platform's native (Macintosh or Windows) resource file format. This function works the same as the above icon file function except that the icon data is read from the resource file instead of the image file. These two functions are mutually exclusive (only one should be provided). Although this function and the method for loading resources is cross platform, the resource formats are not hence the data must be generated differently for each platform. This function is provided for situations where resources in these formats are already available. It is recommended that all new artwork use the icon file method described above as it is cross platform and simpler to create the content.

This function can be used to load the icon from the plugin file's resource data by using the function `fzpl_plugin_get_rlib_idx` to obtain the index for the plugins files resource data. The function `fzrt_rlib_load_icon` must be called to load the resource from the file. Use `FZRT_LOAD_ICON_BW` to indicate black and white icons and indicate color icons using `FZRT_LOAD_ICON_COLOR`. On the Macintosh platform, the black and white icons are read from 'ICON' resources and color icons from 'cicn'. On Windows black and white icons must be stored as a 1 bit depth bitmap resource with the type "FZICON" in the resource file and color icons can be stored as either a native Windows ICON or as an 8 bit deep bitmap resource. Note that on Windows, black and white icons and color icons stored as a bitmap resource will not have an icon mask. **form•Z** releases the memory for the resource when the plugin is unloaded.

All icons are stored in 32 x 32 pixel resources, however, depending on the type of the icon, only part of the resource will be used. Only the top left 30 x 30 pixels of the 32 x 32 are used for the black and white full icon size indicated by `FZ_FUIM_ICON_MONOC`. The bottom and right two pixels are NOT used (and will be cropped). The entire 32 x 32 is used for the color full icon size indicated by `FZ_FUIM_ICON_COLOR`. For the alternate size icons indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively, **form•Z** uses the bottom left 20 x 16 pixels. The top 16 and right 12 pixels are NOT used (and will be cropped).

```
fzrt_error_td my_cmnd_syst_icon_rsrc (
	fz_fuim_icon_enum		which,
	fzrt_icon_ptr			*icon
	)
{
	long				err = FZRT_NOERR;
	short				rlib_index;

	err = fzpl_plugin_get_rlib_idx(my_plugin_runtime_id, &rlib_index );

	if(err == FZRT_NOERR)
	{
		switch(which)
		{
			case FZ_FUIM_ICON_MONOC:
				err = fzrt_rlib_load_icon(
					rlib_index,FZRT_LOAD_ICON_BW,128,icon);
			break;
			case FZ_FUIM_ICON_COLOR:
				err = fzrt_rlib_load_icon(
					rlib_index,FZRT_LOAD_ICON_COLOR,128,icon);
			break;
		}
	}
	return(err);
}
```

The preferences IO function (optional)

```
fzrt_error_td fz_cmnd_cbak_syst_pref_io (
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum			dir,
```

```
        fzpl_vers_td * const            version,
        unsigned long                   size
        );
```

form•Z calls this function to read and write any command specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_cmnd_syst_iost(
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum                     dir,
        fzpl_vers_td * const            version,
        unsigned long                   size
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        if ( dir == FZ_IOST_WRITE ) *version = 1;

        err = fz_iost_one_long(iost,&my_command->value1);
        if(err == FZRT_NOERR)
        {       err = fz_iost_one_long(iost,&my_command->value2);
                if(err == FZRT_NOERR)
                {       err = fz_iost_one_long(iost,&my_command->value3);
                        if(err == FZRT_NOERR)
                        {       err = fz_iost_one_long(iost,&my_command->value4);

                                if(*version >= 1)
                                {       err = fz_iost_one_long(iost,
                                                        &my_command->value5);
                                }
                        }
                }
        }

        return(err);
}
```

2.8.2.2 Project Commands

Project commands are defined using the `FZ_CMND_PROJ_PLUGIN_TYPE` and the `fz_cmnd_cbak_proj_fset` function set as described in the following sections. There are 17 functions in this function set. The following shows the fill in of a `fz_cmnd_cbak_proj_fset` function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above. Note that some of these functions are optional and some are mutually exclusive hence a plugin would never implement all of these functions.

```
fzrt_error_td my_fill_cmnd_cbak_proj_fset (
            const fzpl_fset_def_ptr fset_def,
            fzpl_fset_td * const fset )
{
      fzrt_error_td                     err = FZRT_NOERR;
      fz_cmnd_cbak_proj_fset            *cmnd_proj;

      err = fzpl_glue->fzpl_fset_def_check ( fset_def,
            FZ_CMND_CBAK_PROJ_FSET_VERSION,
            FZPL_TYPE_STRING(fz_cmnd_cbak_proj_fset),
            sizeof ( fz_cmnd_cbak_proj_fset ),
            FZPL_VERSION_OP_NEWER );

      if ( err == FZRT_NOERR )
      {
            cmnd_proj = (fz_cmnd_cbak_proj_fset *)fset;

            cmnd_proj->fz_cmnd_cbak_proj_init             = my_cmnd_proj_init;
            cmnd_proj->fz_cmnd_cbak_proj_finit            = my_cmnd_proj_finit;
            cmnd_proj->fz_cmnd_cbak_proj_info             = my_cmnd_proj_info;
            cmnd_proj->fz_cmnd_cbak_proj_name             = my_cmnd_proj_name;

            cmnd_proj->fz_cmnd_cbak_proj_uuid             = my_cmnd_proj_uuid;
            cmnd_proj->fz_cmnd_cbak_proj_help             = my_cmnd_proj_help;

            cmnd_proj->fz_cmnd_cbak_proj_avail            = my_cmnd_proj_avail;
            cmnd_proj->fz_cmnd_cbak_proj_select           = my_cmnd_proj_select;
            cmnd_proj->fz_cmnd_cbak_proj_active           = my_cmnd_proj_active;

            cmnd_proj->fz_cmnd_cbak_proj_menu             = my_cmnd_proj_menu;
            cmnd_proj->fz_cmnd_cbak_proj_icon_menu        = my_cmnd_proj_icon_menu;
            cmnd_proj->fz_cmnd_cbak_proj_icon_menu_adjacent =
                                                  my_cmnd_proj_icon_menu_adjacent;
            cmnd_proj->fz_cmnd_cbak_proj_icon_rsrc        = my_cmnd_proj_icon_rsrc;
            cmnd_proj->fz_cmnd_cbak_proj_icon_file        = my_cmnd_proj_icon_file;

            cmnd_proj->fz_cmnd_cbak_proj_pref_io          = my_cmnd_proj_pref_io;
            cmnd_proj->fz_cmnd_cbak_proj_data_io          = my_cmnd_proj_data_io;
            cmnd_proj->fz_cmnd_cbak_proj_wind_data_io     = my_cmnd_proj_wind_data_io;
      }

      return err;
}
```

The initialization function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_init(
      void
      );
```

This function is called by **form•Z** once when the plugin is successfully loaded and registered. The initialization function is where the plugin should initialize any data that may be needed by the other functions in the function set.

```
fzrt_error_td my_cmnd_proj_init(
       void
       )
{
       fzrt_error_td       err = FZRT_NOERR;

       /** Do initialization here **/

       return(err);
}
```

The finalization function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_finit(
       void
       );
```

This function is called by **form•Z** once when the plugin is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to free any memory allocated in the initialization function or during the life of the command.

```
fzrt_error_td my_cmnd_proj_finit(
       void
       )
{
       fzrt_error_td       err = FZRT_NOERR;

       /** Free any initialized data here **/

       return(err);
}
```

The info function (required)

```
fzrt_error_td fz_cmnd_cbak_proj_info(
       fz_proj_level_enum  *level
       );
```

This function is called by **form•Z** once when the plugin is successfully loaded to determine the kind of command that is implemented by the function set.

The `level` parameter indicates the context of the tool. **form•Z** uses the value in this parameter to determine when the command should be shown and when it should be updated. The following are the available values:

- `FZ_PROJ_LEVEL_MODEL`: Indicates that the tool operates on the projects modeling content (objects for example).
- `FZ_PROJ_LEVEL_MODEL_WIND`: Indicates that the tool operates on modeling window specific content (views for example) of modeling windows.
- `FZ_PROJ_LEVEL_DRAFT`: Indicates that the tool operates on the projects drafting content (elements for example).
- `FZ_PROJ_LEVEL_DRAFT_WIND`: Indicates that the tool operates on drafting window specific content (views for example) of drafting windows.

```
fzrt_error_td my_cmnd_proj_info(
	fz_proj_level_enum	*level
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/* indicate modeling level */
	*level = FZ_PROJ_LEVEL_MODEL;

	return(err);
}
```

The name function (recommended)

```
fzrt_error_td fz_cmnd_cbak_proj_name(
	char		*name,
	long		max_len
	);
```

This function is called by **form•Z** to get the name of the command. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the command name string is stored in a .fzr file so that it is localizable. This function is recommended for all command plugins. If this function is not provided , the name of the plugin is used.

```
fzrt_error_td my_cmnd_proj_name(
	char		*name,
	long		max_len
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_str)
	   ) == FZRT_NOERR)
	{
		/* copy the string to the name parameter */
		strncpy(name, my_str, max_len);
	}

	return(err);
}
```

The uuid function (recommended)

```
fzrt_error_td fz_cmnd_cbak_proj_uuid
	fzrt_UUID_td	uuid
	);
```

This function is called by **form•Z** to get the UUID of the command. This unique id is used by **form•Z** to distinguish the command from other commands. This function is recommended for all command plugins. If a UUID is not provided, one will be generated internally by **form•Z**. In this situation the UUID will not be the same each time **form•Z** is run and hence persistent information will not be retained. This includes any preference information provided by a supplied `fz_cmnd_cbak_proj_pref_io` function or any user customization like key shortcuts and tool icon layout.

```
#define MY_PROJ_UUID
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td  my_cmnd_proj_uuid(
	fzrt_UUID_td uuid
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/* copy constant UUID to into the uuid parameter */
	fzrt_UUID_copy(MY_PROJ_UUID, uuid);

	return(err);
}
```

The help function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_help(
	char		*help,
	long		max_len
	);
```

This function is called by **form•Z** to display a help string that describes the detail of what the command does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` bytes of data. It is recommended that the help text is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 512 bytes (characters).

```
fzrt_error_td my_cmnd_proj_help(
	char		*help,
	long		max_len
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[512];

	/* Get the help string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)
	   ) == FZRT_NOERR)
	{
		/* copy the string to the help parameter */
		strncpy(help, my_str, max_len);
	}
	return(err);
}
```

The available function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_avail(
	long			windex,
	long			*rv
	);
```

This function is called by **form•Z** at various times to see if the command is available. This is useful if the command is dependent on certain conditions and it is desirable to restrict its use when the conditions are not currently satisfied. If the command is not available, then it is shown as inactive (dimmed) in the **form•Z** interface (menu, icon or palette). Key shortcuts are also disabled for the command when it is not available. If this function is not provided then the command is always available.

Availability is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is available, a value of 0 indicates that the command is unavailable.

```
fzrt_error_td my_cmnd_proj_avail(
	long			windex,
	long			*rv
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/* return 1 for available, 0 for not available */
	*rv = 1;

	return(err);
}
```

The active function (Optional)

```
fzrt_error_td fz_cmnd_cbak_proj_active(
	long			windex,
	long			*rv
	);
```

This function is called by **form•Z** at various times to see if the command is active. This function is needed to implement a state command where the interface element indicates the current state. This If the command is active, then it is shown selected in the **form•Z** interface. Active commands in a menu are indicated with a check mark in front of the command name. Active commands in command palettes are indicated with a highlighted icon.

Activity is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is active, a value of 0 indicates that the command is inactive. The following example shows the active function for a state command.

```
fzrt_error_td my_cmnd_proj_active(
	long			windex,
	long			*rv
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/*** check if state is active ***/
	if(my_command->value1 == 1)	*rv = 1;
	else				*rv = 0;

	return(err);
}
```

The select function (required)

```
fzrt_error_td fz_cmnd_cbak_proj_select(
	long			windex
	);
```

This function is called by **form•Z** when an action or state command is selected from the interface (menu, icon or palette) or when a key shortcut for the command is invoked. The select function is where the real execution for the command takes place. For action commands the desired action should be performed in this function. For state commands, the state should be changed and the appropriate actions should be taken. After the select function is executed, **form•Z** will call the active function to check for active states.

Action command example:

```
fzrt_error_td my_cmnd_proj_select(
	long			windex
	)
{
	fzrt_error_td		err = FZRT_NOERR;


	/*** perform  command action here ***/

	return(err);
}
```

State command example:

```
fzrt_error_td my_cmnd_proj_select(
	long			windex
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/*** toggle state ***/
	my_command->value1  = ! my_command->value1;

	return(err);
}
```

The menu function (Optional)

```
fzrt_error_td fz_cmnd_cbak_proj_menu (
	fz_fuim_menu_ptr			menu_ptr,
	const fzrt_UUID_td			extensions_uuid,
	fzrt_UUID_td				group_uuid,
	long					*position
	);
```

This function is called by **form•Z** to add the command to the Extensions menu. Project commands are grouped at the top of the Extensions menu. The presence of this function places the command in the menu. If this function is not provided, then the command does not appear in the menu. Assigning values to the parameters of the function provides control over the placement of items in the menu. The name that appears in the menu is the name returned in the `fz_cmnd_cbak_proj_name` function.

A group of items can be placed into a pop-out hierarchical menu rather than in the Extensions menu itself. Calling the function `fz_fuim_exts_menu` creates a pop-out menu in the Extensions menu. The `menu_ptr` and `extensions_uuid` parameters provided to the `fz_cmnd_cbak_proj_menu` function are used in the creation of the pop-out menu. The UUID of the new menu should be assigned to the `group_uuid` parameter. The pop-out menu should be created in each `fz_cmnd_cbak_proj_menu` call back function for the group so that if the grouped items are actually in separate plugins, and the user has disabled one of the plugins, the menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists that would occur if all the plugins are enabled.

form•Z will group together all commands in the extensions menu that have the same `group_uuid`. That is, all `fz_cmnd_cbak_proj_menu` implemented functions that return the same `group_uuid` parameter are placed together in the extensions menu in a group separated from other items by a menu separator. The `position` parameter specifies the order of the items. The items in the group are sorted from lowest to highest position. If `position` is set to 0, the items are placed in alphabetic order.

The following is an example of a menu function with a pop-out menu.

```
#define MY_GRUP_UUID
"\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

fzrt_error_td my_cmnd_proj_menu (
	fz_fuim_menu_ptr			menu_ptr,
	const fzrt_UUID_td			extensions_uuid,
	fzrt_UUID_td				group_uuid,
	long					*position
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)
	   ) == FZRT_NOERR)
	{
		/* create the menu group */
		err = fz_fuim_exts_menu(menu_ptr, extensions_uuid, my_str,
					MY_GRUP_UUID);

		if(err == FZRT_NOERR)
		{
			fzrt_UUID_copy(MY_GRUP_UUID, group_uuid);
			*position = 1;
		}
	}
	return(err);
}
```

Nested menus can be created up to 3 levels of hierarchy by passing the uuid of another pop-out menu to the `fz_fuim_exts_menu` function. The following is an example of a nested pop-out menu.

```
#define MY_GRUP_UUID_NEST "\x24\xf6\x35\x41\x6b\xab\x7f\xb4\xa5\x6a\xd5\xaa\x65\x36\xfb\xe0"

fzrt_error_td my_cmnd_proj_menu (
	fz_fuim_menu_ptr		menu_ptr,
	const fzrt_UUID_td		extensions_uuid,
	fzrt_UUID_td			group_uuid,
	long				*position
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_str[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)
	    ) == FZRT_NOERR)
	{
		/* create the menu group */
		if((err = fz_fuim_exts_menu (
```

```
                menu_ptr, extensions_uuid, my_str, MY_GRUP_UUID)) ==
            FZRT_NOERR)
            {

                /* Get title string from the resource file */
                err = fzrt_fzr_get_string(my_rfzr_refid, 1, 3, my_str);

                if(err == FZRT_NOERR)
                {
                    /* create the nested menu group */
                    err = fz_fuim_exts_menu (
                        menu_ptr, MY_GRUP_UUID, my_str,
                    MY_GRUP_UUID_NEST);

                    if(err == FZRT_NOERR)
                    {   fzrt_UUID_copy(MY_GRUP_UUID_NEST, group_uuid);
                        *position = 1;
                    }
                }
            }
        }
        return(err);
}
```

By default menu items are enabled. The `fz_cmnd_cbak_proj_avail` function can be used to disable the command and make its menu item shown dimmed. Menu items for state commands are shown with a check mark when the `fz_cmnd_cbak_proj_active` function indicates that the state for the command is active.

The icon menu function (Optional, mutually exclusive with icon menu adjacent function)

```
fzrt_error_td fz_cmnd_cbak_proj_icon_menu (
        const fzrt_UUID_td              icon_menu_uuid,
        fzrt_UUID_td                    group_uuid,
        fz_fuim_icon_group_enum         *group_pos,
        long                            *group_row,
        long                            *group_col
        );
```

This function is called by **form•Z** to add the command to the commands icon menu palette. The presence of this function places the command in the icon menu palette. If no other parameters are set then the command will get added to a group of icons at the bottom (end) of the icon menu. Note that this only adds the position to the tool menu. The function `fz_cmnd_cbak_proj_icon_rsrc` or `fz_cmnd_cbak_proj_icon_file` must be provided to add custom graphics for the icon. If one of these is not provided, **form•Z** uses a generic plugin icon graphic.

The `group_uuid` parameter is assigned to all commands that should be grouped together. That is, all `fz_cmnd_cbak_proj_icon_menu` implemented functions that return the same `group_uuid` parameter are placed together in the system icon menu in the same group (pop-out tool menu). This group is added to the bottom (end) of the menu. The placement of the item in the group is controlled by the `group_pos` parameter. A value of `FZ_FUIM_ICON_GROUP_START` places the item at the start of the group and a value of `FZ_FUIM_ICON_GROUP_END` places it at the end of the group. Note that these may not always yield constant results because plugin load order can vary hence multiple uses of `FZ_FUIM_ICON_GROUP_END` my note build the menu in the expected order. When `FZ_FUIM_ICON_GROUP_CUSTOM` is selected, then the `group_row` and `group_col` parameters specify the position of the item in the tool menu group.

```
#define MY_GRUP_UUID
"\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

fzrt_error_td my_cmnd_proj_icon_menu (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			group_uuid,
	fz_fuim_icon_group_enum		*group_pos,
	long				*group_row,
	long				*group_col
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	fzrt_UUID_copy(MY_GRUP_UUID, group_uuid);
	*group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
	*group_row = 1;
	*group_col = 1;

	return(err);
}
```

The function `fz_fuim_exts_icon_group` can be called to better control the group containing the set of commands. This adds the ability to name the group and insert the pop-out menu group in the existing menu groups. The icon pop-out menu can be created in each `fz_cmnd_cbak_proj_icon_menu` so that if the grouped items are actually in separate plugins, and the user has disabled one of the plugins, the icon menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists that would occur if all the plugins are enabled. The following is an example of a pop-out menu.

```
fzrt_error_td my_cmnd_proj_icon_menu (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			group_uuid,
	fz_fuim_icon_group_enum		*group_pos,
	long				*group_row,
	long				*group_col
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	err = fz_fuim_exts_icon_group(
		"My  Group", MY_GRUP_UUID, icon_menu_uuid,
		FZRT_UUID_NULL, FZ_FUIM_CMND_POS_NONE,
		FZRT_UUID_NULL, FZ_FUIM_CMND_POS_NONE);

	if(err == FZRT_NOERR)
	{	fzrt_UUID_copy(MY_GRUP_UUID, group_uuid);
		*group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
		*group_row = 1;
		*group_col = 1;
	}
	return(err);
}
```

The icon menu adjacent function (Optional, mutually exclusive with icon menu function)

```
fzrt_error_td fz_cmnd_cbak_proj_icon_menu_adjacent (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			adjacent_uuid,
	fz_fuim_icon_adjacent_enum	*where
	);
```

This function is called by **form•Z** to add the command to the system icon menu. It serves the same purpose as the fz_cmnd_cbak_proj_icon_menu function, however it specifies the location of the icon item quite differently. The location is identified by referencing another command in the icon menu. The adjacent_uuid parameter is the UUID of the command to which the icon should be added adjacent. The where parameter specifies to which side of the adjacent icon the icon should be added. The available options are FZ_FUIM_ICON_ADJACENT_TOP, FZ_FUIM_ICON_ADJACENT_BOTTOM, FZ_FUIM_ICON_ADJACENT_LEFT, FZ_FUIM_ICON_ADJACENT_RIGHT. The default action is specified by FZ_FUIM_ICON_ADJACENT_DEFAULT which currently is the same as FZ_FUIM_ICON_ADJACENT_RIGHT. New pop-out groups can not be created with this function. The following example ads the icon to the right of the **form•Z** save command.

```
fzrt_error_td my_cmnd_proj_icon_menu_adjacent (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			adjacent_uuid,
	fz_fuim_icon_adjacent_enum	*where
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	fzrt_UUID_copy(CMND_SAVE, adjacent_uuid);
	*where = FZ_FUIM_ICON_ADJACENT_RIGHT;

	return(err);
}
```

The icon file function (Optional, mutually exclusive with icon resource function)

```
fzrt_error_td fz_cmnd_cbak_proj_icon_file (
	fz_fuim_icon_enum		which,
	fzrt_floc_ptr			floc,
	long				*hpos,
	long				*vpos,
	fzrt_floc_ptr			floc_mask,
	long				*hpos_mask,
	long				*vpos_mask
	);
```

This function is called by **form•Z** to get an icon for the command from an image file. The icon image can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. **form•Z** will request an icon when the command is displayed in a tool menu using fz_cmnd_cbak_proj_icon_menu or fz_cmnd_cbak_proj_icon_menu_adjacent.

form•Z supports 3 styles of icon display. Recall that these are selectable by the user from the Icon Style menu in the Icons Customization dialog. The first two options (White and Gray) are generated from a black and white source graphic with different treatments at drawing time. The third option is generated from a color source graphic. The first two options are older icon styles that are provided for backward compatibility. The color icons became the default with v 4.0. Note that if an icon of one type or the other (or both) is not provided, then **form•Z** uses a generic plugin icon graphic.

The which parameter indicates the type of source graphic icon that is needed by **form•Z**. For each type of icon source (black and white and color), there are two possible sizes. The full size icon is the size that is used in the main tool palettes and tear off tool palettes. The black and white source full size is 30 x 30 pixels and indicated by FZ_FUIM_ICON_MONOC. The color source is 32 x 32 pixels and indicated by `FZ_FUIM_ICON_COLOR`. The alternate size is the smaller size used for window icons that are drawn in the lower margin of the window. The alternate size for both black and white and color sources is 20 x 16 pixels and indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively.

The `floc` parameter should be filled with the file name and location of the file that contains the icon graphic. The `hpos` and `vpos` parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the plugin file. This makes it simple to find the file. The location of the plugin file can be retained during the `FZPL_PLUGIN_INITIALIZE` stage using the `fzpl_glue->fzpl_plugin_file_get_floc` function.

The `floc_mask` parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the `floc` parameter). The icon mask defines the transparent areas of the icon. The `hpos_mask` and `vpos_mask` parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided than the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of commands by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

```
fzrt_error_td my_cmnd_proj_icon_file (
	fz_fuim_icon_enum		which,
	fzrt_floc_ptr			floc,
	long				*hpos,
	long				*vpos,
	fzrt_floc_ptr			floc_mask,
	long				*hpos_mask,
	long				*vpos_mask
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	switch(which)
	{
		case FZ_FUIM_ICON_MONOC:
			err = fzrt_file_floc_copy(my_plugin_floc,floc);
			if(err == FZRT_NOERR)
			{
				err = fzrt_file_floc_set_name(floc,
						"my_icon_bw.tif");
				*hpos = 0;
				*vpos = 0;
			}
		break;
		case FZ_FUIM_ICON_COLOR:
			err = fzrt_file_floc_copy(my_plugin_floc,floc);
			if(err == FZRT_NOERR)
			{
				err = fzrt_file_floc_set_name(floc,
						"my_icon_col.tif");
				*hpos = 0;
				*vpos = 0;
			}
		break;
	}
	return(err);
}
```

The icon resource function (Optional, mutually exclusive with icon file function)

```
fzrt_error_td fz_cmnd_cbak_proj_icon_rsrc (
	fz_fuim_icon_enum		which,
	fzrt_icon_ptr			*icon
	);
```

This function is called by **form•Z** to load an icon for the command from a platform's native (Macintosh or Windows) resource file format. This function works the same as the above icon file function except that the icon data is read from the resource file instead of the image file. These two functions are mutually exclusive (only one should be provided). Although this function and the method for loading resources is cross platform, the resource formats are not hence the data must be generated differently for each platform. This function is provided for situations where resources in these formats are already available. It is recommended that all new artwork use the icon file method described above as it is cross platform and simpler to create the content.

This function can be used to load the icon from the plugin file's resource data by using the function `fzpl_plugin_get_rlib_idx` to obtain the index for the plugins files resource data. The function `fzrt_rlib_load_icon` must be called to load the resource from the file. Use `FZRT_LOAD_ICON_BW` to indicate black and white icons and indicate color icons using `FZRT_LOAD_ICON_COLOR`. On the Macintosh platform, the black and white icons are read from 'ICON' resources and color icons from 'cicn'. On Windows black and white icons must be stored as a 1 bit depth bitmap resource with the type "FZICON" in the resource file and color icons can be stored as either a native Windows ICON or as an 8 bit deep bitmap resource. Note that on Windows, black and white icons and color icons stored as a bitmap resource will not have an icon mask. **form•Z** releases the memory for the resource when the plugin is unloaded.

All icons are stored in 32 x 32 pixel resources, however, depending on the type of the icon, only part of the resource will be used. Only the top left 30 x 30 pixels of the 32 x 32 are used for the black and white full icon size indicated by `FZ_FUIM_ICON_MONOC`. The bottom and right two pixels are NOT used (and will be cropped). The entire 32 x 32 is used for the color full icon size indicated by `FZ_FUIM_ICON_COLOR`. For the alternate size icons indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively, **form•Z** uses the bottom left 20 x 16 pixels. The top 16 and right 12 pixels are NOT used (and will be cropped).

```
fzrt_error_td my_cmnd_proj_icon_rsrc (
	fz_fuim_icon_enum		which,
	fzrt_icon_ptr			*icon
	)
{
	long				err = FZRT_NOERR;
	short				rlib_index;

	err = fzpl_plugin_get_rlib_idx(my_plugin_runtime_id, &rlib_index );

	if(err == FZRT_NOERR)
	{
		switch(which)
		{
			case FZ_FUIM_ICON_MONOC:
				err = fzrt_rlib_load_icon(
					rlib_index,FZRT_LOAD_ICON_BW,128,icon);
			break;
			case FZ_FUIM_ICON_COLOR:
				err = fzrt_rlib_load_icon(
					rlib_index,FZRT_LOAD_ICON_COLOR,128,icon);
			break;
		}
```

```
	}
	return(err);
}
```

The preferences IO function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_pref_io (
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum			dir,
	fzpl_vers_td * const			version,
	unsigned long				size
	);
```

form•Z calls this function to read and write any command specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_cmnd_proj_pref_io(
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum			dir,
	fzpl_vers_td * const			version,
	unsigned long				size
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	if ( dir == FZ_IOST_WRITE ) *version = 1;

	err = fz_iost_one_long(iost,&my_command->value1);
	if(err == FZRT_NOERR)
	{	err = fz_iost_one_long(iost,&my_command->value2);
		if(err == FZRT_NOERR)
		{	err = fz_iost_one_long(iost,&my_command->value3);
			if(err == FZRT_NOERR)
			{	err = fz_iost_one_long(iost,&my_command->value4);

				if(*version >= 1)
```

```
                        {       err = fz_iost_one_long(iost,&my_command-
>value5);
                        }
                }
        }
    }

    return(err);
}
```

The project data IO function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_data_io (
    long                            windex,
    fz_iost_ptr                     iost,
    fz_iost_dir_td_enum                     dir,
    fzpl_vers_td * const            version,
    unsigned long                   size
    );
```

form•Z calls this function to read and write any command specific project data to a **form•Z** project file. This function is called once when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** project file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_cmnd_proj_data_io (
    long                            windex,
    fz_iost_ptr                     iost,
    fz_iost_dir_td_enum                     dir,
    fzpl_vers_td * const            version,
    unsigned long                   size
    )
{
    fzrt_error_td       err = FZRT_NOERR;

    if ( dir == FZ_IOST_WRITE ) *version = 1;

    err = fz_iost_one_long(iost,&my_command->value1);
    if(err == FZRT_NOERR)
    {       err = fz_iost_one_long(iost,&my_command->value2);
```

```
            if(err == FZRT_NOERR)
            {       err = fz_iost_one_long(iost,&my_command->value3);
                    if(err == FZRT_NOERR)
                    {       err = fz_iost_one_long(iost,&my_command->value4);

                            if(*version >= 1)
                            {       err = fz_iost_one_long(iost,
                                                    &my_command->value5);
                            }
                    }
            }
        }

        return(err);
}
```

The project window data IO function (optional)

```
fzrt_error_td fz_cmnd_cbak_proj_wind_data_io (
        long                            windex,
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum                     dir,
        fzpl_vers_td * const            version,
        unsigned long                   size
        );
```

form•Z calls this function to read and write any command specific project window data to a **form•Z** project file. This function is called once for each window in the project when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** Project file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_cmnd_proj_wind_data_io (
        long                            windex,
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum                     dir,
        fzpl_vers_td * const            version,
        unsigned long                   size
        )
{
        fzrt_error_td       err = FZRT_NOERR;
```

```
	if ( dir == FZ_IOST_WRITE ) *version = 1;

	err = fz_iost_one_long(iost,&my_command->value1);
	if(err == FZRT_NOERR)
	{	err = fz_iost_one_long(iost,&my_command->value2);
		if(err == FZRT_NOERR)
		{	err = fz_iost_one_long(iost,&my_command->value3);
			if(err == FZRT_NOERR)
			{	err = fz_iost_one_long(iost,&my_command->value4);

				if(*version >= 1)
				{	err = fz_iost_one_long(iost,
							&my_command->value5);
				}
			}
		}
	}

	return(err);
}
```

2.8.3 File Translator

form•Z provides two interfaces for file translator plugins. The first is the **form•Z** structured translator interface. This is the interface used by all auto•des•sys authored translators and is the recommended interface to use for most file translator plugins. This method offers automatic access to the standard **form•Z** file import and export option dialogs and features (see 3.13 and 3.14 of the **form•Z** users Manual) and export and import pipelines. A structured translator is also able to take advantage of a lot of functionality that is provided for the translator which makes development faster, minimized maintenance and improves future compatibility. Structured file translators are not available to scripts.

An unstructured translator does not have access to the standard features and export and import pipelines and only supports data import and export. These translators have a simple function for specifying an options interface and a function for performing the export or import. Unstructured plugins are best used when dealing with a data file format that is very specific in nature and does not have support for the full range of data types that **form•Z** supports. An example of a good use of an unstructured file translator is a point cloud importer. Unstructured file translators are not available to scripts.

File translators are divided into image and data categories. The image category is for formats that represent 2d graphic information. The information can be in a bitmap or pixel format like TIFF and Targa or it can be vector like eps and HPGL. The data category are formats that contain 2D and/or 3D data which can be converted to or from **form•Z** modeling or drafting data types. This may include explicit object data or parametric data. Note that some formats may be considered a hybrid between image and data categories. This is often true of image formats that contain vector information. For example the Illustrator format is a vector based image format which is useful for storing rendered hidden line images (exported). It can also be used as a data import format to bring graphics into **form•Z**.

form•Z determines the interface location for the file formats based on the supported functionality implemented in the translator.

• Image formats that have export (vector or bitmap) functionality are added to the Export Image menu in the file menu.
• Image formats that have bitmap import functionality are available throughout the program where bitmap images are supported. This includes the underlay image, the displacement tool, and RenderZone surface style and options. Bitmap image formats can also be imported as drafting image elements from the Import command in the File menu.
• Image formats that have vector import functionality are available as the underlay image and can also be imported as drafting image elements from the Import command in the File menu.
• Data formats that have export functionality are added to the Export menu in the file menu.
• Data formats that have import functionality are available from the Import command in the File menu.

Translator information function set

Each file translator plugin (structured or unstructured) needs to provide a translator information function set (`fz_ffmt_cbak_info_fset`). This tells **form•Z** certain information about the plugin so that the translator can be added to the **form•Z** Interface.

File translator identification

Each file translator must be uniquely identified by a UUID. For efficiency, **form•Z** maps this UUID to an integer run time id called the translator's reference id. The reference id is passed to nearly all file translator functions. The reference id can be used to query **form•Z** for information about a translator and its file format. The reference id is constant for any given run of **form•Z**, but it may be different each time **form•Z** is run. Therefore, the reference id should not be used as a persistent identifier of the translator. Use the file tanslator's UUID as a persistent identifier. The file translator's reference id is of type `fz_ffmt_ref_td`.

File translators can also be identified by a keyword. Keywords are strings that can be used to obtain reference id for a translator for a specific file format without having to know the UUID of a specific file translator. For example, if a plugin needs to read a TIFF image, it can query **form•Z** for a TIFF translator. If one exists, **from•Z** will give it the file translator's reference id. Any translator that sets its keyword to `"TIFF"` is accessible to any plugin that needs to use a TIFF translator. Keywords do not have to be unique among translators, however they should be unique among file formats and all translators that support the same file format should use the same keyword. The **form•Z** defined keywords are defined in formZ_vendor.h. A translator can define its own keyword by publishing it in a C include file.

File translator options

Each file translator can choose to display an options dialog or not. This is the dialog that is invoked when the user clicks on the "Options..." button in the import or export standard file Open dialog (this dialog also invoked when the OK button in the import or export standard file Open dialog is clicked when the "Always Open File Format Options Dialogs" preference is set). File translator options dialogs are discussed in sections 3.13 and 3.14 of the **form•Z** Users Manual.

There are separate options dialogs for image import, image export, model import, model export, draft import and draft export. Each of these dialogs contains two major sections, the common options section and the custom options section. The common options section contains options that are common to all translators of a type (image import, model export, etc). Flags can be set to enable each item on the common options section. Only items for options the file translator supports should be enabled. The translator can also change the default value of any of the options in the common section. This would be necessary if the **form•Z** default value is not supported by the translator.

If a file translator needs to specify options that are not in the common section, it can define custom options. A file translator must do several things to define custom options. It must define the data structure that defines the options and must allocate the memory for storing the values of the options. It must provide a function to set the default values of the custom options. It must provide a function to save and load the options. And finally, it must provide a function to implement the custom section of the options dialog. This function uses the **form•Z** user interface manager function set to add items to the custom section of the options dialog.

More details on this are in the sections for each translator type.

File format Identification

On the Macintosh, **form•Z** uses the file type, the file extension and an optional translator provided file validation function to filter files listed in the file open dialog and to select an appropriate file translator to import the file. On Windows, only the file extension and the file validation function are used.

File translator information callback function set

Each file translator plugin (structured or unstructured) needs to provide a file translator information function set, `fz_ffmt_cbak_info_fset`. This tells **form•Z** certain information about the translator so that it can be added to the **form•Z** interface and associated with the file format it supports.

This function set contains the following functions:

The translator name function (required)

```
fzrt_error_td  fz_ffmt_cbak_name (
            char                *fmt_name,
            long                fmt_name_max_len
            );
```

This function is called by **form•Z** to get a name for the file format. This is the name that will appear in the **form•Z** interface. It is recommended that the format name and sort strings are stored in a .fzr file so that they are localizable.

The `fmt_name` string is the name of the format that will displayed in the **form•Z** interface. `ffmt_name_max_len` is the maximum length for this string.

An example of a translator name function is shown below.

```
fzrt_error_td  my_name(
             char          *fmt_name,
             long          fmt_name_max_len )
{
       fzrt_error_td              err = FZRT_NOERR;

       strncpy(fmt_name, "MY FORMAT NAME", fmt_name_max_len);

       return(err);
}
```

The translator uuid function (required)

```
fzrt_error_td  fz_ffmt_cbak_uuid (
             fzrt_UUID_td        fmt_uuid
             );
```

The file translator's `fmt_id` is a unique identifier for the translator. This can be the same as the plugin's id if the plugin only implements one translator (in other words, only adds one translator information function set to the plugin).

An example of a translator id function is shown below.

```
#define MY_PLUGIN_UUID       "\xfc\x98\x6f\x83\xf2\xd6\x4b\x9c\xb1\xc4\x0\x32\xf\x96\x8a\xfc"

fzrt_error_td  my_uuid(
             fzrt_UUID_td  fmt_uuid )
{
       fzrt_error_td              err = FZRT_NOERR;

       fzrt_UUID_copy(MY_PLUGIN_UUID, fmt_id);

       return(err);
}
```

The translator information function (required)

```
fzrt_error_td  fz_ffmt_cbak_info (
             char                *file_ext,
             long                file_ext_max_len,
             char                *sortby,
             long                sortby_max_len,
             char                *keywd,
             long                keywd_max_len,
             long                *attr_flags
             );
```

This function is called by **form•Z** to get general information about a file translator. This information includes the file extension of the file format, a string for sorting the file format in **form•Z** lists and menus, the file format keyword, and flags describing data needs of the format and capabilities of the translator. It is recommended that the sort string is stored in a .fzr file so that it is localizable. The file extension and keyword strings must not be localized. This function must be implemented by all file translators.

The sortby string is used to alphabetically sort format names displayed in the **form•Z** interface. This string, not the format's name, is used to sort format name in lists and menus. sortby_max_len is the maximum length for this string.

The file_ext string identifies the file's file extension. file_ext_max_len is the maximum length of file_ext.

keywd is a keyword string for the file format supported by the translator. keywd_max_len is the maximum length of keywd.

attr_flags is a set of bit flags that describe the data needs and certain caveats about the translator or file format. Appropriate bits for these flags are defined in fz_ffmt_format_attr_flags_enum. For example a bitmap format that stores depth data needs access to a rendered image's depth buffer. This would be specified by setting the FZ_FFMT_FORMAT_ATTR_NEEDS_DEPTH_BIT bit of the flags. A translator that is capable of exporting procedural textures should set the FZ_FFMT_FORMAT_ATTR_SUPPORTS_PROCTXTR_BIT. This would prevent **form•Z** from displaying a warning to the user that procedural texture will be lost by exporting to the file.

An example of a translator information function is shown below.

```
#define MY_PLUGIN_KEYWD       "My Keyword"

fzrt_error_td  my_info(
             char          *sortby,
             long          sortby_max_len,
             char          *file_ext,
             long          file_ext_max_len,
             char          *keywd,
             long          keywd_max_len,
             long          *attr_flags )
{
       fzrt_error_td              err = FZRT_NOERR;

       strncpy(sortby, "my sort string", sortby_max_len);
       strncpy(file_ext, "ext", file_ext_max_len);
       strncpy(keywd, MY_PLUGIN_KEYWD, keywd_max_len);

       *attr_flags = 0;

       return(err);
}
```

The Macintosh file type function (optional)

```
fzrt_error_td  fz_ffmt_cbak_ftype (
             fzrt_ostype         *type,
             fzrt_ostype         *creator
             );
```

This function is called to get the Macintosh file type and creator associated with the file format. This function is needed only when a Macintosh file type and creator are defined for the file format. On the Macintosh, the file type and creator are assigned to an exported file. The default values for the file type and creator are '????' which represents an unknown file type and creator.

An example of a translator file type function is shown below.

```
fzrt_error_td  my_ftype (
             fzrt_ostype         *type,
             fzrt_ostype         *creator )
{
       *type = 'MYTP';
```

```
        *creator = 'MYCR';

        return(FZRT_NOERR);
}
```

The translator icon from resource function (optional)

```
fzrt_error_td  fz_ffmt_cbak_icon_rsrc (
               fzrt_icon_ptr       *icon
               );
```

This function is called by **form•Z** to get an `icon` that is associated with the file format. This function is needed when the `icon` is stored in the plugin file's resources. The `icon` can be obtained by calling `fzrt_rlib_load_icon` and passing it the resource id of the `icon`.

An example of a translator icon from resource function is shown below.

```
unsigned long my_plugin_runtime_id;
#define MY_ICON_NUM 10

fzrt_error_td  my_icon_rsrc(
               fzrt_icon_ptr       *icon )
{
        long                               err = FZRT_NOERR;
        short                              rlib_index;

        if(icon != NULL)
        {
               err = fzpl_plugin_get_rlib_idx(my_plugin_runtime_id, &rlib_index);
               if(err == FZRT_NOERR)
               {
                      err = fzrt_rlib_load_icon (
                             rlib_index,
                             FZRT_LOAD_ICON_COLOR,MY_ICON_NUM,
                             icon );
               }
        }

        return(err);
}
```

The translator icon from file function (optional)

```
fzrt_error_td  fz_ffmt_cbak_icon_file (
               fzrt_floc_ptr              floc,
               long                       *hpos,
               long                       *vpos,
               fzrt_floc_ptr              floc_mask,
               long                       *hpos_mask,
               long                       *vpos_mask
               );
```

This function is called by **form•Z** to get an icon that is associated with the file format. This function is needed when the icon is stored in a bitmap file. The icon file's format must be TIFF. Two files fully define an icon. The first is the icon's color bitmap that defines the color image of the icon. The second is the icon's mask bitmap that defines the transparent areas of the icon. The mask file is optional.

`floc` is the file location of the icon's color bitmap file. `hpos` and `vpos` are horizontal and vertical offsets (in pixels) into the color bitmap. This allows one image file to contain multiple icons.

`floc_mask` is the file location of the icon's mask file. `hpos_mask` and `vpos_mask` are horizontal and vertical offsets (in pixels) into the mask bitmap. This allows one image file to contain multiple icon masks.

An example of a translator icon from file function is shown below.

```
fzrt_error_td  my_icon_file(
                    fzrt_floc_ptr       floc,
                    long                *hpos,
                    long                *vpos,
                    fzrt_floc_ptr       floc_mask,
                    long                *hpos_mask,
                    long                *vpos_mask )
{
       fzrt_error_td       err;
       fzrt_floc_ptr       floc_local;

       fzrt_file_floc_init(&floc_local);

       err = _fset_glue->fzpl_plugin_file_get_floc (floc_local);
       if(err == FZRT_NOERR)
       {
              err = fzrt_file_floc_copy(floc_local, floc);
              if(err == FZRT_NOERR)
                  err = fzrt_file_floc_set_name(floc, "my_col.tif");
              *hpos = 0;
              *vpos = 0;
       }
       if(err == FZRT_NOERR)
       {
              err = fzrt_file_floc_copy(floc_local, floc_mask);
              if(err == FZRT_NOERR)
                  err = fzrt_file_floc_set_name(floc_mask, "my_mask.tif");
              *hpos_mask = 0;
              *vpos_mask = 0;
       }

       fzrt_file_floc_finit(&floc_local);

       return(err);
}
```

The file validation function (optional)

```
fzrt_boolean  fz_ffmt_cbak_is_file (
              fzrt_floc_ptr       floc,
              const fzrt_ptr      data,
              long                size
              );
```

This function is called to determine if a specific file can be read by a translator. This function is called only if the file extension and, on the Macintosh, the file type matches that specified by the translator. The file's `floc` and the first `size` bytes of the file (`data`) are passed into this function. This function determines if the file is of the supported format by inspecting the contents of the file. If the file format can be verified by inspecting the contents of the first `size` bytes of the file (i.e. check a header at the beginning of the file), then the data passed in should be used for verification. Otherwise, the file must be opened, read, and closed by this function. Checking the `data` passed in is recommended because it is more efficient.

An example of a translator file validation function is shown below.

```
#define MY_HEADER_DATA     0x05551212

fzrt_boolean my_is_file(
              fzrt_floc_ptr       floc,
              const fzrt_ptr      buffer,
              long                size )
{
       fzrt_boolean  is = FALSE;
```

```
	long		testval = MY_HEADER_DATA;

	if(memcmp(buffer, &testval, 4) == 0)	is = TRUE;

	return(is);
}
```

The translator custom options IO function (optional)

```
fzrt_error_td  fz_ffmt_cbak_opts_io (
		fz_iost_ptr		iost,
		fz_iost_dir_td_enum	dir,
		fzpl_vers_td * const	version,
		unsigned long		size
		);
```

This function is called by **form•Z** to save and load the translator's custom options. This function is only needed if the translator has custom options.

This function saves and loads the file translator's custom options via a **form•Z** IO Stream (`iost`). The `dir` parameter specifies whether the options are being read or written. A file translator's custom options are written as a block of data. This data starts with a header that contains the `version` and `size` of the block. If the options are being read, the `version` and block `size` are passed into this function. If the `size` or `version` are not an expected value, an error must be returned. **form•Z** will then skip that block of data. Otherwise, the data should be read and `FZRT_NOERR` returned. If the options are being written, the `version` is set by this function and data is written. If any of the IO Stream functions return an error, writing/reading should stop and the error returned.

An example of a translator custom options IO function is shown below.

```
#define MY_PLUGIN_VERSION		FZPL_VERS_MAKE(1,0,0,0)
long	my_read_opts_flags;
long	my_write_opts_flags;

fzrt_error_td  my_opts_io (
		fz_iost_ptr		iost,
		fz_iost_dir_td_enum	dir,
		fzpl_vers_td * const		version,
		unsigned long		size )
{
	fzrt_error_td		err = FZRT_NOERR;

	if(dir == FZ_IOST_WRITE)	*version = MY_PLUGIN_VERSION;
	else
	{
		if(*version > MY_PLUGIN_VERSION || size < (2*sizeof(long)))
		{
			err = fzrt_error_set(
					FZPL_BAD_VERSION_ERROR,
					FZRT_ERROR_SEVERITY_WARNING,
					FZRT_ERROR_CONTEXT_FZRT, FZPL_CONTEXT_ID );
		}
	}

	if(err == FZRT_NOERR)
	{
		err = fz_iost_long(iost, &my_read_opts_flags, 1);
		err = fz_iost_long(iost, &my_write_opts_flags, 1);
	}

	return(err);
}
```

Structured file translators

Structured image file translators

Image file translators read and/or write bitmap data, vector data or both. What functions an image translator performs (read or write) and which data formats (bitmap or vector) a translator supports are determined by which functions in the `fz_ffmt_cbak_image_fset` are implemented by the translator. Callback functions in the `fz_ffmt_cbak_image_fset` function set that begin with `fz_ffmt_cbak_image_bmap_read` are for reading bitmap images. Callback functions in the `fz_ffmt_cbak_image_fset` function set that begin with `ffmt_cbak_image_bmap_write` are for writing bitmap images. Callback functions in the `fz_ffmt_cbak_image_fset` function set that begin with `fz_ffmt_cbak_image_vect_read` are for reading vector images. Callback functions in the `fz_ffmt_cbak_image_fset` function set that begin with `ffmt_cbak_image_vect_write` are for writing vector images. Callback functions in the `fz_ffmt_cbak_image_fset` function set that begin with `fz_ffmt_cbak_image_read` are functions common to reading bitmap and vector images. Callback functions in the `fz_ffmt_cbak_image_fset` function set that begin with `ffmt_cbak_image_write` are functions common to writing bitmap and vector images.

Two function sets are needed for an image file translator, the translator information function set and the image translator function set. The translator information function set is identified by the following constants, `FZ_FILE_IMAGE_EXTS_TYPE` (plugin type UUID), `FZ_FILE_IMAGE_EXTS_NAME` (plugin type name), and `FZ_FILE_IMAGE_EXTS_VERSION` (plugin type version).

The example below shows the definition of an image file translator and the registration of the two function sets within that plugin.

```
#define MY_STRINGS              1
#define MY_NAME_STR             1
#define MY_PLUGIN_UUID          "\xfc\x98\x6f\x83\xf2\xd6\x4b\x9c\xb1\xc4\x0\x32\xf\x96\x8a\xfc"
#define MY_PLUGIN_VERSION       FZPL_VERS_MAKE(1,0,0,0)
#define MY_PLUGIN_VENDOR        "My Company Name"
#define MY_PLUGIN_URL           "www.myurl.com"

static fzrt_error_td  my_image_register_plugin ()
{
	fzrt_error_td       err = FZRT_NOERR;
	long                num_failed = 0;
	char                pname[FZPL_NAME_SIZE];

	/* Register the plugin */

	err = fzrt_fzr_get_string (
			_fz_rsrc_ref,
			MY_STRINGS,
			MY_NAME_STR,
			pname );
	if ( err == FZRT_NOERR )
	{
		err = fset_glue->fzpl_plugin_register (
				MY_PLUGIN_UUID,
				pname,
				MY_PLUGIN_VERSION,
				MY_PLUGIN_VENDOR,
				MY_PLUGIN_URL,
				FZ_FILE_IMAGE_EXTS_TYPE,
				FZ_FILE_IMAGE_EXTS_VERSION,
				NULL,
				0,
				NULL,
				&my_plugin_runtime_id );
	}
	if ( err == FZRT_NOERR )
	{
		/* Add the function sets implemented by the plugin */
```

```
		err = fset_glue->fzpl_plugin_add_fset (
					my_plugin_runtime_id,
					FZ_FFMT_CBAK_INFO_FSET_TYPE,
					FZ_FFMT_CBAK_INFO_FSET_VERSION,
					FZ_FFMT_CBAK_INFO_FSET_NAME,
					FZPL_TYPE_STRING(fz_ffmt_cbak_info_fset),
					sizeof ( fz_ffmt_cbak_info_fset ),
					my_fill_translator_info_fset,
					FALSE);

		if(err == FZRT_NOERR)
		{
			err = fset_glue->fzpl_plugin_add_fset (
						my_plugin_runtime_id,
						FZ_FFMT_CBAK_IMAGE_FSET_TYPE,
						FZ_FFMT_CBAK_IMAGE_FSET_VERSION,
						FZ_FFMT_CBAK_IMAGE_FSET_NAME,
						FZPL_TYPE_STRING(fz_ffmt_cbak_image_fset),
						sizeof ( fz_ffmt_cbak_image_fset ),
						my_fill_image_cbak_fset,
						FALSE);
		}
	}

	return(err);
}
```

The example below shows the function set fill functions for the `fz_ffmt_cbak_info_fset` and the `fz_ffmt_cbak_image_fset` function sets.

```
fzrt_error_td  my_translator_info_get_fset (
		const fzpl_fset_def_ptr	fset_def,
		fzpl_fset_td * const		fset )
{
	fzrt_error_td				err = FZRT_NOERR;
	fz_ffmt_cbak_info_fset		*info_funcs;

	err = fzpl_glue->fzpl_fset_def_check ( fset_def,
				FZ_FFMT_CBAK_INFO_FSET_VERSION,
				FZPL_TYPE_STRING(fz_ffmt_cbak_info_fset),
				sizeof(fz_ffmt_cbak_info_fset),
				FZPL_VERSION_OP_NEWER );
	if ( err == FZRT_NOERR )
	{
		info_funcs = (fz_ffmt_cbak_info_fset *)fset;

		info_funcs->fz_ffmt_cbak_name			= my_name;
		info_funcs->fz_ffmt_cbak_uuid			= my_uuid;
		info_funcs->fz_ffmt_cbak_info			= my_info;
		info_funcs->fz_ffmt_cbak_ftype			= my_ftype;
		info_funcs->fz_ffmt_cbak_icon_rsrc		= my_icon_rsrc;
		info_funcs->fz_ffmt_cbak_icon_file		= my_icon_file;
		info_funcs->fz_ffmt_cbak_opts_io		= my_opts_io;
		info_funcs->fz_ffmt_cbak_is_file		= my_is_file;
	}

	return(err);
}

fzrt_error_td  my_get_image_cbak_fset (
		const fzpl_fset_def_ptr	fset_def,
		fzpl_fset_td * const		fset )
{
	fzrt_error_td				err = FZRT_NOERR;
	fz_ffmt_cbak_image_fset		*image_funcs;

	err = _fset_glue->fzpl_fset_def_check ( fset_def,
```

```
                    FZ_FFMT_CBAK_IMAGE_FSET_VERSION,
                    FZPL_TYPE_STRING(fz_ffmt_cbak_image_fset),
                    sizeof ( fz_ffmt_cbak_image_fset ),
                    FZPL_VERSION_OP_NEWER );
    if ( err == FZRT_NOERR )
    {
        image_funcs = (fz_ffmt_cbak_image_fset *)fset;

        image_funcs->fz_ffmt_cbak_image_read_dlog_cust         = my_read_dlog_cust;
        image_funcs->fz_ffmt_cbak_image_read_predloginit       = my_read_predloginit;
        image_funcs->fz_ffmt_cbak_image_read_opts_default      = my_read_opts_default;
        image_funcs->fz_ffmt_cbak_image_read_opts_flags        = my_read_opts_get_flags;
        image_funcs->fz_ffmt_cbak_image_read_opts_changed      = my_read_opts_changed;

        image_funcs->fz_ffmt_cbak_image_bmap_read_info         = my_bmap_read_info;
        image_funcs->fz_ffmt_cbak_image_bmap_read              = my_bmap_read;

        image_funcs->fz_ffmt_cbak_image_vect_read_frame        = my_vect_read_frame;
        image_funcs->fz_ffmt_cbak_image_vect_read              = my_vect_read;
        image_funcs->fz_ffmt_cbak_image_vect_read_raw_data     = my_vect_read_raw;
        image_funcs->fz_ffmt_cbak_image_vect_read_printer_data = my_vect_read_printer;
        image_funcs->fz_ffmt_cbak_image_read_plat_native_draft_image
                                                     = my_read_platform_data;

        image_funcs->fz_ffmt_cbak_image_write_dlog_cust        = my_write_dlog_cust;
        image_funcs->fz_ffmt_cbak_image_write_predloginit      = my_write_predloginit;
        image_funcs->fz_ffmt_cbak_image_write_opts_default     = my_write_opts_default;
        image_funcs->fz_ffmt_cbak_image_write_opts_flags       = my_write_opts_get_flags;
        image_funcs->fz_ffmt_cbak_image_write_opts_changed     = my_write_opts_changed;

        image_funcs->fz_ffmt_cbak_image_bmap_write_file_begin  = my_bmap_write_file_begin;
        image_funcs->fz_ffmt_cbak_image_bmap_write_file_end    = my_bmap_write_file_end;
        image_funcs->fz_ffmt_cbak_image_bmap_write_image_begin = my_bmap_write_image_begin;
        image_funcs->fz_ffmt_cbak_image_bmap_write_image_scanline_byte =
                                                          my_bmap_write_image_scanline;
        image_funcs->fz_ffmt_cbak_image_bmap_write_image_end   = my_bmap_write_image_end;
        image_funcs->fz_ffmt_cbak_image_bmap_write_progress_str = my_bmap_write_progress_str;
        image_funcs->fz_ffmt_cbak_image_bmap_write_err_label   = my_bmap_write_err_label;

        image_funcs->fz_ffmt_cbak_image_vect_write_begin       =
                                                     my_vect_write_begin;
        image_funcs->fz_ffmt_cbak_image_vect_write_end         =
                                                     my_vect_write_end;
        image_funcs->fz_ffmt_cbak_image_vect_write_progress_str     =
                                                     my_vect_write_progress_str;
        image_funcs->fz_ffmt_cbak_image_vect_write_err_label   =
                                                     my_vect_write_err_label;
        image_funcs->fz_ffmt_cbak_image_vect_write_line        =
                                                     my_vect_write_line;
        image_funcs->fz_ffmt_cbak_image_vect_write_point       =
                                                     my_vect_write_point;
        image_funcs->fz_ffmt_cbak_image_vect_write_simple_text =
                                                     my_vect_write_simple_text;
        image_funcs->fz_ffmt_cbak_image_vect_write_set_line_color   =
                                                     my_vect_write_set_line_color;
        image_funcs->fz_ffmt_cbak_image_vect_write_set_fill_color   =
                                                     my_vect_write_set_fill_color;
        image_funcs->fz_ffmt_cbak_image_vect_write_set_line_weight  =
                                                     my_vect_write_set_line_weight;
        image_funcs->fz_ffmt_cbak_image_vect_write_lineset     =
                                                     my_vect_write_lineset;
        image_funcs->fz_ffmt_cbak_image_vect_write_begin_compound   =
                                                     my_vect_write_begin_compound;
        image_funcs->fz_ffmt_cbak_image_vect_write_end_compound     =
                                                     my_vect_write_end_compound;
        image_funcs->fz_ffmt_cbak_image_vect_write_save_gstate =
                                                     my_vect_write_save_gstate;
        image_funcs->fz_ffmt_cbak_image_vect_write_restore_gstate   =
```

```
                                                             my_vect_write_restore_gstate;
            image_funcs->fz_ffmt_cbak_image_vect_write_set_line_style   =
                                                             my_vect_write_set_line_style;
            image_funcs->fz_ffmt_cbak_image_vect_write_can_do_arc  =
                                                             my_vect_write_can_do_arc;
            image_funcs->fz_ffmt_cbak_image_vect_write_arc = my_vect_write_arc;
            image_funcs->fz_ffmt_cbak_image_vect_write_set_text_angle   =
                                                             my_vect_write_set_text_angle;
            image_funcs->fz_ffmt_cbak_image_vect_write_set_text_font    =
                                                             my_vect_write_write_text_font;
            image_funcs->fz_ffmt_cbak_image_vect_write_write_text_char    =
                                                             my_vect_write_text_char;
            image_funcs-> fz_ffmt_cbak_image_vect_write_set_fill_pattern    =
                                                             my_vect_write_fill_pattern;
            image_funcs->fz_ffmt_cbak_image_vect_write_string_header    =
                                                             my_vect_write_string_header;
            image_funcs->fz_ffmt_cbak_image_vect_write_string_newline   =
                                                             my_vect_write_string_newline;
            image_funcs->fz_ffmt_cbak_image_vect_write_string_font =
                                                             my_vect_write_string_font;
            image_funcs->fz_ffmt_cbak_image_vect_write_string_write     =
                                                             my_vect_write_string_write;
            image_funcs->fz_ffmt_cbak_image_vect_write_string_trailer   =
                                                             my_vect_write_string_trailer;
      }

      return(err);
}
```

Import options

Image import translators can display an import options dialog. The "Options..." button on the import standard file Open dialog will be enabled if the options flags set by the `fz_ffmt_cbak_image_read_opts_flags` function has the `FZ_FFMT_OPTS_INIT_HAS_READ_OPTS_BIT` bit set. Individual items in the common section of the options dialog are enabled by setting the appropriate bits of the flags parameter to `fz_ffmt_cbak_image_read_opts_flags`. The appropriate bits are defined in `fz_ffmt_image_read_iface_opts_flags_enum`. Image import options are discussed in section 3.14.1 of the **form•Z** Users Manual.

The call back functions to import an image are defined in the `fz_cbak_ffmt_image_fset`.

The `fz_ffmt_cbak_image_fset` contains the following functions to support image import options:

The image translator import options dialog enable function (required)

```
fzrt_error_td  fz_ffmt_cbak_image_read_opts_flags (
             fz_ffmt_ref_td      ffmt_id,
             long                *flags,
             long                *opts_flags
             );
```

This function is called by **form•Z** to get the enable state of the image import "Options..." button and the enable states of each item in the image import options dialog. The `flags` parameter is used to set the enable states of items in the common section of the options dialog. Appropriate bits for this parameter are defined in `fz_ffmt_image_read_iface_opts_flags_enum`. By default, all items are disabled. The `opts_flags` parameter enables the "Options..."button on the "Image Import" standard file Open dialog by setting it to `FZ_FFMT_OPTS_INIT_HAS_READ_OPTS_BIT`. If the "Options..." button is to be disabled, `opts_flags` should be set to `0` (this is the default).

An example of an image translator import options dialog enable function is shown below.

```
fzrt_error_td  my_read_opts_get_flags (
```

```
             fz_ffmt_ref_td      ffmt_id,
             long                *flags,
             long                *opts_flags )
{
       fzrt_error_td       err = FZRT_NOERR;

       FZ_SETBIT(*flags, FZ_FFMT_IMAGE_READ_IFACE_OPTS_ENABLEIMPORTASDRAFTBITMAP_BIT);
       FZ_SETBIT(*flags, FZ_FFMT_IMAGE_READ_IFACE_OPTS_ENABLEORIGINALSIZE_BIT);
       FZ_SETBIT(*flags, FZ_FFMT_IMAGE_READ_IFACE_OPTS_ENABLEPROPORTIONS_BIT);
       FZ_SETBIT(*flags, FZ_FFMT_IMAGE_READ_IFACE_OPTS_ENABLESTOREINPROJECT_BIT);

       FZ_SETBIT(*opts_flags , FZ_FFMT_OPTS_INIT_HAS_READ_OPTS_BIT);

       return(err);
}
```

The image translator import options defaults function (optional)

```
fzrt_error_td  fz_ffmt_cbak_image_read_opts_default (
             fz_ffmt_ref_td             ffmt_id
             );
```

This function is called by **form•Z** to set default values of options. All custom options and any common options whose default values the file translator wishes to change must be set here. This function is only needed if the translator has custom options or the translator needs to change default values of any of the common options.

form•Z will have set the default values for common options prior to calling this function. This function can then change any of those values by calling `fz_ffmt_image_read_opts_parm_set` using the options pointer obtained from `fz_ffmt_image_read_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_image_read_opts_parm_get`.

An example of an image translator import options defaults function is shown below.

```
long   my_read_opts_flags;

fzrt_error_td  my_read_opts_default (
                    fz_ffmt_ref_td      ffmt_id )
{
       fzrt_error_td              err = FZRT_NOERR;
       fz_type_td                 fz_type;
       fz_xy_td                   size = {24.0, 24.0};

       /* Change a default value in the common options */
       fz_type_set_xy(&size, &fz_type);
       fz_ffmt_image_read_opts_parm_set (
                           ffmt_id,
                           FZ_FFMT_IMAGE_READ_OPTS_PARM_SIZE,
                           &fz_type );
       /* Set a default value for a custom option */
       my_read_opts_flags = 0;

       return(err);
}
```

This example changes the image value options to 2 feet (the **form•Z** default is 1 foot) and initializes a custom option.

The image translator import options changed function (optional)

```
fzrt_error_td  fz_ffmt_cbak_image_read_opts_changed (
             fz_ffmt_ref_td                  ffmt_id,
             fz_ffmt_image_read_opts_enum     which
             );
```

This function is called by **form•Z** when the user changes the value of an option in the common section of the options dialog. This allows the translator to override the behavior of the common section of the options dialog by setting values of options or setting enable states of items. The `which` parameter specifies which parameter's value changed. Values of the common options can be set by calling `fz_ffmt_image_read_opts_parm_set` using the options pointer obtained from `fz_ffmt_image_read_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_image_read_opts_parm_get`. The enable states of items can be changed by first getting the enable flags by calling `fz_ffmt_image_read_get_dlog_flags`, then changing the enable bit of the item whose state needs to change and calling `fz_ffmt_image_read_set_dlog_flags`. Appropriate bits are defined in `fz_ffmt_image_read_iface_opts_flags_enum`. All these functions are in the `fz_ffmt_image_fset` function set.

An example of an image translator import options changed function is shown below.

```
fzrt_error_td  my_read_opts_changed (
             fz_ffmt_ref_td                     ffmt_id,
             fz_ffmt_image_read_opts_enum       which
             )
{
       fzrt_error_td                      err = FZRT_NOERR;
       fz_type_td                         fz_type;
       fz_xy_td                           size = {24.0, 24.0};

       if(which == FZ_FFMT_IMAGE_READ_OPTS_PARM_SIZE)
       {
             fz_ffmt_image_read_opts_parm_get (
                                   ffmt_id,
                                   FZ_FFMT_IMAGE_READ_OPTS_PARM_SIZE,
                                   &fz_type );
             fz_type_get_xy(&fz_type, &size);

             size.x = (round(size.x/12.0)*12);
             size.y = (round(size.y/12.0)*12);

             fz_type_set_xy(&size, &fz_type);
             fz_ffmt_image_read_opts_parm_set (
                                   ffmt_id,
                                   FZ_FFMT_IMAGE_READ_OPTS_PARM_SIZE,
                                   &fz_type );
       }

       return(err);
}
```

This example rounds the image world size that the user entered to the nearest foot.

The image translator import custom options dialog function (optional)

```
fzrt_error_td  fz_ffmt_cbak_image_read_dlog_cust (
             fz_fuim_tmpl_ptr    fuim_tmpl,
             short               parent,
             fz_ffmt_ref_td      ffmt_id
             );
```

This function is called by **form•Z** to add items to the custom section of the options dialog. This function should add items by calling functions in the `fz_fuim_fset` function set using `fuim_mngr` and `parent` parameter (passed into this function) as the top level parent for all items. The `ffmt_id` parameter specifies the file translator's reference id.

An example of an image translator import custom options dialog function is shown below.

```
#define MY_STRINGS           1
#define MY_COMPRESS_STR      2

#define MY_COMPRESS_BIT      1
long   my_read_opts_flags;

fzrt_error_td  my_read_dlog_cust (
             fz_fuim_tmpl_ptr    fuim_tmpl,
             short               parent,
             fz_ffmt_ref_td      ffmt_id )
{
       fzrt_error_td       err = FZRT_NOERR;
       short               gindx;
       char                title[256];

       err = fzrt_fzr_get_string (
                    _fz_rsrc_ref,
                    MY_STRINGS,
                    MY_COMPRESS_STR,
                    title );

       if(err == FZRT_NOERR)
       {
             if((gindx = fz_fuim_new_check(fuim_tmpl, parent, 0,
                          FZ_FUIM_FLAG_GFLT | FZ_FUIM_FLAG_HORZ, title, NULL, NULL)) > -1)
             {
                    fz_fuim_item_encod_long(fuim_tmpl, gindx, & my_read_opts_flags,
                                        TRUE, FZ_FUIM_BIT2_MASK(MY_COMPRESS_BIT));
             }
       }

       return(err);
}
```

The image translator import pre-options dialog function (optional)

```
fzrt_error_td  fz_ffmt_cbak_image_read_predloginit (
             long                windex,
             fz_ffmt_ref_td      ffmt_id,
             );
```

This function is called by **form•Z** just prior to displaying the options dialog. This is done so the file translator can check the current state of **form•Z** and make any adjustments to the values of options or the enable states of items on the options dialog. For example, a translator may have a custom option that is only appropriate if the rendering is vector and not appropriate for a bitmap rendering. In this case, this function would check the type of the current rendering and disable the option's dialog item if the render type is bitmap. Options values and item enable states can be changed as described in for the `fz_ffmt_cbak_image_read_opts_changed` function.

Structured image bitmap file import

Bitmap images are imported into **form•Z** in several ways, import to draft, view file, texture map and underlay image. Each of these are accomplished by simply reading the image pixels into a pixel buffer (`fzrt_pbuf_ptr`) and **form•Z** does the rest.

The `fz_ffmt_cbak_image_fset` contains the following functions to support importing image bitmap files:

The image bitmap translator import file information function (required for bitmap import)

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_read_info (
             long                windex,
             fz_ffmt_ref_td      ffmt_id,
             fzrt_floc_ptr       floc,
             long                *width,
```

```
            long                *height,
            short               *pixel_depth,
            fzrt_boolean        *has_alpha,
            double              *resolution
            );
```

This function is called by **form•Z** to get information about a specific bitmap image file. This information includes the `width` and `height` of the image (in pixels), `pixel_depth` (the number of bits per pixel, i.e. 8 for grayscale, 24 for RGB), `has_alpha` (whether or not the image has an alpha (transparency) channel), and the `resolution` of the image (in pixels per inch). If the resolution of the image is not known (not stored in the file format), use 72.0.

An example of an image bitmap translator import file information function is shown below.

```
fzrt_error_td  my_bmap_read_info (
            long                windex,
            fz_ffmt_ref_td      ffmt_id,
            fzrt_floc_ptr       floc,
            long                *width,
            long                *height,
            short               *pixel_depth,
            fzrt_boolean        *has_alpha,
            double              *resolution )
{
      fzrt_error_td       err = FZRT_NOERR;
      short               num_color_channels;
      my_file_td          file;

      err = my_file_open(floc, &file);
      if(err == FZRT_NOERR)
      {
            err = my_read_header_data(&file, &width, &height, &num_color_channels);
            if(err = FZRT_NOERR)
            {
                  if(num_color_channels = 1)
                  {
                        *pixel_depth = 8;
                        *has_alpha = FALSE;
                  }
                  if(num_color_channels = 3)
                  {
                        *pixel_depth = 24;
                        *has_alpha = FALSE;
                  }
                  if(num_color_channels = 4)
                  {
                        *pixel_depth = 32;
                        *has_alpha = TRUE;
                  }
                  *resolution = 72.0;
            }
            if (err == FZRT_NOERR)
                  err = my_file_close(&file);
      }

      return(err);
}
```

The image bitmap translator import pixels function (required for bitmap import)

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_read (
            long                          windex,
            fz_ffmt_ref_td                ffmt_id,
            fzrt_floc_ptr                 floc,
            fz_ffmt_image_channel_enum    which_channel,
```

```
            fzrt_pbuf_ptr                    *pbuf
            );
```

This function is called by **form•Z** to read the pixels of a bitmap image. The pixels are read into the pixel buffer passed into this function. To avoid two memory buffers for all the pixels (one for the pixel buffer and a working buffer for the translator), bitmap images are read a single scanline at a time. This way the translator only needs to allocate a buffer for one scanline. **form•Z** provides some functions in the fz_ffmt_image_fset function set which are used to assemble the scanlines, and put the pixels into a form that can be drawn to a window. Reading pixels from a bitmap image is as follows.

Initialize an image scanline data pointer
loop until all of image read
    read a scanline
    add the scanlines to the image scanline data
Uninitialize the image scanline data pointer

To initialize an image scanline data pointer, call `ffmt_image_scanline_data_init` in the `fz_ffmt_image_fset` function set. This function takes several parameters that describe the image pixels. The first parameter is the pixel buffer to read he image into. The last parameter is a pointer to an image scanline data pointer to initialize. The middle parameters describe the image pixels as follows:

`img_width` - the number of pixels in a single scanline.
`bits_per_pixel` - the number of bit for each pixel. Valid values are 8, 16, 24, and 32.
`bytes_per_pixel` - the number of bytes per pixel.
`channels` - the number of color channels per pixel.
`which_channel` - which channel(s) to add to the pixel buffer. This must be the which_channel parameter passed into this function.
`rgb_order` - the order of the color channels (RGB or BGR)
`alpha_first` - if TRUE, the aplha channel precedes the rgb channels.
`img_type` - True color or colormapped.
`color_map` - If the img_type is colormapped, this must point to a colormap.
`map_entry_bits` - bits used by each entry in the colormap
`map_entry_bytes` - bytes used by each entry in the colormap.
`map_n_entries` - number of entries in the color map.
`map_order` - RGB-RGB-RGB or RRR-GGG-BBB
`map_type` - the type of colormap needed by the image.
`gamma` - a gamma value to apply to the image.

Once the image scanline data pointer is initialized each scanline read is added to the pixel buffer by calling `ffmt_image_scanline_data_import` in the `fz_ffmt_image_fset` function set. The first parameter is the pixel buffer to write the pixels to, the second parameter is the initialized image scanline data pointer, the third parameter is a pointer to the scanline pixel data, the fourth parameter is the scanline row number, The fifth parameter is a pixel offset into the scanline (some images may sufficiently large that only a part of a scanline can be read. This parameter allows for that. The last parameter is the number of pixels added to the image.
Once the whole image is read, `ffmt_image_scanline_data_finit` in the `fz_ffmt_image_fset` function set is called to uninitialize the image scanline data pointer.

An example of an image bitmap translator import file information function is shown below.

```
fzrt_error_td  my_bmap_read (
            long                             windex,
            fz_ffmt_ref_td                   ffmt_id,
            fzrt_floc_ptr                    floc,
            fz_ffmt_image_channel_enum       which_channel,
            fzrt_pbuf_ptr                    *pbuf )
{
      fzrt_error_td              err = FZRT_NOERR;
      short                      num_color_channels;
      my_file_td                 file;
      long                       width, height, y;
```

```
	short						pixel_depth, bytes_per_pixel;
	fz_ffmt_image_import_ptr	img_in_ptr;
	fzrt_ptr					pixels;

	err = my_file_open(floc, &file);
	if(err == FZRT_NOERR)
	{
		err = my_read_header_data(&file, &width, &height, &num_color_channels);
		if(err = FZRT_NOERR)
		{
			if(num_color_channels = 1)
			{
				pixel_depth = 8;
				bytes_per_pixel = 1;
			}
			if(num_color_channels = 3)
			{
				pixel_depth = 24;
				bytes_per_pixel = 3;
			}
			if(num_color_channels = 4)
			{
				pixel_depth = 32;
				bytes_per_pixel = 4;
			}
			pixels = (fzrt_ptr)fzrt_new_ptr(bytes_per_pixel * width);
			if(pixels == NULL)
			{
				err = fzrt_error_set (
						FZRT_MALLOC_ERROR,
						FZRT_ERROR_SEVERITY_ERROR,
						FZRT_ERROR_CONTEXT_FZRT,  0 );

			}
			else
			{
				err = fz_ffmt_image_bmap_scanline_init (
							pbuf,
							width,
							pixel_depth,
							bytes_per_pixel,
							num_color_channels,
							which_channel,
							FZ_FFMT_IMAGE_ORDER_RGB,
							FALSE,
							FZ_FFMT_IMAGE_TYPE_TRUE_COLOR,
							NULL,
							0,
							0,
							0,
							0,
							FZ_FFMT_IMAGE_MAP_TYPE_GRAY_BLACK_FIRST,
							0.0,
							&img_in_ptr );
				for(y = 0; y < height && err == FZRT_NOERR; y++)
				{
					err = my_read_scanline(y, pixels);
					if(err == FZRT_NOERR)
						fz_ffmt_image_bmap_scanline_import (
									pbuf,
									img_in_ptr,
									pixels,
									(unsigned short)y,
									(unsigned short)0,
									(unsigned short)width );
				}
				fz_ffmt_image_bmap_scanline_finit (&img_in_ptr);
			}
```

```
        }
    }

    if (err == FZRT_NOERR)
        err = my_file_close(&file);

    return(err);
}
```

Structured image vector file import

Vector images are imported into **form•Z** in one of two ways, as objects in the modeling environment, or as draft elements in the drafting environment. To acomplish this, **form•Z** first calls the translators import frame function to get the extents of the vector data. Then **form•Z** calls the translator's read function. This function reads the contents of the file and constructs model objects using functions in `fz_model_fset` or draft elements using functions in `fz_draft_fset`.

The `fz_ffmt_cbak_image_fset` contains the following functions to support importing image vector files:

The image vector translator import frame function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_read_frame (
            long                windex,
            fz_ffmt_ref_td      ffmt_id,
            fzrt_floc_ptr       floc,
            long                size,
            fzrt_rect           *rect
            );
```

This function is called by **form•Z** to get the extents of the vector data contained in a file. `windex` is the active window and `ffmt_id` is the reference id of the file format. floc contains the file's name and path. `rect` is filled by this plugin with the extents of the vector data. **form•Z** will scale and offset this rect as dictated by the import options.

An example of an image vector translator import frame function is shown below.

```
fzrt_error_td  my_vect_read_frame (
            long                windex,
            fz_ffmt_ref_td      ffmt_id,
            fzrt_floc_ptr       floc,
            long                size,
            fzrt_rect           *rect )
{
    fzrt_error_td       err = FZRT_NOERR;
    my_file_td          file;
    long                top,left,bottom,right;

    err = my_file_open(floc, &file);
    if(err == FZRT_NOERR)
    {
        err = my_file_read_extents(&file, &left, &top, &right, &bottom);
        if(err == FZRT_NOERR)
        {
            rect->top = top;
            rect->left = left;
            rect->right = right;
            rect->bottom = bottom;
        }
        if (err == FZRT_NOERR)
            err = my_file_close(&file);
    }

    return(TRUE);
```

```
}
```

The image vector translator import function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_read (
             long                 windex,
             fz_ffmt_ref_td       ffmt_id,
             fzrt_floc_ptr        floc
             );
```

This function is called by **form•Z** to import the vector data in a file. This function creates model objects from the vector geometry by calling functions in the `fz_model_fset` function set.

A simple example of an image vector translator import function which imports circles is shown below.

```
fzrt_error_td  my_vect_read (
             long                 windex,
             fz_ffmt_ref_td       ffmt_id,
             fzrt_floc_ptr        floc )
{
       fzrt_error_td        err = FZRT_NOERR;
       my_file_td           file;
       double               cx, cy, radius;
       fzrt_boolean         read_more_data = TRUE;
       fz_xyz_td            origin;
       fz_objt_ptr          obj;

       err = my_file_open(floc, &file);
       if(err == FZRT_NOERR)
       {
              while(read_more_data)
              {
                     err = my_file_read_circle(&file, &cx, &cy, &radius, &read_more_data);

                     if(err == FZRT_NOERR)
                     {
                            origin.x = cx;
                            origin.y = cy;
                            origin.z = 0.0;
                            err = fz_objt_cnstr_circle(windex,
                                                       radius,
                                                       FZ_OBJT_MODEL_TYPE_SMOD,
                                                       &origin,
                                                       NULL,
                                                       NULL,
                                                       &obj );
                     }
              }

              if (err == FZRT_NOERR)
                     err = my_file_close(&file);
       }

       return(err);
}
```

Export options

Image export translators can display an export options dialog. The "Options..." button on the image export standard file Open dialog will be enabled if the options flags set by the `fz_ffmt_image_write_opts_flags` function has the `FZ_FFMT_OPTS_INIT_HAS_WRITE_OPTS_BIT` bit set. Individual items in the common section of the options dialog are enabled by setting the appropriate bits of the flags parameter to `fz_ffmt_image_write_opts_flags`. The appropriate bits are defined in fz_ffmt_image_write_iface_opts_flags_enum. Image export options are discussed in section 3.14.2 of the **form•Z** Users Manual.

The call back functions to export an image are defined in the fz_ffmt_cbak_image_fset.

The fz_ffmt_cbak_image_fset contains the following functions to support image export options:

The image translator export options dialog enable function (Required)

```
fzrt_error_td  fz_ffmt_cbak_image_write_opts_flags (
             fz_ffmt_ref_td      ffmt_id,
             long                *flags,
             long                *opts_flags
             );
```

This function is called by **form•Z** to get the enable state for the image export "Options..." button and the enable states for each item on the image export options dialog. The flags parameter is used to set the enable states of items on the common section of the options dialog. Appropriate bits for this parameter are defined in fz_ffmt_image_write_iface_opts_flags_enum. By default, all items are disabled. The opts_flags parameter enables the "Options..."button on the "Image Export" standard file Open dialog by setting it to FZ_FFMT_OPTS_INIT_HAS_WRITE_OPTS_BIT. If the "Options..." button is to be disabled, opts_flags should be set to 0 (this is the default).

An example of an image translator export options dialog enable function is shown below.

```
fzrt_error_td my_write_opts_get_flags (
             fz_ffmt_ref_td      ffmt_id,
             long *              flags,
             long *              opts_flags )
{
       fzrt_error_td       err = FZRT_NOERR;

       FZ_SETBIT(*flags, FZ_FFMT_IMAGE_WRITE_IFACE_OPTS_ENABLEPREVIEW_BIT);

       FZ_SETBIT(*opts_flags, FZ_FFMT_OPTS_INIT_HAS_WRITE_OPTS_BIT);

       return(err);
}
```

The image translator export options defaults function (optional)

```
fzrt_error_td  fz_ffmt_cbak_image_write_opts_default (
             fz_ffmt_ref_td             ffmt_id
             );
```

This function is called by **form•Z** to set default values of options. All custom options and any common options whose default values the file translator wishes to change must be set here. This function is only needed if the translator has custom options or the translator needs to change default values of any of the common options.

form•Z will have set the default values for common options prior to calling this function. This function can then change any of those values by calling fz_ffmt_image_write_opts_parm_set using the options pointer obtained from fz_ffmt_image_write_opts_get_ptr. If the translator needs to inspect the value of an option, it can be obtained by calling fz_ffmt_image_write_opts_parm_get.

An example of an image translator export options defaults function is shown below.

```
long   my_write_opts_flags;

fzrt_error_td  my_write_opts_default (
                    fz_ffmt_ref_td             ffmt_id )
```

```
{
	fzrt_error_td				err = FZRT_NOERR;
	fz_type_td				fz_type;
	long					flags;

	/* Change a default value in the common options */
	fz_ffmt_image_write_opts_parm_get(image_write_opts,
		FZ_FFMT_IMAGE_WRITE_OPTS_PARM_FLAGS, &fz_type);
	fz_type_get_long(&fz_type, &flags);
	FZ_SETBIT(flags, FZ_FFMT_IMAGE_WRITE_OPTS_PREVIEW_BIT);
	fz_type_set_long(&flags, &fz_type);
	fz_ffmt_image_write_opts_parm_set(image_write_opts,
		FZ_FFMT_IMAGE_WRITE_OPTS_PARM_FLAGS, &fz_type);

	/* Set a default value for a custom option */
	my_write_opts_flags = 0;

	return(err);
}
```

This function sets the "Include Preview" option (this not set in the **form•Z** default) and initializes a custom option.

The image translator export options changed function (optional)

```
fzrt_error_td  fz_ffmt_cbak_image_write_opts_changed (
		fz_ffmt_ref_td				ffmt_id,
		fz_ffmt_image_write_opts_enum		which
		);
```

This function is called by **form•Z** when the user changes the value of an option in the common section of the options dialog. This allows the translator to override the behavior of the common section of the options dialog by setting values of options or setting enable states of items. The which parameter specifies which parameter's value changed. Values of the common options can be set by calling `fz_ffmt_image_write_opts_parm_set` using the options pointer (`fz_ffmt_image_write_opts_ptr`) obtained from `fz_ffmt_image_write_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_image_write_opts_parm_get`. The enable states of items can be changes by first getting the enable flags by calling `fz_ffmt_image_write_get_dlog_flags`, then changing the enable bit of the item whose state needs to change and calling `fz_ffmt_image_write_set_dlog_flags`. Appropriate bits are defined in `fz_ffmt_image_write_iface_opts_flags_enum`. All these functions are in the `fz_ffmt_image_fset` function set.

The image translator export custom options dialog function (Optional)

```
fzrt_error_td  fz_ffmt_cbak_image_write_dlog_cust (
		fz_fuim_tmpl_ptr	fuim_tmpl,
		short			parent,
		fz_ffmt_ref_td		ffmt_id
		);
```

This function is called by **form•Z** to add items to the custom section of the options dialog. This function should add items by calling functions in the `fz_fuim_fset` function set using the `fz_fuim_mngr_ptr` and `parent` parameter (passed into this function) as the top level parent for all items. The `ffmt_id` parameter specifies the file translator's reference id.

An example of an image translator export custom options dialog function is shown below.

```
#define MY_STRINGS		1
#define MY_COMPRESS_STR		2

#define MY_READ_SUB_IMAG_BIT 1
```

```
long   my_write_opts_flags;

fzrt_error_td my_write_dlog_cust (
             fz_fuim_tmpl_ptr    fuim_tmpl,
             short               parent,
             fz_ffmt_ref_td      ffmt_id )
{
       fzrt_error_td err = FZRT_NOERR;
       short         gindx;
       char          title[256];

       err = fzrt_fzr_get_string (
                     _fz_rsrc_ref,
                     MY_STRINGS,
                     MY_COMPRESS_STR,
                     title );

       if(err == FZRT_NOERR)
       {
             if((gindx = fz_fuim_new_check(fuim_tmpl, parent, 0,
                           FZ_FUIM_FLAG_GFLT | FZ_FUIM_FLAG_HORZ, title, NULL, NULL)) > -1)
             {
                    fz_fuim_item_encod_long(fuim_tmpl, gindx, & my_write_opts_flags,
                                        TRUE, FZ_FUIM_BIT2_MASK(MY_READ_SUB_IMAG_BIT));
             }
       }

       return(FZRT_NOERR);
}
```

The image translator export pre-options dialog function (Optional)

```
fzrt_error_td  fz_ffmt_cbak_image_write_predloginit (
             long                windex,
             fz_ffmt_ref_td      ffmt_id
             );
```

This function is called by **form•Z** just prior to displaying the options dialog. This is done so the file translator can check the current state of **form•Z** and make any adjustments to the values of options or the enable states of items on the options dialog. For example, a translator may have a custom option that is only appropriate if the rendering is vector and not appropriate for a bitmap rendering. In this case, this function would check the type of the current rendering and disable the option's dialog item if the render type is bitmap. Options values and item enable states can be changed as described in for the `fz_ffmt_image_read_opts_changed` function.

Structured image bitmap file export

form•Z only exports renderings and draft window contents as bitmap images. The export process is more structured than the import process. **form•Z** calls several translator functions to write a bitmap image. The export of a bitmap image is as follows.

Initialize the export process, open the file and write the file header data
Write any bitmap image header data
Loop until entire image is written
    Write a few scanlines at a time
Write any trailing data to the file
Close the file and uninitialize the export process

To perform each of these steps **form•Z** calls the following image translator functions in the order listed.

`fz_ffmt_cbak_image_bmap_write_file_begin`
`fz_ffmt_cbak_image_bmap_write_image_begin`
`fz_ffmt_cbak_image_bmap_write_image_band` - This may be called multiple times.
`fz_ffmt_cbak_image_bmap_write_image_end`
`fz_ffmt_cbak_image_bmap_write_file_end`

The `fz_ffmt_cbak_image_fset` contains the following functions to support reading image bitmap files:

The image bitmap export file begin function

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_write_file_begin (
             long                windex,
             fz_ffmt_ref_td             ffmt_id,
             void                **data_ptr,
             fzrt_floc_ptr       floc
             );
```

This function is called by **form•Z** to open a file for writing. This function should write the image file header and allocate any memory required for parameters and working data. If the file open or memory allocation fails an error should be returned. `floc` is the file to open. `data_ptr` is a hook for passing translator defined data to subsequent translator functions. A common component of this data is the opened file pointer.

An example of an image bitmap export file begin function is shown below.

```
typedef struct
{
      my_file_td   file;
}  my_trans_data_td;

fzrt_error_td  my_bmap_write_file_begin (
             long                windex,
             fz_ffmt_ref_td      ffmt_id,
             void                **data_ptr,
             fzrt_floc_ptr       floc )
{
      fzrt_error_td       err = FZRT_NOERR;
      my_trans_data_td    *my_data = NULL;

      *data_ptr = NULL;
      my_data = (my_trans_data_td *)fzrt_new_ptr_clear(sizeof(my_trans_data_td));
      if(my_data != NULL)
      {
             *data_ptr = my_data ;
             err = my_open(my_data);
             if(err == FZRT_NOERR)
             {
                   my_write_file_header(my_data);
             }
      }
      else
      {
             err = fzrt_error_set (
                                 FZRT_MALLOC_ERROR,
                                 FZRT_ERROR_SEVERITY_ERROR,
                                 FZRT_ERROR_CONTEXT_FZRT,  0 );
      }
      return err;
}
```

The image bitmap export image begin function

```
fzrt_error_td fz_ffmt_cbak_image_bmap_write_image_begin (
             long                windex,
             fz_ffmt_ref_td             ffmt_id,
             void                *data,
             long                width,
             long                height,
             fzrt_boolean        has_alpha
             );
```

This function is called by **form•Z** to begin writing an image. `data` is a pointer to the translator data created in `ffmt_cbak_image_bmap_write_file_begin`. `width` and `height` specify the number of columns and rows of pixels for the image being exported. `fz_rndr_ibuf_get_parm` in the `fz_rndr_mngr_fset` function set can be used to get additional information about the image being exported.

All images exported from formZ have either 3 or 4 color channels (red, green, blue or red, green, blue, alpha). If `has_alpha` is set to `TRUE`, the exported image will have 4 color channels. Otherwise, the exported image will have 3 color channels.

An example of an image bitmap export image begin function is shown below.

```
typedef struct
{
	my_file_td	file;
	long		width;
	long		height;
	long		pix_depth;
}  my_trans_data_td;

fzrt_error_td  my_bmap_write_image_begin (
		long			windex,
		fz_ffmt_ref_td		ffmt_id,
		void			*data,
		long			width,
		long			height,
		fzrt_boolean		has_alpha
		)
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = (my_trans_data_td *)data;
	short			res, pix_depth;
	double			dres;
	fz_type_td		var_data;
	double			gamma;

	if(my_data != NULL)
	{
		fz_wind_image_opts_get_parm_data(windex, FZ_WIND_IMAGE_OPTS_SIZE_RES_VALUE,
								&var_data);
		fz_type_get_short(&var_data, &res);
		fz_wind_image_opts_get_parm_data(windex, FZ_WIND_IMAGE_OPTS_SIZE_TYPE,
				&var_data);
		fz_type_get_long(&var_data, (long *)&size_type);
		fz_wind_image_opts_get_parm_data(windex, FZ_WIND_IMAGE_OPTS_CUSTOM_TYPE,
						&var_data);
		fz_type_get_long(&var_data, (long *)&dimn_type);
		fz_wind_image_opts_get_parm_data(windex, FZ_WIND_IMAGE_OPTS_SIZE_RES_TYPE,
				&var_data);
		fz_type_get_long(&var_data, (long *)&res_type);

		if(has_alpha)	my_data->pix_depth = 32;
		else		my_data->pix_depth = 24;
		my_data->width = width;
		my_data->height = height;

		fz_rndr_ibuf_get_parm(windex, FZRT_UUID_NULL, FZ_RNDR_IBUF_PARM_GAMMA,
						&fz_type);
		fz_type_get_float(&fz_type, &gamma);

		if(size_type == FZ_WIND_IMAGE_SIZE_TYPE_CUSTOM &&
		   dimn_type == FZ_WIND_IMAGE_CUST_DIMN_TYPE_SIZE)
		{	/* convert to english */
			if(res_type == FZ_WIND_IMAGE_RES_TYPE_CM)
			{
```

```
                        dres = res * FZ_METRIC_FACTOR;
                        res = (short)floor(dres + 0.5);
                    }
                }
                else    res = 72;

                my_write_image_header(my_data, width, height, res, pix_depth, gamma);
        }
        return err;
}
```

The image bitmap export image band function

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_write_image_scanline_byte (
                long                        windex,
                fz_ffmt_ref_td                      ffmt_id,
                fz_rgb_uchar_td             *pixels,
                unsigned char               *alpha,
                long                        row
                );
```

This function is called by **form•Z** to write a single scanline of an image. `data` is a pointer to the translator data created in `ffmt_cbak_image_bmap_write_file_begin.` `pixels` contains red,green,blue pixelsof the image beign exported. `alpha` contains the alpha value pixels. If the image being exported does not contain any alpha pixels, this pointer will be `NULL`. `fz_rndr_ibuf_get_parm` in the `fz_rndr_mngr_fset` function set can be used to get additional information about the image being exported. **form•Z** will call this function for each scanline (row) of pixels (a band is one or more scanlines) until all the entire image has been exported. Scanlines are exported in order from the top of the image to the bottom. `row` is the index of the scanline being exported.

An example of an image bitmap export band function is shown below.

```
typedef struct
{
        my_file_td    file;
}  my_trans_data_td;

fzrt_error_td  my_bmap_write_image_scanline (
                long                windex,
                fz_ffmt_ref_td              ffmt_id,
                void                *data,
                fz_rgb_uchar_td     *pixels,
                unsigned char       *alpha,
                long                row
                )
{
        fzrt_error_td       err = FZRT_NOERR;
        my_trans_data_td    *my_data = (my_trans_data_td *)data;

        if(my_data != NULL)
        {
                my_write_image_pixels(my_data, pixels, alpha, row);
        }
        return err;
}
```

The image bitmap export image end function

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_write_image_end (
                long                        windex,
                fz_ffmt_ref_td                      ffmt_id,
                void                        *data,
                );
```

This function is called by **form•Z** to end the writing of a bitmap image. `data` is a pointer to the translator data created in `ffmt_cbak_image_bmap_write_file_begin`.

An example of an image bitmap export image end function is shown below.

```
typedef struct
{
	my_file_td   file;
}  my_trans_data_td;

fzrt_error_td my_bmap_write_image_end (
		long			windex,
		fz_ffmt_ref_td		ffmt_id,
		void			*data
		)
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = (my_trans_data_td *)data;


	if(my_data != NULL)
	{
		my_write_image_trailer(my_data);
	}
	return err;
}
```

The image bitmap export file end function

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_write_file_end (
		long				windex,
		fz_ffmt_ref_td			ffmt_id,
		void				**data_ptr,
		fzrt_floc_ptr			floc,
		fzrt_error_td			err
		);
```

This function is called by **form•Z** to close an image file and cleanup if any error occurred during the export of the image. This function should free any memory allocated in fz_`ffmt_cbak_image_bmap_write_file_begin`. `data` is a pointer to the translator data created in fz_`ffmt_cbak_image_bmap_write_file_begin`. `floc` is the file. `err` is the last encountered error. This can be used for any extra cleanup in case of an error. `err` is set to `FZRT_NOERR` if no error occurred during export.

An example of an image bitmap export file end function is shown below.

```
typedef struct
{
	my_file_td   file;
}  my_trans_data_td;

fzrt_error_td  my_bmap_write_file_end (
		long			windex,
		fz_ffmt_ref_td		ffmt_id,
		void			**data_ptr,
		fzrt_floc_ptr		floc,
		fzrt_error_td		err)
{
	fzrt_error_td		err2 = FZRT_NOERR;
	my_trans_data_td	*my_data = *((my_trans_data_td **)data_ptr);

	if(err != FZRT_NOERR)		err2 = my_cleanup_from_error(my_data, err);
	if(my_data != NULL)
	{
		if(my_data->file != NULL)
```

```
            {
                  err2 = my_write_file_trailer(my_data);
                  if (err2 == FZRT_NOERR)
                        err2 = my_file_close(my_data);
            }
            fzrt_dispose_ptr((fzrt_ptr)my_data);
            *data_ptr = NULL;
      }

      return(err2);
}
```

The image bitmap export progress string function

```
fzrt_error_td  fz_ffmt_image_cbak_bmap_write_progress_str (
            long                        windex,
            fz_ffmt_ref_td                    ffmt_id,
            char                        *str,
            long                        max_len
            );
```

During the export of an image, **form•Z** displays a progress dialog. This function is called by **form•Z** to get a string for display on the image export progress dialog.

An example of an image bitmap export progress string function is shown below.

```
#define MY_STRINGS                1
#define MY_WRITING_FILE_STR       3

fzrt_error_td  my_bmap_write_progress_str(
                  long                  windex,
                  fz_ffmt_ref_td        ffmt_id,
                  char                  *str,
                  long                  max_len)
{
      char str1[256];

      if(str != NULL && max_len > 0)
      {
            fzrt_fzr_get_string(_fz_rsrc_ref, MY_STRINGS, MY_WRITING_FILE_STR, str1);
            strncpy(str, str1, max_len);
      }

      return(FZRT_NOERR);
}
```

The image bitmap export error label function

```
fzrt_error_td  fz_ffmt_cbak_image_bmap_write_err_label (
            long                        windex,
            fz_ffmt_ref_td                    ffmt_id,
            char                        *str,
            long                        max_len,
            short                       *err_id
            );
```

If an error occurs when exporting an image, **form•Z** will display an error dialog indicating the error. This function is called by **form•Z** to obtain a string to display to the user. This is a general error message for all errors. Specific error strings are returned by the error string function registered with the plugin.

An example of an image bitmap export error label function is shown below.

```
#define MY_STRINGS                1
#define MY_WRITE_ERR_STR          4
```

```
fzrt_error_td  my_bmap_write_err_label(
            long                windex,
            fz_ffmt_ref_td             ffmt_id,
            char                *str,
            long                max_len,
            short               *err_id )
{
      fzrt_error_td err = FZRT_NOERR;
      char str1[256];

      err = fzrt_fzr_get_string(_fz_rsrc_ref, MY_STRINGS, MY_WRITE_ERR_STR, str1);
      if(err == FZRT_NOERR)      strncpy(str, str1, max_len);

      return(err);
}
```

Structured image vector file export

form•Z only exports vector renderings and draft window contents as vector images. The export process is more structured than the import process. **form•Z** calls several translator functions to export a vector image.

Vector data consists of geometry and attributes. **form•Z** can export vector geometry as single points, single line segments, linesets (linesets are a connected set of line segments), text, circles, ellipses, and arcs. **form•Z** exports geometry in a pixel coordinate system. This is normally screen pixels except in the case where a model rendering is exported with a user specified image size. The x axis is horizontal and the y axis is vertical with positive y is bottom up. The positive y can by changed to top down by setting the `invrt_proj` parameter of the `fz_ffmt_cbak_image_vect_write_begin` to `TRUE`.

Attributes that **form•Z** exports consist of line color, fill color, line weight, line style, and fill pattern. The line color is the color used to draw points, line segments and linesets. The fill color is the color used to fill the interior of linesets. The line weight is described in section 5.15.2 of the **form•Z** Users Manual. The line style is described in section 5.15.1 of the **form•Z** Users Manual. The fill pattern is an 8x8 pixel pattern. Each pixel is represented by one bit. A pixel value of 0 designates a transparent pixel (don't draw that pixel). A pixel value of 1 designates the current fill color. **form•Z** will set an attribute then all subsequent exported geometry will have that attribute. For example, if the line color is set to red all following points and line will have the color red until the line color is set to another color.

The export of a bitmap image is as follows.

```
      Initialize the export process and open the file
      Loop until all vector data is written
            If attributes have changed
                  set new attributes
            Write vector data
      Close the file and uninitialize the export process
```

The `fz_ffmt_cbak_image_fset` contains the following functions to support reading image bitmap files:

The image vector export begin function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_begin (
            long                                        windex,
            fz_ffmt_ref_td                                     ffmt_id,
            void                                        **data_ptr,
            fzrt_floc_ptr                               floc
            fzrt_boolean                                *invrt_proj,
            fz_ffmt_image_vect_write_text_method_enum          *text_method,
            double                                      *deflt_line_weight,
            fzrt_boolean                                *full_faces,
            fzrt_boolean                                *pnts_as_pnts
            );
```

This function is called by **form•Z** to open a file for writing. This function should write the image file header and allocate any memory required for parameters and working data. If the file open or memory allocation fails an error should be returned.

An example of an image vector export begin function is shown below.

```
typedef struct
{
	my_file_td		file;
	fzrt_rgb_color_td	line_color;
	fzrt_rgb_color_td	fill_color;
	double			line_weight;
	my_line_style_td	line_style;
	my_fill_pat_td		fill_pattern;
}  my_trans_data_td;

fzrt_error_td  my_vect_write_begin (
		long							windex,
		fz_ffmt_ref_td						ffmt_id,
		void							**data_ptr,
		fzrt_floc_ptr						floc,
		fzrt_boolean						*invrt_proj,
		fz_ffmt_image_vect_write_text_method_enum		*text_method,
		double							*deflt_line_weight,
		fzrt_boolean						*full_faces,
		fzrt_boolean						*pnts_as_pnts )
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = NULL;

	*data_ptr = NULL;
	my_data = (my_trans_data_td *)fzrt_new_ptr_clear(sizeof(my_trans_data_td));
	if(my_data != NULL)
	{
		*data_ptr = my_data ;
		err = my_open(&my_data->file);
		if(err == FZRT_NOERR)
		{
			my_data->line_color.red = 0;
			my_data->line_color.green = 0;
			my_data->line_color.blue = 0xffff;
			my_data->fill_color.red = 0xffff;
			my_data->fill_color.green = 0xffff;
			my_data->fill_color.blue = 0;
			my_data->line_weight= 1.0;
			my_init_line_style(my_data->line_style);
			my_init_fill_pat(my_data->fill_pattern);
			err = my_write_file_header(my_data->file);
			if(err == FZRT_NOERR)
			{
				*invrt_proj = 0;
				*text_method = FZ_FFMT_IMAGE_VECT_WRITE_TEXT_METH_AS_PATHS;
				*deflt_line_weight = 1.0;
				*full_faces = FALSE;
				*pnts_as_pnts = FALSE;
			}
		}
	}
	else
	{
		err = fzrt_error_set (
					FZRT_MALLOC_ERROR,
					FZRT_ERROR_SEVERITY_ERROR,
					FZRT_ERROR_CONTEXT_FZRT,  0 );
	}
```

```
	return err;
}
```

The image vector export end function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_end (
		long				windex,
		fz_ffmt_ref_td			ffmt_id,
		void				**data_ptr,
		fzrt_floc_ptr			floc,
		fzrt_error_td			err
		);
```

This function is called by **form•Z** to close an image file and cleanup if any error occurred during the export of the image. This function should free any memory allocated in `fz_ffmt_cbak_image_vect_write_begin`. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. `floc` is the file. `err` is the last encountered error. This can be used for any extra cleanup in case of an error. `err` is set to `FZRT_NOERR` if no error occurred during export.

An example of an image vector export end function is shown below.

```
typedef struct
{
	my_file_td		file;
	fzrt_rgb_color_td	line_color;
	fzrt_rgb_color_td	fill_color;
	double			line_weight;
	my_line_style_td	line_style;
	my_fill_pat_td		fill_pattern;
}  my_trans_data_td;

fzrt_error_td  my_vect_write_end (
		long			windex,
		fz_ffmt_ref_td		ffmt_id,
		void			**data_ptr,
		fzrt_floc_ptr		floc,
		fzrt_error_td		err)
{
	fzrt_error_td		err2 = FZRT_NOERR;
	my_trans_data_td	*my_data = *((my_trans_data_td **)data_ptr);

	if(my_data != NULL)
	{
		if(my_data->file != NULL)
		{
			err2 = my_write_file_trailer(my_data->file);
			if (err2 == FZRT_NOERR)
				err2 = my_file_close(my_data->file);
		}
		fzrt_dispose_ptr((fzrt_ptr)my_data);
		*data_ptr = NULL;
	}

	return(err2);
}
```

The image vector export point function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_point (
			long			windex,
			fz_ffmt_ref_td		ffmt_id,
			void			*data,
			double			x,
			double			y
			);
```

This function is called by **form•Z** to write a single point to the file. `data` is a pointer to the translator data created in `ffmt_image_vect_write_begin`. `x` and `y` specify the location of the point in image coordinates.

An example of an image vector export point function is shown below.

```
typedef struct
{
	my_file_td		file;
	fzrt_rgb_color_td	line_color;
	fzrt_rgb_color_td	fill_color;
	double			line_weight;
	my_line_style_td	line_style;
	my_fill_pat_td			fill_pattern;
}  my_trans_data_td;

fzrt_error_td  my_vect_write_point (
			long			windex,
			fz_ffmt_ref_td			ffmt_id,
			void			*data,
			double			x,
			double			y )
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = (my_trans_data_td *)data;

	err = my_write_point(&my_data->file, my_data->line_color, x, y);

	return(err);
}
```

The image vector export line function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_line (
			long			windex,
			fz_ffmt_ref_td			ffmt_id,
			void			*data,
			double			x1,
			double			y1,
			double			x2,
			double			y2
			);
```

This function is called by **form•Z** to write a single line segment to a file. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. `x1`, `y1` and `x2`, `y2` are the end points of the line segment.

An example of an image vector export line function is shown below.

```
typedef struct
{
	my_file_td		file;
	fzrt_rgb_color_td	line_color;
	fzrt_rgb_color_td	fill_color;
	double			line_weight;
	my_line_style_td	line_style;
	my_fill_pat_td		fill_pattern;
}  my_trans_data_td;

fzrt_error_td  my_vect_write_line (
			long			windex,
			fz_ffmt_ref_td			ffmt_id,
			void			*data,
			double			x1,
```

```
                    double              y1,
                    double              x2,
                    double              y2)
{
       fzrt_error_td       err = FZRT_NOERR;
       my_trans_data_td    *my_data = (my_trans_data_td *)data;

       err = my_write_line(&my_data->file, my_data->line_color, my_data->line_style,
                                  x1, y2, x2, y2);

       return(err);
}
```

The image vector export lineset function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_lineset (
                    long                         windex,
                    fz_ffmt_ref_td                      ffmt_id,
                    void                         *data,
                    long                         how,
                    fz_ffmt_image_lineset_ptr  line_set
                    );
```

This function is called by **form•Z** to write a lineset to a file. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. The `how` parameter specifies what the lineset represents. It could represent an outline, a filled polygon, both, or a clipping region. The number of points and an array of points in the lineset are obtained by calling `fz_ffmt_image_lineset_get_pnts` in the `fz_ffmt_image_fset` function set. `fz_ffmt_image_lineset_get_pnts` also sets an indicator of whether the lineset is open or closed. If an open lineset is specified by the `how` parameter as being filled, it should be filled as if it is a closed lineset.

An example of an image vector export lineset function is shown below.

```
fzrt_error_td  my_vect_write_lineset (
                    long                         windex,
                    fz_ffmt_ref_td                      ffmt_id,
                    void                         *data,
                    long                         how,
                    fz_ffmt_image_lineset_ptr  line_set )
{
       fzrt_error_td       err = FZRT_NOERR;
       my_trans_data_td    *my_data = (my_trans_data_td *)data;
       fz_xy_td            *lset_pnts = NULL;
       fzrt_boolean        close_lineset;
       long                n;

       err = fz_ffmt_image_lineset_get_pnts (windex, line_set, NULL, &n, &close_lineset);
       if(err == FZRT_NOERR && n > 0)
       {
              lset_pnts = (fz_xy_td *)fzrt_new_ptr ( sizeof(fz_xy_td) * n );
              if(lset_pnts != NULL )
              {
                     err = fz_ffmt_image_lineset_get_pnts (windex, line_set, lset_pnts, &n,
                                                         &close_lineset);
                     if(err == FZRT_NOERR )
                     {
                            if(FZ_CHKBIT(how , FZ_FFMT_LINESET_FLAGS_FILL_BIT))
                            {
                                   err = my_write_fill_polyline(my_data->file,
                                                         my_data->fill_color,
                                                         my_data->fill_pattern,
                                                         n, lset_pnts);
                            }
                            if(FZ_CHKBIT(how , FZ_FFMT_LINESET_FLAGS_STROKE_BIT))
                            {
```

```
                        err = my_write_outline_polyline(my_data->file,
                                                        my_data->line_color,
                                                        my_data->line_style,
                                                        my_data->line_weight,
                                                        n, lset_pnts);
                    }
                    if(FZ_CHKBIT(how , FZ_FFMT_LINESET_FLAGS_CLIP_BIT))
                    {
                        err = my_write_clip_polyline(my_data->file, n, lset_pnts);
                    }
                }
            }
        }

        return(err);
}
```

The image vector export begin compound function

```
fzrt_error_td fz_ffmt_cbak_image_vect_write_begin_compound (
                    long                        windex,
                    fz_ffmt_ref_td              ffmt_id,
                    void                        *data,
                    fz_ffmt_image_lineset_ptr   line_set
                    );
```

form•Z may at times need to represent multiple linesets a parts of a single vector object. This could be because several linesets are grouped or joined as a single object or it could be because a filled lineset has one or more holes. For example, if text is exported as paths (linesets), the lower case 'i' is exported as a single object with 2 linesets, the lower case 'e' is exported as two linesets with the second representing a hole. This function is called by **form•Z** to begin a grouping of linesets. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`.

The image vector export end compound function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_end_compound (
                    long                        windex,
                    fz_ffmt_ref_td                     ffmt_id,
                    void                        *data,
                    fz_ffmt_image_lineset_ptr   line_set
                    );
```

This function is called by **form•Z** to end a grouping of linesets which was begun by `fz_ffmt_image_vect_write_begin_compound`. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`.

The image vector export can do arc function

```
fzrt_boolean  fz_ffmt_cbak_image_vect_write_can_do_arc (
                    long                        windex,
                    fz_ffmt_ref_td                     ffmt_id,
                    void                        *data,
                    double                      cx,
                    double                      cy,
                    double                      rx,
                    double                      ry,
                    double                      start_ang,
                    double                      end_ang,
                    double                      pitch,
                    long                        how
                    );
```

This function is called by **form•Z** to determine if the translator can export a specific circle, ellipse, or arc. Arcs can be circular or elliptical. If the translator can not export the specified arc as an arc, it should return FALSE and **form•Z** will then export the arc as a lineset. For example, if a translator can not export elliptical arcs or rotated ellipses, it should return FALSE when such an arc is detected and the arc or ellipse will be exported as a lineset. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. cx and cy designate the center of the arc, circle or ellipse. rx specifies the radius along the x axis and ry specifies the radius along the y axis. For circles and circular arcs rx will equal ry. start_ang and end_ang specify the starting and ending angles for an arc. For circles and ellipses, these values will be 0.0 and `FZ_2PI` respecitvely. pitch specifies a rotation to be applied to the generated circle, ellipse or arc. The `how` parameter specifies what the lineset represents. It could represent an outline, a filled polygon, both, or a clipping region.

An example of an image vector export can do arc function is shown below.

```
fzrt_boolean  my_vect_write_can_do_arc (
                    long                        windex,
                    fz_ffmt_ref_td                     ffmt_id,
                    void                        *data,
                    double                      cx,
                    double                      cy,
                    double                      rx,
                    double                      ry,
                    double                      start_ang,
                    double                      end_ang,
                    double                      pitch,
                    long                        how )
{
        fzrt_boolean        rv = TRUE;

        if(rx != ry)        /* ellipse */
        {
                if(start_ang != 0.0 || end_ang != FZ_2PI)
                {
                        /* Elliptical Arc */
                        rv = FALSE;
                }
                else if(fmod(pitch, FZ_PI2) != 0.0)
                {
                        /* Rotated Ellipse */
                        rv = FALSE;
                }
        }

        return(rv);
}
```

The image vector export arc function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_arc (
                    long                        windex,
                    fz_ffmt_ref_td                     ffmt_id,
                    void                        *data,
                    double                      cx,
                    double                      cy,
                    double                      rx,
                    double                      ry,
                    double                      start_ang,
                    double                      end_ang,
                    double                      pitch,
                    long                        how
                    );
```

This function is called by **form•Z** to export circles, ellipses, and arcs. Arcs can be circular or elliptical. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. cx and cy designate the center of the arc, circle or ellipse. rx specifies the radius along the x axis and ry specifies the radius along the y axis. For circles and circular arcs rx will equal ry. start_ang and end_ang specify the starting and ending angles for an arc. For circles and ellipses, these values will be 0.0 and `FZ_2PI` respecitvely. `pitch` specifies a rotation to be applied to the generated circle, ellipse or arc. The `how` parameter specifies what the lineset represents. It could represent an outline, a filled polygon, both, or a clipping region.

An example of an image vector export arc function is shown below.

```
fzrt_error_td  my_vect_write_arc (
                    long                        windex,
                    fz_ffmt_ref_td              ffmt_id,
                    void                        *data,
                    double                      cx,
                    double                      cy,
                    double                      rx,
                    double                      ry,
                    double                      start_ang,
                    double                      end_ang,
                    double                      pitch,
                    long                        how )
{
      fzrt_error_td       err = FZRT_NOERR;
      my_trans_data_td    *my_data = (my_trans_data_td *)data;
      double              my_rx, my_ry;

      if(rx != ry)        /* ellipse */
      {
            if((long)(pitch/FZ_PI2) & 1)      /* if rotated 90 deg */
            {                                 /* swap rx and ry */
                  my_rx = ry;
                  my_ry = rx;
            }
            else
            {
                  my_rx = rx;
                  my_ry = ry;
            }
            if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_FILL_BIT))
            {
                  err = my_write_fill_ellipse(my_data->file, my_data->fill_color,
                                          my_data->fill_pattern,
                                          cx, cy, my_rx, my_ry);
            }
            if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_STROKE_BIT))
            {
                  err = my_write_outline_ellipse(my_data->file, my_data->line_color,
                                          my_data->line_style,
                                          my_data->line_weight,
                                          cx, cy, my_rx, my_ry);
            }
            if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_CLIP_BIT))
            {
                  err = my_write_clip_ellipse(my_data->file, cx, cy, my_rx, my_ry);
            }
      }
      else                /* circle */
      {
            if(fmod((end_ang-start_ang),_2PI) == 0.0)            /* closed circle */
            {
                  if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_FILL_BIT))
                  {
                        my_write_fill_circle(my_data->file, my_data->fill_color,
                                          my_data->fill_pattern, cx, cy, rx);
                  }
                  if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_STROKE_BIT))
                  {
```

```
                        my_write_outline_circle(my_data->file, my_data->fill_color,
                                                my_data->fill_pattern, cx, cy, rx);
                    }
                    if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_CLIP_BIT))
                    {
                        my_write_clip_circle(my_data->file, cx, cy, rx);
                    }
                }
                else                                    /* arc */
                {
                    if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_FILL_BIT))
                    {
                        my_write_fill_arc(my_data->file, my_data->fill_color,
                                                my_data->fill_pattern, cx, cy, rx,
                                                start_ang + pitch,
                                                end_ang + pitch);
                    }
                    if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_STROKE_BIT))
                    {
                        my_write_outline_arc(my_data->file, my_data->fill_color,
                                                my_data->fill_pattern, cx, cy, rx,
                                                start_ang + pitch,
                                                end_ang + pitch);
                    }
                    if(FZ_CHKBIT(how, FZ_FFMT_LINESET_FLAGS_CLIP_BIT))
                    {
                        my_write_clip_arc(my_data->file, cx, cy, rx,
                                                start_ang + pitch,
                                                end_ang + pitch);
                    }
                }
        }

        return(err);
}
```

The image vector export simple text function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_simple_text (
                    long                windex,
                    fz_ffmt_ref_td              ffmt_id,
                    void                *data,
                    char                *fname,
                    double              txsize,
                    double              x,
                    double              y,
                    char                *str
                    );
```

This function is called by **form•Z** to export simple text. Simple text is text that is all the same font and size. It has no style (normal, bold, italic, etc) unless the style is implicit in the font. It is not rotated and follows a linear path. Any new line characters will be ignored and the text will export on one line. Currently, **form•Z** only calls this function to export axis labels. `fname` is the name of the font to be applied to the text and `txsize` is the font's point size. `x, y` is the position of the lower left corner of the text. `str` is the actual text to export.

An example of an image vector export point function is shown below.

```
#define MY_NORMAL_TEXT_STYLE      1

typedef struct
{
        my_file_td          file;
        fzrt_rgb_color_td   line_color;
        fzrt_rgb_color_td   fill_color;
        double              line_weight;
```

```
	my_line_style_td	line_style;
	my_fill_pat_td		fill_pattern;
}  my_trans_data_td;

fzrt_error_td  my_vect_write_simple_text (
			long			windex,
			fz_ffmt_ref_td			ffmt_id,
			void			*data,
			char			*fname,
			double			txsize,
			double			x,
			double			y,
			char			*str )
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = (my_trans_data_td *)data;

	err = my_write_text(my_data->file, my_data->line_color, font, MY_NORMAL_TEXT_STYLE,
				tx_size, x, y, str);

	return(err);
}
```

The image vector export set line color function

```
fzrt_error_td  ffmt_cbak_image_vect_write_set_line_color (
			long				windex,
			fz_ffmt_ref_td				ffmt_id,
			void				*data,
			const fzrt_rgb_color_td *const	rgb_color
			);
```

This function is called by **form•Z** to set the line color. All points and lines exported after this call will have the color specified by `rgb_color`. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`.

An example of an image vector export set line color function is shown below.

```
typedef struct
{
	my_file_td		file;
	fzrt_rgb_color_td	line_color;
	fzrt_rgb_color_td	fill_color;
	double			line_weight;
	my_line_style_td	line_style;
	my_fill_pat_td		fill_pattern;
}  my_trans_data_td;

fzrt_error_td my_vect_write_set_line_color (
			long				windex,
			fz_ffmt_ref_td				ffmt_id,
			void				*data,
			const fzrt_rgb_color_td *const	rgb_color )
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = (my_trans_data_td *)data;

	my_data->line_color = *rgb_color;

	return(rv);
}
```

The image vector export set fill color function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_set_fill_color (
```

```
                    long                            windex,
                    fz_ffmt_ref_td                         ffmt_id,
                    void                            *data,
                    const fzrt_rgb_color_td *const   rgb_color
                    );
```

This function is called by **form•Z** to set the fill color. All filled linesets exported after this call will be filled with the color specified by `rgb_color`. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`.

An example of an image vector export set fill color function is shown below.

```
typedef struct
{
      my_file_td          file;
      fzrt_rgb_color_td   line_color;
      fzrt_rgb_color_td   fill_color;
      double              line_weight;
      my_line_style_td    line_style;
      my_fill_pat_td      fill_pattern;
}  my_trans_data_td;

fzrt_error_td  my_vect_write_set_fill_color (
                    long                            windex,
                    fz_ffmt_ref_td                         ffmt_id,
                    void                            *data,
                    const fzrt_rgb_color_td *const   rgb_color )
{
      fzrt_error_td       err = FZRT_NOERR;
      my_trans_data_td    *my_data = (my_trans_data_td *)data;

      my_data->fill_color = *rgb_color;

      return(err);
}
```

The image vector export set line style function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_set_line style (
                    long                windex,
                    fz_ffmt_ref_td             ffmt_id,
                    void                *data,
                    char                type,
                    short               lsty_flag,
                    double              delta,
                    double              d1,
                    double              ddd,
                    double              drawn
                    );
```

This function is called by **form•Z** to set the line style. All lines exported after this call will have the specified line style. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. `type` designates an item in the **form•Z** draft line style palette. If type is -1, the line style is set to solid.

The image vector export set line weight function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_set_line_weight (
                    long                windex,
                    fz_ffmt_ref_td             ffmt_id,
                    void                *data,
                    double              line_weight
                    );
```

This function is called by **form•Z** to set the line weight. All lines exported after this call will have the specified line weight. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`. `line_weight` is specified in points (1 point = 1/72 inch). Fractional values are allowed.

An example of an image vector export set fill color function is shown below.

```
typedef struct
{
	my_file_td          file;
	fzrt_rgb_color_td   line_color;
	fzrt_rgb_color_td   fill_color;
	double              line_weight;
	my_line_style_td    line_style;
	my_fill_pat_td      fill_pattern;
}  my_trans_data_td;

fzrt_error_td my_vect_write_set_line_weight (
                    long                        windex,
                    fz_ffmt_ref_td              ffmt_id,
                    void                        *data,
                    double                      line_weight)
{
	fzrt_error_td       err = FZRT_NOERR;
	my_trans_data_td    *my_data = (my_trans_data_td *)data;

	my_data->line_weight = line_weight;

	return(err);
}
```

The image vector export set fill pattern function

```
fzrt_error_td fz_ffmt_cbak_image_vect_write_set_fill_pattern (
                    long                        windex,
                    fz_ffmt_ref_td                      ffmt_id,
                    void                        *data,
                    fzrt_pen_pattern_ptr                pat,
                    ffmt_image_lineset_ptr      line_set
                    );
```

This function is called by **form•Z** to set the fill pattern. `data` is a pointer to the translator data created in `fz_ffmt_cbak_image_vect_write_begin`.

The image vector export progress string function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_progress_str (
             long                        windex,
             fz_ffmt_ref_td              ffmt_id,
             char                        *str,
             long                        max_len
             );
```

During the export of an image, **form•Z** displays a progress dialog. This function is called by **form•Z** to get a string for display on the image export progress dialog.

An example of an image bitmap export progress string function is shown below.

```
#define MY_STRINGS                1
#define MY_WRITING_FILE_STR       3

fzrt_error_td my_vect_write_progress_str(
                    long                windex,
```

```
                    fz_ffmt_ref_td      ffmt_id,
                    char                *str,
                    long                max_len)
{
       char str1[256];

       if(str != NULL && max_len > 0)
       {
              fzrt_fzr_get_string(_fz_rsrc_ref, MY_STRINGS, MY_WRITING_FILE_STR, str1);
              strncpy(str, str1, max_len);
       }

       return(FZRT_NOERR);
}
```

The image vector export error label function

```
fzrt_error_td  fz_ffmt_cbak_image_vect_write_err_label (
              long                        windex,
              fz_ffmt_ref_td              ffmt_id,
              char                        *str,
              long                        max_len,
              short                       *err_id
              );
```

If an error occurs when exporting an image, **form•Z** will display an error dialog indicating the error. This function is called by **form•Z** to obtain a string to display to the user. This is a general error message for all errors. Specific error strings are returned by the error string function registered with the plugin.

An example of an image bitmap export error label function is shown below.

```
#define MY_STRINGS                  1
#define MY_WRITE_ERR_STR            4

fzrt_error_td my_vect_write_err_label(
              long          windex,
              fz_ffmt_ref_td       ffmt_id,
              char          *str,
              long          max_len,
              short         *err_id )
{
       err = FZRT_NOERR;
       char str1[256];

       err = fzrt_fzr_get_string(_fz_rsrc_ref, MY_STRINGS, MY_WRITE_ERR_STR, str1);
       if(err == FZRT_NOERR)       strncpy(str, str1, max_len);

       return(err);
}
```

Structured data file translators

Data file translators read and/or write 2D and 3D model and draft data. What functions a data translator performs (read or write; model or draft) a translator supports are determined by which functions and function sets are implemented by the translator. At this time only model data translators are supported in the API.

Structured data model file translators

Data model file translators read and/or write model data. What functions a data translator performs (read or write) a translator supports are determined by which functions in the `fz_ffmt_cbak_data_model_fset` are implemented by the translator. Callback functions in the `fz_ffmt_cbak_data_model_fset` function set that begin with `fz_ffmt_cbak_data_model_read` are for reading model data. Callback functions in the fz_ffmt_cbak_data_model_fset function set that begin with fz_ffmt_cbak_data_model_write are for writing model data.

The transform options (described in section 3.13.1 of the **form•Z** Users Manual) are shared between the import and export options. When accessing the transform options, it is important to check the transform direction. This is defined as either import or export. If the direction is import that means that the transform options are defined for an import operation. Therefore if these options are accessed by an exporter, the values of the options will need to be inverted. By default, **form•Z** defines the transform options for import. Normally a translator wil not need to access the transform options since **form•Z** transforms data on import and export.

Two function sets are needed for a data model file translator, the translator information function set and the data model function set. The translator information function set is identified by the following constants, FZ_FILE_DATA_EXTS_TYPE (plugin type UUID), FZ_FILE_DATA_EXTS_NAME (plugin type name), and FZ_FILE_DATA_EXTS_VERSION (plugin type version).

The example below shows the definition of a data model file translator and the registration of the two function sets within that plugin.

```
#define MY_STRINGS           1
#define MY_NAME_STR          1
#define MY_PLUGIN_UUID       "\xfc\x98\x6f\x83\xf2\xd6\x4b\x9c\xb1\xc4\x0\x32\xf\x96\x8a\xfc"
#define MY_PLUGIN_VERSION    FZPL_VERS_MAKE(1,0,0,0)
#define MY_PLUGIN_VENDOR     "My Company Name"
#define MY_PLUGIN_URL        "www.myurl.com"

static fzrt_error_td  my_data_translator_register_plugin ()
{
       fzrt_error_td       err = FZRT_NOERR;
       long                num_failed = 0;
       char                pname[FZPL_NAME_SIZE];

       /* Register the plugin */

       err = fzrt_fzr_get_string (
                    _fz_rsrc_ref,
                    MY_STRINGS,
                    MY_NAME_STR,
                    pname );
       if ( err == FZRT_NOERR )
       {
              err = fset_glue->fzpl_plugin_register (
                           MY_PLUGIN_UUID,
                           pname,
                           MY_PLUGIN_VERSION,
                           MY_PLUGIN_VENDOR,
                           MY_PLUGIN_URL,
                           FZ_FILE_DATA_EXTS_TYPE,
                           FZ_FILE_DATA_EXTS_VERSION,
                           NULL,
                           0,
                           NULL,
                           &my_plugin_runtime_id );
       }
       if ( err == FZRT_NOERR )
       {
              /* Add the function sets implemented by the plugin */

              err = fset_glue->fzpl_plugin_add_fset (
                           my_plugin_runtime_id,
                           FZ_FFMT_CBAK_INFO_FSET_TYPE,
                           FZ_FFMT_CBAK_INFO_FSET_VERSION,
                           FZ_FFMT_CBAK_INFO_FSET_NAME,
                           FZPL_TYPE_STRING(fz_ffmt_cbak_info_fset),
                           sizeof ( fz_ffmt_cbak_info_fset ),
```

```
                        my_fill_translator_info_fset,
                        FALSE);

            if(err == FZRT_NOERR)
            {
                  err = fset_glue->fzpl_plugin_add_fset (
                              my_plugin_runtime_id,
                              FZ_FFMT_CBAK_DATA_MODEL_FSET_TYPE,
                              FZ_FFMT_CBAK_DATA_MODEL_FSET_VERSION,
                              FZ_FFMT_CBAK_DATA_MODEL_FSET_NAME,
                              FZPL_TYPE_STRING(fz_ffmt_cbak_data_model_fset),
                              sizeof ( fz_ffmt_cbak_data_model_fset ),
                              my_fill_data_model_cbak_fset,
                              FALSE);
            }
      }

      return(err);
}
```

The example below shows the function set fill functions for the `fz_ffmt_cbak_info_fset` and the `fz_ffmt_cbak_data_model_fset` function sets.

```
fzrt_error_td  my_fill_translator_info_fset (
            const fzpl_fset_def_ptr    fset_def,
            fzpl_fset_td * const       fset )
{
      fzrt_error_td                    err = FZRT_NOERR;
      fz_ffmt_cbak_info_fset           *info_funcs;

      err = _fset_glue->fzpl_fset_def_check ( fset_def,
                        FZ_FFMT_CBAK_INFO_FSET_VERSION,
                        FZPL_TYPE_STRING(fz_ffmt_cbak_info_fset),
                        sizeof(fz_ffmt_cbak_info_fset),
                        FZPL_VERSION_OP_NEWER );
      if ( err == FZRT_NOERR )
      {
            info_funcs = (fz_ffmt_cbak_info_fset *)fset;

            info_funcs->fz_ffmt_cbak_name              = my_name;
            info_funcs->fz_ffmt_cbak_uuid              = my_uuid;
            info_funcs->fz_ffmt_cbak_info              = my_info;
            info_funcs->fz_ffmt_cbak_ftype             = my_ftype;
            info_funcs->fz_ffmt_cbak_icon_rsrc         = my_icon_rsrc;
            info_funcs->fz_ffmt_cbak_icon_file         = my_icon_file;
            info_funcs->fz_ffmt_cbak_opts_io           = my_opts_io;
            info_funcs->fz_ffmt_cbak_is_file           = my_is_file;
      }

      return(err);
}

fzrt_error_td  my_fill_data_model_cbak_fset(
            const fzpl_fset_def_ptr    fset_def,
            fzpl_fset_td * const       fset )
{
      fzrt_error_td                    err = FZRT_NOERR;
      fz_ffmt_cbak_data_model_fset     *data_model_funcs;

      err = _fset_glue->fzpl_fset_def_check ( fset_def,
                        FZ_FFMT_CBAK_DATA_MODEL_FSET_VERSION,
                        FZPL_TYPE_STRING(fz_ffmt_cbak_data_model_fset),
                        sizeof ( fz_ffmt_cbak_data_model_fset ),
                        FZPL_VERSION_OP_NEWER );
      if ( err == FZRT_NOERR )
      {
            data_model_funcs = (fz_ffmt_cbak_data_model_fset *)fset;
```

```
            data_model_funcs->fz_ffmt_cbak_data_model_read_dlog_cust    = my_read_dlog_cust;
            data_model_funcs->fz_ffmt_cbak_data_model_read_predloginit  = my_read_predloginit;
            data_model_funcs->fz_ffmt_cbak_data_model_read_opts_default = my_read_opts_default;
            data_model_funcs->fz_ffmt_cbak_data_model_read_opts_flags   = my_read_opts_get_flags;
            data_model_funcs->fz_ffmt_cbak_data_model_read_opts_changed = my_read_opts_changed;

            data_model_funcs->fz_ffmt_cbak_data_model_read              = my_model_read;

            data_model_funcs->fz_ffmt_cbak_data_model_write_dlog_cust   = my_write_dlog_cust;
            data_model_funcs->fz_ffmt_cbak_data_model_write_predloginit = my_write_predloginit;
            data_model_funcs->fz_ffmt_cbak_data_model_write_opts_default = my_write_opts_default;
            data_model_funcs->fz_ffmt_cbak_data_model_write_opts_flags  = my_write_opts_get_flags;
            data_model_funcs->fz_ffmt_cbak_data_model_write_opts_changed = my_write_opts_changed;

            data_model_funcs->fz_ffmt_cbak_data_model_write_begin       = my_write_begin;
            data_model_funcs->fz_ffmt_cbak_data_model_write_end         = my_write_end;
            data_model_funcs->fz_ffmt_cbak_data_model_write_grup_begin = my_write_group_begin;
            data_model_funcs->fz_ffmt_cbak_data_model_write_grup_end    = my_write_group_end;
            data_model_funcs->fz_ffmt_cbak_data_model_write_points      = my_write_points;
            data_model_funcs->fz_ffmt_cbak_data_model_write_lines       = my_write_lines;
            data_model_funcs->fz_ffmt_cbak_data_model_write_polylines   = my_write_polylines;
            data_model_funcs->fz_ffmt_cbak_data_model_write_faces       = my_write_faces;
            data_model_funcs->fz_ffmt_cbak_data_model_write_objt        = my_write_objt;
            data_model_funcs->fz_ffmt_cbak_data_model_write_smod_solid  = my_write_smod_solid;
            data_model_funcs->fz_ffmt_cbak_data_model_write_smod_trimmed_surf =
                                                                   my_write_smod_trimmed_surf;
            data_model_funcs->fz_ffmt_cbak_data_model_write_can_do_smooth =
                                                                   my_write_can_do_smooth;
            data_model_funcs->fz_ffmt_cbak_data_model_write_ctrl        = my_write_ctrl;
            data_model_funcs->fz_ffmt_cbak_data_model_write_can_do_ctrl = my_write_can_do_ctrl;
            data_model_funcs->fz_ffmt_cbak_data_model_write_err_label   = my_write_err_label;
            data_model_funcs->fz_ffmt_cbak_data_model_write_units_conv  = my_write_units_conv;
            data_model_funcs->fz_ffmt_cbak_data_model_write_symb_def_start =
                                                                   my_write_symb_def_start;
            data_model_funcs->fz_ffmt_cbak_data_model_write_symb_def_end =
                                                                   my_write_symb_def_end;

            data_model_funcs->fz_ffmt_cbak_data_model_tmap_list         = my_tmap_list;
            data_model_funcs->fz_ffmt_data_cbak_model_tform_opts_changed =
                                                                   my_tform_opts_changed;
      }

      return(err);
}
```

Surface styles and texture maps

```
fzrt_error_td fz_ffmt_cbak_data_model_tmap_list
                          fz_ffmt_ref_td            ffmt_id,
                          fz_ffmt_ref_td            *fmt_list,
                          long                *fmt_list_knt_ptr )
```

Model files typically store bitmap texture data in one of two ways, a separate bitmap image file which the model file references, or bitmap pixels stored within the model file. If the model file stores bitmap textures in separate files, the data model file translator need to implement the `fz_ffmt_data_cbak_model_tmap_list` function to tell **form•Z** what bitmap file formats the model format supports. If this function is not implemented, **form•Z** will assume that the bitmap pixels are stored in the model file. This will disable the Save Texture Maps As menu on the Texture Map Import Options dialog and the Image File Format menu and Options… button on the Wrapped Texture Options and the Rendered Texture Options dialogs. If the `fz_ffmt_data_cbak_model_tmap_list` function is implemented, **form•Z** will call it twice. The first time will be to get a count of the supported bitmap formats. In this case the `fmt_list` parameter will be set to `NULL`. **form•Z** will then allocate memory for the array of bitmap formats and call the `fz_ffmt_data_cbak_model_tmap_list` function a second time. This time with `fmt_list` set to the array to fill. **form•Z** can manage texture bitmap file format conversions as specified by the texture options.

The example below shows the `fz_ffmt_data_cbak_model_tmap_list` function.

```
fzrt_error_td  my_tmap_list(
                           fz_ffmt_ref_td             ffmt_id,
                           fz_ffmt_ref_td             *fmt_list,
                           long                 *fmt_list_knt_ptr )
{
       long                       knt, i, num, max;
       fz_ffmt_ref_td             *ref_ids = NULL;
       long                       num_ids = 0;
       fzrt_error_td              err = FZRT_NOERR;

       knt = 0;
       max = 0;

       if(fz_ffmt_keyword_to_ref_id_list(TIFF_PLUGIN_KEYWD, NULL, &num))
       {
              knt += num;
              if(num > max) max = num;
       }
       if(fz_ffmt_keyword_to_ref_id_list(TGA_PLUGIN_KEYWD, NULL, &num))
       {
              knt += num;
              if(num > max) max = num;
       }

       if(max > 0)
       {
              ref_ids = (fz_ffmt_ref_td *)fzrt_new_ptr(max * sizeof(fz_ffmt_ref_td));
              num_ids = max;
       }

       if(wave_list != NULL && ref_ids != NULL)
       {
              knt = 0;
              if(fz_ffmt_keyword_to_ref_id_list(TIFF_PLUGIN_KEYWD, ref_ids, &num))
              {
                     for(i = 0; i < num; i++)
                     {
                            fmt_list[knt + i] = ref_ids[i];
                     }
                     knt += num;
              }
              if(fz_ffmt_keyword_to_ref_id_list(TGA_PLUGIN_KEYWD, ref_ids, &num))
              {
                     for(i = 0; i < num; i++)
                     {
                            fmt_list[knt + i] = ref_ids[i];
                     }
                     knt += num;
              }
       }

       if(fmt_list_knt_ptr != NULL)      *fmt_list_knt_ptr = knt;
       if(ref_ids != NULL) fzrt_dispose_ptr((fzrt_ptr)ref_ids);

       return(err);

}
```

If desired, **form•Z** can render procedural textures to bitmap images. In doing so, a bitmap file is created for each model face that is textured with a procedural texture.

To manage file format conversions and rendered textures, **form•Z** provides two tables. The file table which maps a source texture file to the converted file, the destination texture file. And, the style table which stores a mapping from surface styles associated with objects and faces with entries in the file table. **form•Z** also converts all the different shader parameters to a consistent representation consisting of ambient, diffuse and specular colors. Data model file translators can use these parameters or they can access the shader parameters directly through functions in the `fz_rmtl_fset` function set.

For textures the style table stores three indices into the texture file table, an index for a color texture, an index for a transparency texture, and an index for a bump map texture. Negative file indices represent rendered textures, positive file indices represent bitmap textures stored in the surface styles palette. A file indx of 0 means no texture.

The style and texture file tables are available for both import and export. For import, These tables are prefilled by **form•Z** with the surface styles in the surface style palette and any rendered textures. For export, these tables are initialized but are empty. When reading texture and surface style data from the file, the translator can add entries to these tables. When adding an entry, **form•Z** will search the table for duplicate entries. If a duplicate entry is found, The index of the matching entry is returned. If no duplicate entry is found, a new entry is created and the index of that entry is returned. **form•Z** will convert the entries in the style and textutre file tables into surface styles in the surface style palette. Thus the translator does not have to worry about creating duplicate entries in the surface style palette. Of course the translator has the option of creating entries in the surface style palette directly using function in the `fz_rmtl_fset` function set.

The example below shows how to retrieve data from the style and texture file tables for export by looping over all the styles.

```
	fzrt_error_td				err = FZRT_NOERR;
	fz_ffmt_tmap_style_ptr			style;
	fz_ffmt_tmap_file_ptr				file;
	fz_ffmt_tmap_filetab_ptr		filetab;
	fz_ffmt_tmap_styletab_ptr		styletab;
	char					rmtl_name[64];
	char					color_name[128];
	char					transp_name[128];
	char					bmap_name[128];
	double					ambient, diffuse, specular;
	double					transp, spec_expo, refr;
	fz_rgb_float_td				diff_col, spec_col;
	float					bmp_amp;
	long					color, trans, bump;
	fzrt_floc_ptr				floc;
	long					num, sindx, num_styles;

	fzrt_file_floc_init(&floc);

	err = fz_ffmt_data_model_styletab(ffmt_id, &styletab);
	if(err == FZRT_NOERR)
	{
		err = fz_ffmt_data_model_tmap_filetab(ffmt_id, &filetab);
		if(err == FZRT_NOERR)
		{
			fz_ffmt_data_model_styletab_count(styletab, &num);
			for (sindx = 0, num_styles = num; sindx < num_styles; ++sindx)
			{
				if (fz_ffmt_data_model_styletab_entry(styletab,
								sindx + 1, &style))
				{
					fz_ffmt_data_model_styletab_surf_name(style, rmtl_name,
64);
					fz_ffmt_data_model_styletab_ambient(style, &ambient);
					fz_ffmt_data_model_styletab_diffuse(style, &diffuse);
					fz_ffmt_data_model_styletab_diffuse_color(style,
&diff_col);
					fz_ffmt_data_model_styletab_specular(style, &specular);
```

```
                    fz_ffmt_data_model_styletab_specular_color(
                                  style, &spec_col);
                    fz_ffmt_data_model_styletab_transparency(style, &transp);
                    fz_ffmt_data_model_styletab_spec_expo(style, &spec_expo);
                    fz_ffmt_data_model_styletab_refraction(style, &refr);

                    /* Color Texture Map */
                    fz_ffmt_data_model_styletab_color_file_id(style, &color);
                    if (color != 0)
                    {
                           if(fz_ffmt_data_model_tmap_filetab_entry(filetab,
                                                                    color,
                                                                    &file))
                           {
                                  /* Get the file name of the destination
                                         (converted) bitmap file */
                                  fz_ffmt_data_model_tmap_filetab_dst_floc(
                                                       file,floc);
                                  fzrt_file_floc_get_name(
                                                floc, color_name, 128);
                           }
                    }
                    /* Transparency Texture Map */
                    fz_ffmt_data_model_styletab_transparency_file_id(
                                                style, &trans);
                    if (trans != 0)
                    {
                           if(fz_ffmt_data_model_tmap_filetab_entry(filetab,
                                                                    trans,
                                                                    &file))
                           {
                                  /* Get the file name of the destination
                                         (converted) bitmap file */
                                  fz_ffmt_data_model_tmap_filetab_dst_floc(
                                                       file,floc);
                                  fzrt_file_floc_get_name(
                                         floc,transp_name,128);
                           }
                    }
                    /* Bump Map Texture Map */
                    fz_ffmt_data_model_styletab_bmap_file_id(style, &bump);
                    if (bump != 0)
                    {
                           if(fz_ffmt_data_model_tmap_filetab_entry(filetab,
                                                                    bump,
                                                                    &file))
                           {
                                  /* Get the file name of the destination
                                         (converted) bitmap file */
                                  fz_ffmt_data_model_tmap_filetab_dst_floc(
                                                       file, floc);
                                  fzrt_file_floc_get_name(floc, bmap_name, 128);
                                  fz_ffmt_data_model_styletab_bmap_amplitude(
                                                style,&bmp_amp);
                           }
                    }
             }
      }
   }
}
fzrt_file_floc_finit(&floc);
```

The example below shows how to retrieve data from the style and texture file tables for export of a specific object or face.

```
fzrt_error_td              err = FZRT_NOERR;
fz_ffmt_tmap_styletab_ptr  styletab;
```

```
    fzrt_boolean                do_textures = TRUE;
    long                        cindx = -1;

    err = fz_ffmt_data_model_styletab(ffmt_id, &styletab);
    if(err == FZRT_NOERR)
    {
        err = fz_ffmt_data_model_sid_from_face(windex, ffmt_id, styletab,
                                            obj, face_indx, &style);
    }
```

Import options

Data model import translators can display an import options dialog. The "Options..." button on the import standard file Open dialog will be enabled if the options flags set by the `fz_ffmt_cbak_data_model_read_opts_flags` function has the `FZ_FFMT_OPTS_INIT_HAS_READ_OPTS_BIT` bit set. Individual items in the common section of the options dialog are enabled by setting the appropriate bits of the flags parameter to `fz_ffmt_cbak_data_model_read_opts_flags`. The appropriate bits are defined in `fz_ffmt_data_model_read_iface_opts_flags_enum`. Data model import options are discussed in section 3.13.1 of the **form•Z** Users Manual.

The call back functions to import model data are defined in the `fz_cbak_ffmt_data_model_fset`.

The `fz_ffmt_cbak_data_model_fset` contains the following functions to support data model import options:

The data model translator import options dialog enable function (required)

```
fzrt_error_td  fz_ffmt_cbak_data_model_read_opts_flags (
            fz_ffmt_ref_td              ffmt_id,
            long                *flags,
            long                *opts_flags
            );
```

This function is called by **form•Z** to get the enable state of the data model import "Options..." button and the enable states of each item in the image import options dialog. The `flags` parameter is used to set the enable states of items in the common section of the options dialog. Appropriate bits for this parameter are defined in `fz_ffmt_data_model_read_iface_opts_flags_enum`. By default, all items are disabled. The `opts_flags` parameter enables the "Options..."button on the "Image Import" standard file Open dialog by setting it to `FZ_FFMT_OPTS_INIT_HAS_READ_OPTS_BIT`. If the "Options..." button is to be disabled, `opts_flags` should be set to `0` (this is the default).

An example of an data model translator import options dialog enable function is shown below.

```
fzrt_error_td  my_read_opts_get_flags (
            fz_ffmt_ref_td              ffmt_id,
            long *              flags,
            long *              opts_flags )
{
    fzrt_error_td       err = FZRT_NOERR;

    FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLEFORMZUNITS_BIT);
    FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLEJOINCOPLANAR_BIT);

    FZ_SETBIT(*opts_flags, FZ_FFMT_OPTS_INIT_HAS_READ_OPTS_BIT);

    return(err);
}
```

The data model translator import options defaults function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_read_opts_default (
             fz_ffmt_ref_td            ffmt_id
             );
```

This function is called by **form•Z** to set default values of options. All custom options and any common options whose default values the file translator wishes to change must be set here. This function is only needed if the translator has custom options or the translator needs to change default values of any of the common options.

form•Z will have set the default values for common options prior to calling this function. This function can then change any of those values by calling `fz_ffmt_data_model_read_opts_parm_set` using the options pointer obtained from `fz_ffmt_data_model_read_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_data_model_read_opts_parm_get`.

An example of an data model translator import options defaults function is shown below.

```
long   my_read_opts_flags;

fzrt_error_td my_read_opts_default (
                    fz_ffmt_ref_td      ffmt_id )
{
       fzrt_error_td                    err = FZRT_NOERR;
       fz_type_td                       fz_type;
       fz_ffmt_read_color_meth_enum     color_meth = FZ_FFMT_READ_COLOR_METH_CURRENTCOLOR;

       /* Change a default value in the common options */
       fz_type_set_enum(&color_meth, &fz_type);
       fz_ffmt_data_model_read_opts_parm_set(
                          ffmt_id,
                          FZ_FFMT_DATA_MODEL_READ_OPTS_PARM_COLR_METHOD,
                          &fz_type);
       /* Set a default value for a custom option */
       my_read_opts_flags = 0;

       return(err);
}
```

The data model translator import options changed function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_read_opts_changed (
             fz_ffmt_ref_td                          ffmt_id,
             fz_ffmt_image_read_opts_enum            which
             );
```

This function is called by **form•Z** when the user changes the value of an option in the common section of the options dialog. This allows the translator to override the behavior of the common section of the options dialog by setting values of options or setting enable states of items. The `which` parameter specifies which parameter's value changed. Values of the common options can be set by calling `fz_ffmt_data_model_read_opts_parm_set` using the options pointer obtained from `fz_ffmt_data_model_read_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_data_model_read_opts_parm_get`. The enable states of items can be changed by first getting the enable flags by calling `fz_ffmt_data_model_read_get_dlog_flags`, then changing the enable bit of the item whose state needs to change and calling `fz_ffmt_data_model_read_set_dlog_flags`. Appropriate bits are defined in `fz_ffmt_data_model_read_iface_opts_flags_enum`. All these functions are in the `fz_ffmt_data_model_fset` function set.

The data model translator import custom options dialog function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_read_dlog_cust (
             fz_fuim_tmpl_ptr    fuim_tmpl,
```

```
            short               parent,
            fz_ffmt_ref_td              ffmt_id
            );
```

This function is called by **form•Z** to add items to the custom section of the options dialog. This function should add items by calling functions in the `fz_fuim_fset` function set using `fuim_mngr` and `parent` parameter (passed into this function) as the top level parent for all items. The `ffmt_id` parameter specifies the file translator's reference id.

An example of a data model translator import custom options dialog function is shown below.

```
#define MY_STRINGS          1
#define MY_COMPRESS_STR     2

#define MY_COMPRESS_BIT     1
long   my_read_opts_flags;

fzrt_error_td  my_read_dlog_cust (
            fz_fuim_tmpl_ptr    fuim_tmpl,
            short               parent,
            fz_ffmt_ref_td              ffmt_id )
{
      fzrt_error_td err = FZRT_NOERR;
      short         gindx;
      char          title[256];

      err = fzrt_fzr_get_string (_fz_rsrc_ref,MY_STRINGS,MY_COMPRESS_STR, title );
      if(err == FZRT_NOERR)
      {
            if((gindx = fz_fuim_new_check(fuim_tmpl, parent, 0,
                        FZ_FUIM_FLAG_GFLT | FZ_FUIM_FLAG_HORZ, title, NULL, NULL)) > -1)
            {
                  fz_fuim_item_encod_long(fuim_tmpl, gindx, & my_read_opts_flags,
                                    TRUE, FZ_FUIM_BIT2_MASK(MY_COMPRESS_BIT));
            }
      }

      return(err);
}
```

The data model translator import pre-options dialog function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_read_predloginit(
            long                windex,
            fz_ffmt_ref_td              ffmt_id
            );
```

This function is called by **form•Z** just prior to displaying the options dialog. This is done so the file translator can check the current state of **form•Z** and make any adjustments to the values of options or the enable states of items on the options dialog. For example, a translator may have a custom option that is only appropriate if the current view's projection is panormaic. In this case, this function would check the type of the current view and disable the option's dialog item if the view's projection is panormaic. Options values and item enable states can be changed as described in for the `fz_ffmt_data_model_read_opts_changed` function.

Import

Model data can only be imported into a **form•Z** model project. To accomplish this, **form•Z** simply calls the translator's read function. This function reads the contents of the file and constructs model objects using functions in `fz_model_fset`.

After importing all data in a file, **form•Z** will apply the following model import options to the imported data.

Transformation

form•Z Units
Format Units
Construct 3D Solids
Same Color Surfaces
Same Layer Surfaces
Join Adjacent Coplanar Faces

The application of other options is the responsibility of the translator.

The `fz_ffmt_cbak_data_model_fset` contains the following functions to support importing model data files:

The data model translator import read function

```
fzrt_error_td  fz_ffmt_cbak_data_model_read (
             long                windex,
             fz_ffmt_ref_td             ffmt_id,
             fzrt_floc_ptr       floc
             );
```

This function is called by **form•Z** to import the model data in a file. This function creates model objects from the file's contents by calling functions in the `fz_model_fset` function set. This function also imports lights, views, surface style, layers, etc. `floc` is the file to read.

A simple example of a data model translator import function which imports spheres is shown below.

```
fzrt_error_td  my_model_read (
             long                windex,
             fz_ffmt_ref_td             ffmt_id,
             fzrt_floc_ptr       floc )
{
      fzrt_error_td       err = FZRT_NOERR;
      my_file_td          file;
      double              cx, cy, cz, radius;
      fzrt_boolean        read_more_data = TRUE;
      fz_xyz_td           origin, radii;
      fz_xyz_td           rot = {0.0, 0.0, 0.0};
      fz_objt_ptr         obj;

      err = my_file_open(floc, &file);
      if(err == FZRT_NOERR)
      {
             while(read_more_data)
             {
                    err = my_file_read_sphere(&file, &cx, &cy, &cz, &radius,
                                                     &read_more_data);

                    if(err == FZRT_NOERR)
                    {
                           origin.x = cx;
                           origin.y = cy;
                           origin.z = cz;
                           radii.x = radius;
                           radii.y = radius;
                           radii.z = radius;
                           err = fz_objt_cnstr_sphr(windex,
                                                     &radii,
                                                     &origin,
                                                     NULL,
                                                     NULL,
                                                     NULL,
                                                     &obj );
                    }
             }

             if (err == FZRT_NOERR)
```

```
                err = my_file_close(&file);
        }

        return(err);
}
```

Symbol import

A **form•Z** symbol consists of a symbol definition. When a symbol is placed, a symbol instance is created. When importing files with symbols, the translator can call `fz_objt_symb_def_create` in the `fz_model_fset` to create a symbol definition. The function, `fz_objt_symb_ins_place` in the `fz_model_fset` is used to create a symbol instance. A symbol definition must be created before it can be instanced (placed).

Export options

Data model export translators can display an export options dialog. The "Options..." button on the export standard file Open dialog will be enabled if the options flags set by the `fz_ffmt_data_model_write_opts_flags` function has the `FZ_FFMT_OPTS_INIT_HAS_WRITE_OPTS_BIT` bit set. Individual items in the common section of the options dialog are enabled by setting the appropriate bits of the flags parameter to `fz_ffmt_data_model_write_opts_flags`. The appropriate bits are defined in `fz_ffmt_data_model_write_iface_opts_flags_enum`. Image export options are discussed in section 3.13.2 of the **form•Z** Users Manual.

The call back functions to export an image are defined in the `fz_ffmt_cbak_data_model_fset`.

The `fz_ffmt_cbak_data_model_fset` contains the following functions to support image export options:

The data model translator export options dialog enable function (required)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_opts_flags (
            fz_ffmt_ref_td              ffmt_id,
            long                *flags,
            long                *opts_flags
            );
```

This function is called by **form•Z** to get the enable state for the data model export "Options..." button and the enable states for each item on the image export options dialog. The `flags` parameter is used to set the enable states of items on the common section of the options dialog. Appropriate bits for this parameter are defined in `fz_ffmt_data_model_write_iface_opts_flags_enum`. By default, all items are disabled. The `opts_flags` parameter enables the "Options..."button on the "Export" standard file Open dialog by setting it to `FZ_FFMT_OPTS_INIT_HAS_WRITE_OPTS_BIT`. If the "Options..." button is to be disabled, `opts_flags` should be set to `0` (this is the default).

An example of an data model translator export options dialog enable function is shown below.

```
fzrt_error_td  my_write_opts_get_flags (
            fz_ffmt_ref_td              ffmt_id,
            long *              flags,
            long *              opts_flags )
{
        fzrt_error_td       err = FZRT_NOERR;

        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLEFORMZUNITS_BIT);
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLEFFMTUNITS_BIT);
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLECONSTRUCT3DSOLIDS_BIT);
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLESAMECOLOR_BIT);
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLESAMELAYER_BIT);
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLEJOINCOPLANAR_BIT     );
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLECOLOR_BIT);
        FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLEIMPORTTEXTUREMAPS_BIT);
```

```
	FZ_SETBIT(*flags, FZ_FFMT_DATA_MODEL_READ_IFACE_OPTS_ENABLETEXTUREOPTS_BIT);

	FZ_SETBIT(*opts_flags, FZ_FFMT_OPTS_INIT_HAS_WRITE_OPTS_BIT);

	return(err);
}
```

The data model translator export options defaults function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_opts_default (
		fz_ffmt_ref_td			ffmt_id
		);
```

This function is called by **form•Z** to set default values of options. All custom options and any common options whose default values the file translator wishes to change must be set here. This function is only needed if the translator has custom options or the translator needs to change default values of any of the common options.

form•Z will have set the default values for common options prior to calling this function. This function can then change any of those values by calling `fz_ffmt_data_model_write_opts_parm_set` using the options pointer obtained from `fz_ffmt_data_model_write_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_data_model_write_opts_parm_get`.

The data model translator export options changed function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_opts_changed (
		fz_ffmt_ref_td					ffmt_id,
		fz_ffmt_data_model_write_opts_enum		which
		);
```

This function is called by **form•Z** when the user changes the value of an option in the common section of the options dialog. This allows the translator to override the behavior of the common section of the options dialog by setting values of options or setting enable states of items. The which parameter specifies which parameter's value changed. Values of the common options can be set by calling `fz_ffmt_data_model_write_opts_parm_set` using the options pointer (`fz_ffmt_data_model_write_opts_ptr`) obtained from `fz_ffmt_data_model_write_opts_get_ptr`. If the translator needs to inspect the value of an option, it can be obtained by calling `fz_ffmt_data_model_write_opts_parm_get`. The enable states of items can be changes by first getting the enable flags by calling `fz_ffmt_data_model_write_get_dlog_flags`, then changing the enable bit of the item whose state needs to change and calling `fz_ffmt_data_model_write_set_dlog_flags`. Appropriate bits are defined in `fz_ffmt_data_model_write_iface_opts_flags_enum`. All these functions are in the `fz_ffmt_data_model_fset` function set.

The data model translator export custom options dialog function (optional)

```
fzrt_error_td  ffmt_cbak_data_model_write_dlog_cust (
		fz_fuim_tmpl_ptr	fuim_tmpl,
		short			tindx,
		short			parent,
		fz_ffmt_ref_td			ffmt_id
		);
```

This function is called by **form•Z** to add items to the custom section of the options dialog. This function should add items by calling functions in the `fz_fuim_fset` function set using the `fz_fuim_mngr_ptr` and `parent` parameter (passed into this function) as the top level parent for all items. The `ffmt_id` parameter specifies the file translator's reference id.

An example of a data model translator export custom options dialog function is shown below.

```
#define MY_STRINGS		1
#define MY_COMPRESS_STR		2
```

```
#define MY_READ_SUB_IMAG_BIT 1
long   my_write_opts_flags;

fzrt_error_td  my_write_dlog_cust (
             fz_fuim_tmpl_ptr    fuim_tmpl,
             short               parent,
             fz_ffmt_ref_td      ffmt_id )
{
       fzrt_error_td       err = FZRT_NOERR;
       short               gindx;
       char                title[256];

       err = fzrt_fzr_get_string (
                    _fz_rsrc_ref,
                    MY_STRINGS,
                    MY_COMPRESS_STR,
                    title );

       if(err == FZRT_NOERR)
       {
             if((gindx = fz_fuim_new_check(fuim_tmpl, parent, 0,
                          FZ_FUIM_FLAG_GFLT | FZ_FUIM_FLAG_HORZ, title, NULL, NULL)) > -1)
             {
                    fz_fuim_item_encod_long(fuim_tmpl, gindx, & my_write_opts_flags,
                                        TRUE, FZ_FUIM_BIT2_MASK(MY_READ_SUB_IMAG_BIT));
             }
       }

       return(err);
}
```

The data model translator export pre-options dialog function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_predloginit (
             long                windex,
             fz_ffmt_ref_td             ffmt_id,
             );
```

This function is called by **form•Z** just prior to displaying the options dialog. This is done so the file translator can check the current state of **form•Z** and make any adjustments to the values of options or the enable states of items on the options dialog. For example, a translator may have a custom option that is only appropriate if the project's units are English and not appropriate for Metric units. In this case, this function would check the type of the project's units and disable the option's dialog item if the units is Metric. Options values and item enable states can be changed as described in for the fz`_ffmt_data_model_read_opts_changed` function.

Export

form•Z only exports model objects to model files. The export process is more structured than the import process. **form•Z** calls several translator functions to export a model data.

Normally **form•Z** exports the contents of a project to one file. However if the Grouping Method option is not set to Single Group and the Separate Files option is set, the project may be split into several files. To support grouping and file splitting, **form•Z** calls several export functions as follows.

```
for each file
      Begin file export (open the file, write file header, etc)
            for each group
                  Begin group export
                        loop over all objects in group
                              export object
                  End group export
```

End file export (close file, etc.)

If the Grouping Method is set to By Color (which really means by surface style), **form•Z** will split objects with multiple face surface styles into multiple objects prior to export.

If the Grouping Method is set to By Group or By Layer and the translator supports hierarchial grouping (it can represent **form•Z**'s nested groups/layer groups), then the Begin group export may happen several times before an End group export happens. A call to the Begin group export function followed by another call to the Begin group function represents a parent child relationship for groups. The first call represents the parent group of the second call. Begin group exports move deeper down the grouping tree and End grouping functions move up. Begins and Ends will be ballanced.

If the Grouping Method is set to By Group or By Layer and the translator does not supports hierarchial grouping, the Begin group export End group export pair will be called once for each group or for each Layer in the project.

If the Grouping Method is set to Single Group, The Begin file export/End file export and Begin group export/End group export pairs will each be call only once.

When exporting an object, **form•Z** looks at both the object type and the export options to determine how the object should be exported. The export of each object is as follows.

```
exported = false
If the object is a controlled object and the Controlled Objects export option is As Parametric Data
    If the translator can export this parametric object
        (The translator needs to check the specific controlled object type (sphere, NURBS, etc.))
            Export the object as a controlled object
            exported = true
If not exported and object is a smooth object and the Smooth export option is not set to Facetted
    If the translator can export this smooth object
        If the Smooth export option is set to As Smooth Solids
            Export the object as a smooth solid
            exported = true
        Else if the Smooth export option is set to As Trimmed Surfaces
            Export the object as a trimmed surface
            exported = true
If not exported
    If the Facetted option is set to As Object and the object is not points or open wire
        If the translator has implemented an object export function
            Export the object as a facetted object
            exported = true
    If not exported or the Facetted option is set to As Faces and the object is not points or open wire
        If the translator has implemented a faces export function
            Export the object as individual faces
            exported = true
    If not exported or if the Facetted option is set to As Polylines and the object is not points
        If the translator has implemented a polylines export function
            Export the object as polylines
            exported = true
    If not exported or if the Facetted option is set to As Lines and the object is not points
        If the translator has implemented a lines export function
            Export the object as individual line segments
            exported = true
    If not exported or if the Facetted option is set to As Points
        If the translator has implemented a points export function
            Export the object as individual points
            exported = true
```

In addition to objects, other data in a **form•Z** project (i.e. lights, views and layers) can be exported as well although in a less structured way. Light data, view data and layer data can be access by functions in the `fz_lite_fset`, `fz_view_fset`, and `fz_layr_fset` function sets respectively. It is up to the translator to access this data at a place in the export process that is appropriate for the file format; typically one of the begin or end functions.

Prior to export, **form•Z** will transform all Lights, views, texture mappings and objects by a combination of the transform defined in the transform options and any units conversion that needs to take place. Also, prior to export, **form•Z** will decompose all objects as specified by the Decomposition options.

Export method menu items enable states for Plain Objects, Smooth, & Controlled Objects are controlled by existance of the following functions.

- Plain Objects - Facetted - As Points - `fz_ffmt_cbak_data_model_write_points`
- Plain Objects - Facetted - As Lines - `fz_ffmt_cbak_data_model_write_lines`
- Plain Objects - Facetted - As Polylines - `fz_ffmt_cbak_data_model_write_polylines`
- Plain Objects - Facetted - As Faces - `fz_ffmt_cbak_data_model_write_faces`
- Plain Objects - Facetted - As Objects - `fz_ffmt_cbak_data_model_write_objt`
- Plain Objects - Smooth - As Facetted - Any of the above functions
- Plain Objects - Smooth - As Smooth Solids - `fz_ffmt_cbak_data_model_write_smod_solid`
- Plain Objects - Smooth - As Trimmed Surfaces - `fz_ffmt_cbak_data_model_write_smod_trimmed_surf`
- Controlled Objects - As Plain Objects - Any of the above functions
- Controlled Objects - As Parametric Data - `fz_ffmt_cbak_data_model_write_ctrl`

Each item in the Plain Objects - Facetted menu corresponds to a facetted object's topological level as follows:

- point - point
- line - segment
- polyline - outline
- face - face
- object - object

Topological levels are described in section 4.0.1 of the **form•Z** Users manual. If the decomposition option, Connect Holes To Face Edged is set, each face will only contain one outline.

The `fz_ffmt_cbak_data_model_fset` contains the following functions to support writing model data files:

The data model translator export begin function

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_begin (
                    long                windex,
                    fz_ffmt_ref_td              ffmt_id,
                    void                **data_ptr,
                    fzrt_floc_ptr       floc
                    );
```

This function is called by **form•Z** to begin the export of a project. This function should allocate any memory required for parameters and working data. If the memory allocation fails an error should be returned.

```
typedef struct
{
       my_file_td   file;
}  my_trans_data_td;

fzrt_error_td  my_write_begin (
             long                windex,
             fz_ffmt_ref_td              ffmt_id,
             void                **data_ptr,
             fzrt_floc_ptr       floc )
{
       fzrt_error_td       err = FZRT_NOERR;
       my_trans_data_td    *my_data = NULL;
```

```
        *data_ptr = NULL;
        my_data = (my_trans_data_td *)fzrt_new_ptr_clear(sizeof(my_trans_data_td));
        if(my_data == NULL)
        {
                err = fzrt_error_set (
                                        FZRT_MALLOC_ERROR,
                                        FZRT_ERROR_SEVERITY_ERROR,
                                        FZRT_ERROR_CONTEXT_FZRT,  0 );
        }
        else
        {
                *data_ptr = my_data ;

                err = my_open(my_data);
                if(err == FZRT_NOERR)
                {
                        err = my_write_file_header(my_data);
                        if(err == FZRT_NOERR) err = my_write_views(my_data);
                        if(err == FZRT_NOERR) err = my_write_lights(my_data);
                        if(err == FZRT_NOERR) err = my_write_layers(my_data);
                        if(err == FZRT_NOERR) err = my_write_surface_styles(my_data);
                        if(err == FZRT_NOERR) err = my_write_symbols(windex, ffmt_id, my_data);

                        if(err != FZRT_NOERR)
                        {
                                my_file_close(my_data);
                        }
                }
        }
        return err;
}
```

The data model translator export end function

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_end (
                        long                    windex,
                        fz_ffmt_ref_td                  ffmt_id,
                        void                    **data_ptr,
                        fzrt_floc_ptr           floc,
                        fzrt_error_td           err
                        );
```

This function is called by **form•Z** to end the export of a project. `floc` is the file to open in the case the the Separate Files option is not set. If the Separate Files option is set, this is the name of the folder the files will be written to. `data_ptr` is a pointer to the translator data created in `fz_ffmt_cbak_data_model_write_begin`. `floc` is the file.

```
typedef struct
{
        my_file_td   file;
}  my_trans_data_td;

fzrt_error_td  my_write_end (
                long                    windex,
                fz_ffmt_ref_td                  ffmt_id,
                void                    **data_ptr,
                fzrt_floc_ptr           floc,
                fzrt_error_td           err
                )
{
        fzrt_error_td           err2 = FZRT_NOERR;
        my_trans_data_td        *my_data = *((my_trans_data_td **)data_ptr);

        if(err == FZRT_NOERR)       err = my_cleanup_after_err(my_data, err);
        if(my_data != NULL)
```

```
	{
		err2 = my_write_file_trailer(my_data);
		my_file_close(my_data);

		fzrt_dispose_ptr((fzrt_ptr)my_data);
		*data_ptr = NULL;
	}

	return(err2);
}
```

The data model translator export group begin function

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_grup_begin (
			long			windex,
			fz_ffmt_ref_td		ffmt_id,
			void			*data,
			long			cur_indx,
			char			*name
			);
```

This function is called by **form•Z** to designate the beginning of a group. `data` is a pointer to the translator data created in `fz_ffmt_cbak_data_model_write_begin`.

The data model translator export group end function

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_group_end (
			long			windex,
			fz_ffmt_ref_td		ffmt_id,
			void			*data,
			long			cur_indx,
			char			*name
			);
```

This function is called by **form•Z** to the end of a group. `data` is a pointer to the translator data created in `fz_ffmt_cbak_data_model_write_begin`.

The data model translator export error label function

```
fzrt_error_td  ffmt_cbak_data_model_write_err_label (
			long			windex,
			fz_ffmt_ref_td		ffmt_id,
			char			*str,
			long			max_len,
			short			*err_id
			);
```

If an error occurs when exporting an image, **form•Z** will display an error dialog indicating the error. This function is called by **form•Z** to obtain a string to display to the user. This is a general error message for all errors. Specific error strings are returned by the error string function registered with the plugin.

An example of a data model export error label function is shown below.

```
#define MY_STRINGS			1
#define MY_WRITE_ERR_STR		4

fzrt_error_td  my_write_err_label(
		long			windex,
		fz_ffmt_ref_td		ffmt_id,
		char			*str,
		long			max_len,
		short			*err_id )
{
	fzrt_error_td err = FZRT_NOERR;
```

```
	char str1[256];

	err = fzrt_fzr_get_string(_fz_rsrc_ref, MY_STRINGS, MY_WRITE_ERR_STR, str1);
	if(err == FZRT_NOERR)		strncpy(str, str1, max_len);

	return(err);
}
```

The data model translator export units conversion function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_units_conv (
			long				windex,
			fz_ffmt_ref_td			ffmt_id,
			fz_unit_type_enum		units_type,
			double				*conv_factor
			);
```

This function is called by **form•Z** with a scale factor, `conv_factor`, to convert positions in the **form•Z** project to appropriate units for the file format. For English projects, **form•Z** stores all positions in units of Inches. For Metric projects, **form•Z** stored all positions in units of Centimenters. For example, if the **form•Z** project is in English units (positions stored as inches) and the file format specifies that positions are stores in meters, `conv_factor` would be the multiplier to convert inches to meters. This function is not needed if the file format is unitless. In this case, **form•Z** uses a default `conv_factor` of 1.0. `units_type` specifies the units of the **form•Z** project.

An example of a data model export error label function is shown below for a file format which stores data in units of Meters.

```
fzrt_error_td  my_write_units_conv (
			long				windex,
			fz_ffmt_ref_td			ffmt_id,
			fz_unit_type_enum		units_type,
			double				*conv_factor )
{
	fzrt_error_td err = FZRT_NOERR;

	if(units_type == FZ_UNIT_TYPE_ENGLISH)
	{
		*conv_factor = 0.0254000508;	/* Inches to Meters */
	}
	else
	{
		*conv_factor = 0.01;		/* Centimeters to Meters */
	}

	return(err);
}
```

The data model translator export points function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_points (
			long			windex,
			fz_ffmt_ref_td		ffmt_id,
			void			*data,
			fz_objt_ptr		obj,
			fz_xyz_td		*vertex_normals,
			fz_xy_td		*vertex_texture_uvs
			);
```

This function is called by **form•Z** to export the geometry of an object as individual points. In other words, only the location, normals and texture coordinates of the objects vertices are exported. This is the only function which will export point objects and point cloud objects. `obj` is the object to export. The locations of the vertices can be obtained by calling `fz_objt_point_get_xyz` in the `fz_model_fset` function set. `vertex_normals` is an array of vertex normals which correspond to vertices in the object. The order of the normals in the array match the order of the vertices stored in the object. For instance, `vertex_normals[10]` is the normal for the point at the location filled by `fz_objt_point_get_xyz(windex, obj, 10, FZ_OBJT_MODEL_TYPE_FACT, &location)`. `vertex_texture_uvs` are the uv texture coordinates which correspond to vertices in the object. They are ordered the same as `vertex_normals`. This function is only needed if the file format supports point objects.

A simple example of a data model export points function is shown below.

```
fzrt_error_td  my_write_points (
                    long                windex,
                    fz_ffmt_ref_td      ffmt_id,
                    void                *data,
                    fz_objt_ptr         obj,
                    fz_xyz_td           *vertex_normals,
                    fz_xy_td            *vertex_texture_uvs )
{
        fzrt_error_td       err = FZRT_NOERR;
        long                num_verts;
        long                i;
        fz_xyz_td           location;
        my_trans_data_td    *my_data = (my_trans_data_td *)data;

        err = fz_objt_get_point_count(windex, obj, FZ_OBJT_MODEL_TYPE_FACT, &num_verts);
        if(err == FZRT_NOERR)
        {
                for(i = 0; i < num_verts && err == FZRT_NOERR; i++)
                {
                        err = fz_objt_point_get_xyz(windex, obj, 10, FZ_OBJT_MODEL_TYPE_FACT,
                                                          &location);
                        if(err == FZRT_NOERR)
                        {
                                err = my_write_vertex(my_data, location.x, location.y, location.x,
                                                vertex_normals[i].x,
                                                vertex_normals[i].y,
                                                vertex_normals[i].z,
                                                vertex_texture_uvs[i].x, vertex_texture_uvs[i].y);
                        }
                }
        }

        return(err);
}
```

The data model translator export lines function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_lines (
                    long                windex,
                    fz_ffmt_ref_td      ffmt_id,
                    void                *data,
                    fz_objt_ptr         obj,
                    fz_xyz_td           *vertex_normals,
                    fz_xy_td            *vertex_texture_uvs
                    );
```

This function is called by **form•Z** to export the geometry of an object as individual line segments. `obj` is the object to export. The locations of the vertices can be obtained described in the `fz_ffmt_data_model_write_points` function. `vertex_normals` and `vertex_texture_uvs` parameters are as described `ffmt_data_model_write_points`. Which points form line segments can be determined from calls to `fz_objt_segt_get_start_pindx` and `fz_objt_segt_get_end_pindx` functions in the `fz_model_fset` function set. This function is only needed if the file format supports line segment objects.

A simple example of a data model export lines function is shown below.

```
fzrt_error_td  my_write_lines (
```

```
                    long                windex,
                    fz_ffmt_ref_td      ffmt_id,
                    void                *data,
                    fz_objt_ptr         obj,
                    fz_xyz_td           *vertex_normals,
                    fz_xy_td            *vertex_texture_uvs )
{
        fzrt_error_td       err = FZRT_NOERR;
        long                num_verts;
        long                num_lines;
        long                i;
        long                start, end;
        fz_xyz_td           location;
        my_trans_data_td    *my_data = (my_trans_data_td *)data;

        /* Export Vertices */
        err = fz_objt_get_point_count (windex, obj, FZ_OBJT_MODEL_TYPE_FACT, &num_verts);
        if(err == FZRT_NOERR)
        {
                for(i = 0; i < num_verts && err == FZRT_NOERR; i++)
                {
                        err = fz_objt_point_get_xyz(windex, obj, 10, FZ_OBJT_MODEL_TYPE_FACT,
&location);
                        if(err == FZRT_NOERR)
                        {
                                err = my_write_vertex(my_data, location.x, location.y, location.x,
                                                vertex_normals[i].x, vertex_normals[i].y,
                                                vertex_normals[i].z,
                                                vertex_texture_uvs[i].x, vertex_texture_uvs[i].y);
                        }
                }
        }

        /* Export segments */
        if(err == FZRT_NOERR)
        {
                err = fz_objt_get_segt_count(windex, obj, FZ_OBJT_MODEL_TYPE_FACT,&num_lines);
                for(i = 0; i < num_lines && err == FZRT_NOERR; i++)
                {
                        err = fz_objt_segt_get_start_pindx(windex, obj, i,
                                                                        FZ_OBJT_MODEL_TYPE_FACT, &start);
                        if(err == FZRT_NOERR)
                        {
                                err = fz_objt_segt_get_end_pindx(windex, obj, i,
                                                                        FZ_OBJT_MODEL_TYPE_FACT, &end);
                        }
                        if(err == FZRT_NOERR)
                        {
                                err = my_write_line(my_data, start, end);
                        }
                }
        }

        return(err);
}
```

The data model translator export polylines function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_polylines (
                    long                windex,
                    fz_ffmt_ref_td      ffmt_id,
                    void                *data,
                    fz_objt_ptr         obj,
                    fz_xyz_td           *vertex_normals,
                    fz_xy_td            *vertex_texture_uvs
                    );
```

This function is called by **form•Z** to export the geometry of an object as a connected set of line segments. `obj` is the object to export. The locations of the vertices can be obtained described in the `fz_ffmt_data_model_write_points` function. `vertex_normals` and `vertex_texture_uvs` parameters are as described. Which points form line segments can be determined frim calls to `fz_objt_segt_get_start_pindx` and `fz_objt_segt_get_end_pindx` functions in the `fz_model_fset` function set. This function is only needed if the file format supports polyline connected objects.

A simple example of a data model export polylines function is shown below.

```
fzrt_error_td  my_write_polylines (
                    long                    windex,
                    fz_ffmt_ref_td          ffmt_id,
                    void                    *data,
                    fz_objt_ptr             obj,
                    fz_xyz_td               *vertex_normals,
                    fz_xy_td                *vertex_texture_uvs )
{
        fzrt_error_td           err = FZRT_NOERR;
        long                    num_verts, num_lines, num_faces, num_curvs;
        long                    i,j,k;
        long                    beg_seg_index, crv_index;
        fz_xyz_td               location;
        long*                   pnts;
        my_trans_data_td        *my_data = (my_trans_data_td *)data;

        /* Export Vertices */
        err = fz_objt_get_point_count (windex, obj, FZ_OBJT_MODEL_TYPE_FACT, &num_verts);
        if(err == FZRT_NOERR)
        {
                for(i = 0; i < num_verts && err == FZRT_NOERR; i++)
                {
                        err = fz_objt_point_get_xyz(windex, obj, 10, FZ_OBJT_MODEL_TYPE_FACT,
                                                    &location);
                        if(err == FZRT_NOERR)
                        {
                                err = my_write_vertex(my_data, location.x, location.y, location.x,
                                                vertex_normals[i].x, vertex_normals[i].y,
                                                vertex_normals[i].z,
                                                vertex_texture_uvs[i].x, vertex_texture_uvs[i].y);
                        }
                }
        }

        /* allocate memory for pnts array */
        . . .

        /* Export polylines */
        if(err == FZRT_NOERR)
        {
                err = fz_objt_get_face_count(windex, obj, FZ_OBJT_MODEL_TYPE_FACT,&num_faces);
                if(err == FZRT_NOERR)
                {
                        for(i = 0; i < num_faces && err == FZRT_NOERR; i++)
                        {
                                fz_objt_face_get_curv_count(windex, obj, i,
                                                        FZ_OBJT_MODEL_TYPE_FACT,&num_curvs);
                                err = fz_objt_face_get_cindx(windex, obj, i,
                                                FZ_OBJT_MODEL_TYPE_FACT, &crv_index);

                                for(j = 0; j < num_curvs && err == FZRT_NOERR; j++)
                                {
                                        fz_objt_curv_get_segt_count(windex, obj, crv_index,
                                                FZ_OBJT_MODEL_TYPE_FACT, &num_lines)

                                        fz_objt_curv_get_sindx(windex, obj, crv_index,
```

```
                                        FZ_OBJT_MODEL_TYPE_FACT, &beg_seg_index);

                            k = beg_seg_index;
                            while(err == FZRT_NOERR)
                            {
                                   err = fz_objt_segt_get_start_pindx(windex, obj,
                                          k, FZ_OBJT_MODEL_TYPE_FACT, &pnts[k]);

                                   fz_objt_segt_get_next(windex, obj, k,
                                          FZ_OBJT_MODEL_TYPE_FACT, &k);

                                   if (k == beg_seg_index || /* at beg again */
                                       k == -1 /* open line */)
                                   {
                                          break;
                                   }
                            }
                            if(err == FZRT_NOERR)
                            {
                                   err = my_write_polyline(my_data, pnts);
                            }
                            fz_objt_curv_get_next(windex, obj, crv_index,
                                   FZ_OBJT_MODEL_TYPE_FACT, &crv_index);
                     }
              }
       }
       }

       /* deallocate memory for pnts array */
       . . .

       return(err);
}
```

The data model translator export faces function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_faces (
                     long                windex,
                     fz_ffmt_ref_td      ffmt_id,
                     void                *data,
                     fz_objt_ptr         obj,
                     fz_xyz_td           *face_normals,
                     fz_xyz_td           *vertex_normals,
                     fz_xy_td            *vertex_texture_uvs
                     );
```

This function is called by **form•Z** to export the geometry of an object as individual faces. Faces are a collection of curves (outlines) which are made up of an ordered set of line segments (edges). The curves that make up a face can be accessed by the `fz_objt_face_get_cindx` function in the `fz_model_fset` function set. If the Connect Holes To Face Edges option is set, each face will only have one curve. The segments that make up a curve can be accessed by the `fz_objt_curv_get_sindx` function in the `fz_model_fset` function set. Determining which points form line segments is described in the `fz_ffmt_data_model_write_lines` function. `face_normals` is an array of face normals which corresponds to the faces in an object. The order of the normals in the array matches the order of the faces stored in the object. For instance, `face_normals[10]` is the normal for the face with index 10 of the object. This function is only needed if the file format supports faces of objects.

A simple example of a data model export faces function is shown below.

```
fzrt_error_td  my_write_faces (
                     long                windex,
                     fz_ffmt_ref_td      ffmt_id,
                     void                *data,
                     fz_objt_ptr         obj,
```

```
                    fz_xyz_td           *face_normals,
                    fz_xyz_td           *vertex_normals,
                    fz_xy_td            *vertex_texture_uvs )
{
        fzrt_error_td       err = FZRT_NOERR;
        long                num_verts, num_lines, num_faces, num_curvs;
        long                i, j, k;
        long                beg_seg_index, crv_index;
        fz_xyz_td           location;
        long*               pnts;
        my_trans_data_td    *my_data = (my_trans_data_td *)data;

        /* Export Vertices */
        err = fz_objt_get_point_count(windex, obj, FZ_OBJT_MODEL_TYPE_FACT, &num_verts);
        if(err == FZRT_NOERR)
        {
                for(i = 0; i < num_verts && err == FZRT_NOERR; i++)
                {
                        err = fz_objt_point_get_xyz(windex, obj, 10, FZ_OBJT_MODEL_TYPE_FACT,
                                                    &location);
                        if(err == FZRT_NOERR)
                        {
                                err = my_write_vertex(my_data, location.x, location.y, location.x,
                                                vertex_normals[i].x, vertex_normals[i].y,
                                                vertex_normals[i].z,
                                                vertex_texture_uvs[i].x, vertex_texture_uvs[i].y);
                        }
                }
        }

        /* allocate memory for pnts array */
        . . .

        /* Export faces */
        if(err == FZRT_NOERR)
        {
                err = fz_objt_get_face_count(windex, obj, FZ_OBJT_MODEL_TYPE_FACT,&num_faces);
                if(err == FZRT_NOERR)
                {
                        for(i = 0; i < num_faces && err == FZRT_NOERR; i++)
                        {
                                fz_objt_face_get_curv_count(windex, obj, i,
                                                        FZ_OBJT_MODEL_TYPE_FACT,&num_curvs);
                                err = fz_objt_face_get_cindx(windex, obj, i,
                                                FZ_OBJT_MODEL_TYPE_FACT, &crv_index);

                                /* first curve is outline of face, subsequent curves
                                   are holes in face */

                                for(j = 0; j < num_curvs && err == FZRT_NOERR; j++)
                                {
                                        fz_objt_curv_get_segt_count(windex, obj, crv_index,
                                                FZ_OBJT_MODEL_TYPE_FACT, &num_lines)

                                        fz_objt_curv_get_sindx(windex, obj, crv_index,
                                                FZ_OBJT_MODEL_TYPE_FACT, &beg_seg_index);

                                        k = beg_seg_index;
                                        while(err == FZRT_NOERR)
                                        {
                                                err = fz_objt_segt_get_start_pindx(windex, obj,
                                                        k, FZ_OBJT_MODEL_TYPE_FACT, &pnts[k]);

                                                fz_objt_segt_get_next(windex, obj, k,
                                                        FZ_OBJT_MODEL_TYPE_FACT, &k);

                                                if (k == beg_seg_index || /* at beg again */
                                                    k == -1 /* open line */)
```

```
                                        {
                                              break;
                                        }
                                  }
                                  if(err == FZRT_NOERR)
                                  {
                                        err = my_write_face_polyline(my_data, pnts);
                                  }
                                  fz_objt_curv_get_next(windex, obj, crv_index,
                                        FZ_OBJT_MODEL_TYPE_FACT, &crv_index);
                            }
                      }
                }
          }

          /* deallocate memory for pnts array */
          . . .

          return(err);
}
```

The data model translator export object function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_objt (
                    long                    windex,
                    fz_ffmt_ref_td          ffmt_id,
                    void                    *data,
                    fz_objt_ptr             obj,
                    fz_xyz_td               *face_normals,
                    fz_xyz_td               *vertex_normals,
                    fz_xy_td                *vertex_texture_uvs
                    );
```

This function is called by **form•Z** to export the geometry and topology of an object.

The data model translator export can do smooth function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_can_do_smooth (
                    long                                    windex,
                    fz_ffmt_ref_td                          ffmt_id,
                    void                                    *data,
                    fz_objt_ptr                             obj,
                    fzrt_boolean                            do_tmap,
                    fz_ffmt_data_model_write_smod_meth_enum *smod_method
                    );
```

This function is called by **form•Z** to determine if a specific smooth object can be exported as a smooth object.

The data model translator export trimmed surface function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_smod_trimmed_surf (
                    long                    windex,
                    fz_ffmt_ref_td          ffmt_id,
                    void                    *data,
                    fz_objt_ptr             obj,
                    fzrt_boolean            do_tmap
                    );
```

This function is called by **form•Z** to export an object as a trimmed surface.

The data model translator export smooth solid function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_smod_solid (
                    long                    windex,
                    fz_ffmt_ref_td          ffmt_id,
```

```
                    void                *data,
                    fz_objt_ptr         obj,
                    fzrt_boolean        do_tmap
                    );
```

This function is called by **form•Z** to export an object as a smooth solid.

The data model translator export can do controlled object function (optional)

```
fzrt_boolean  fz_ffmt_cbak_data_model_write_can_do_ctrl (
                    long                                    windex,
                    fz_ffmt_ref_td                          ffmt_id,
                    void                                    *data,
                    fz_objt_ptr                             obj,
                    fzrt_boolean                            do_tmap,
                    fz_ffmt_data_model_write_cntl_meth_enum cntl_method
                    );
```

This function is called by **form•Z** to determine if a controlled smooth object can be exported as a controlled object. This function should look at the object's specific type (text, sphere, cone, sweep, symbol instance, etc.). If the file format supports that specific object type, this function should return TRUE, otherwise it should return FALSE.

A simple example of a data model export can do controlled object function is shown below.

```
fzrt_boolean  my_write_can_do_ctrl(
                    long                                    windex,
                    fz_ffmt_ref_td                          ffmt_id,
                    void                                    *data,
                    fz_objt_ptr                             obj,
                    fzrt_boolean                            do_tmap,
                    fz_ffmt_data_model_write_cntl_meth_enum cntl_method )
{
      fzrt_boolean        rv = FALSE;
      fzrt_UUID_td        otype;

      if(cntl_method == FZ_FFMT_DATA_MODEL_WRITE_CNTL_AS_CNTL)
      {
            fz_objt_cntl_get_uuid(windex, obj, otype);

            if(fzrt_UUID_is_equal(otype, FZ_OBJT_TYPE_SPHR))
            {
                  rv = TRUE;
            }
      }

      return(rv);
}
```

In this example the file format only supports spheres as parametric objects. All other object types will be expored as plain objects.

The data model translator export controlled object function (optional)

```
fzrt_error_td  ffmt_cbak_data_model_write_ctrl (
                    long                                    windex,
                    fz_ffmt_ref_td                          ffmt_id,
                    void                                    *data,
                    fz_objt_ptr                             obj,
                    fzrt_boolean                            do_tmap
                    fz_ffmt_data_model_write_cntl_meth_enum cntl_method
                    );
```

This function is called by **form•Z** to export a controlled object.

A simple example of a data model export can do controlled object function is shown below.

```
fzrt_error_td  my_write_ctrl(
                    long                        windex,
                    fz_ffmt_ref_td              ffmt_id,
                    void                        *data,
                    fz_objt_ptr                 obj,
                    fzrt_boolean                do_tmap )
{
        fzrt_error_td       err = FZRT_NOERR;
        fzrt_UUID_td        otype;
        my_trans_data_td    *my_data = (my_trans_data_td *)data;
        fz_type_td          fz_type;
        fz_xyz_td           radii,origin;

        fz_objt_cntl_get_uuid (windex, obj, otype);

        if(fzrt_UUID_is_equal(otype, FZ_OBJT_TYPE_SPHR))
        {
                err = fz_objt_edit_sphr_parm_get(windex, obj,
                                                        FZ_OBJT_SPHR_PARM_RADII, &fz_type);
                if(err == FZRT_NOERR)
                {
                        fz_type_get_xyz(&fz_type, &radii);
                }
                err = fz_objt_edit_sphr_parm_get(windex, obj,
                                                        FZ_OBJT_SPHR_PARM_ORIGIN, &fz_type);
                if(err == FZRT_NOERR)
                {
                        fz_type_get_xyz(&fz_type, &origin);
                }
                if(err == FZRT_NOERR)
                {
                        err = my_write_sphere(my_data, origin.x, origin.y, origin.z,
                                                radii.x, radii.y, radii.z);
                }
        }

        return(err);
}
```

In this example the file format only supports spheres as parametric objects. All other object types will be expored as plain objects.

Symbol export

Symbols are described in section 4.20 of the **form•Z** Users Manual. To export symbols both the symbol definition and symbol instance need tp be exported. If the export Symbol Option Explode Symbols is selected, **form•Z** explodes each symbol instance prior to export. The symbols are then exported as any other object. For file formats which don't support object instancing, the Explode Symbols option should be set and the "Symbol Options…" button on the export options dialog should be disabled.

To export symbol definitions (when Explode Symbols is not selected) a translator needs to call `fz_ffmt_data_model_write_sdefs` from a place in the export process that's appropriate for the file format. Then for each symbol definition to be exported, **form•Z** will call the translator's `fz_ffmt_data_model_write_symb_def_begin`. This will let the translator know that a symbol is being exported and which symbol it is. Then for each object in the symbol, **form•Z** exports the object as it would any other object. At the conclusion of writing the objects in the symbol definition, **form•Z** calls the translator's `fz_ffmt_data_model_write_symb_def_end` to let the translator that export of a symbol definition is done.

Symbol definitions can contain lights. The lights can be accessed from either the translator's `fz_ffmt_data_model_write_symb_def_begin or fz_ffmt_data_model_write_symb_def_end` function. The `fz_symb_fset` function set has the functions, `fz_symb_lev_get_n_lights` which gets the number of lights in a symbol definition and `fz_symb_lev_get_light` to get a light from the symbol definition.

Symbol instanced are written at the same time as other objects. Symbol objects are controlled objects so a translator must implement the `fz_ffmt_cbak_data_model_write_can_do_ctrl` and `fz_ffmt_cbak_data_model_write_ctrl` functions. Symbol instances are identified by the `FZ_OBJT_TYPE_SYMB` model type.

The data model translator export symbol definition begin function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_symb_def_start (
                    long                windex,
                    fz_ffmt_ref_td      ffmt_id,
                    void                *data,
                    fz_symb_lib_ptr     lib_ptr,
                    fz_symb_def_ptr     def_ptr,
                    fz_symb_lev_ptr     lev_ptr
                    );
```

This function is called by **form•Z** to let a translator know that the export of a symbol definition is beginning.

```
typedef struct
{
      my_file_td   file;
}  my_trans_data_td;

static long   _seed = 0;    /* Used for generating unique names for symbol definitions */

/* This function is called from my_write_file_begin (previously implemented) */
fzrt_error_td  my_write_symbols(
                    long                windex,
                    fz_ffmt_ref_td      ffmt_id,
                    my_trans_data_td    *data)
{
      fzrt_error_td                     err = FZRT_NOERR;
      fzrt_boolean                      do_tmaps;
      long                              flags;
      fz_type_td                        fz_type;
      short                             grup_method, sym_method;
      short                             fact_method, smooth_method, cntl_method;
      my_trans_data_td                  *my_data = (my_trans_data_td *)data;
      fz_objt_tria_type_enum            triang_type;
      fzrt_boolean                      non_planar_only;
      fzrt_boolean                      strict_planarity;
      double                            angle;

      _seed = 0;

      fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
                              FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_GRUP_METHOD,
                              &fz_type);
      fz_type_get_short(&fz_type, &grup_method);

      fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
                              FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_FLAGS,
                              &fz_type);
      fz_type_get_long(&fz_type, &flags);
      do_tmaps = FZ_CHKBIT(flags,
                        FZ_FFMT_DATA_MODEL_WRITE_OPTS_TEXTUREMAPS_BIT) ? TRUE : FALSE;

      fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
                              FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_SYMB_METHOD,
                              &fz_type);
      fz_type_get_short(&fz_type, &sym_method);
```

```
	fz_ffmt_data_model_write_opts_get_dcomp_opts(ffmt_id, &triang_type, &non_planar_only,
							&strict_planarity);

	fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
				FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_FACT_METHOD,
				&fz_type);
	fz_type_get_short(&fz_type, &fact_method);
	fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
				FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_SMOD_METHOD,
				&fz_type);
	fz_type_get_short(&fz_type, &smooth_method);
	fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
				FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_CNTL_METHOD,
				&fz_type);
	fz_type_get_short(&fz_type, &cntl_method);

	fz_ffmt_data_model_write_opts_parm_get(ffmt_id,
				FZ_FFMT_DATA_MODEL_WRITE_OPTS_PARM_SMTH_ANG,
				&fz_type);
	fz_type_get_double(&fz_type, &angle);

	if (sym_method != FZ_FFMT_DATA_WRITE_SYMB_METH_SYMEXPLODE)
	{
		err = fz_ffmt_data_model_write_sdefs(
				windex,
				ffmt_id,
				sym_method,
				do_tmaps,
				grup_method,
				fact_method,
				smooth_method,
				cntl_method,
				angle,
				NULL,
				my_data,
				triang_type,
				non_planar_only,
				strict_planarity
				);
	}
	return(err);
}

/* The call to ffmt_3d_objt_write_sdefs  causes form·Z to iterate over all symbol definitions
and call this for each definition. */
fzrt_error_td  my_write_symb_def_start (
			long			windex,
			fz_ffmt_ref_td		ffmt_id,
			void			*data,
			fz_symb_lib_ptr		lib_ptr,
			fz_symb_def_ptr		def_ptr,
			fz_symb_lev_ptr		lev_ptr
			)
{
	fzrt_error_td		err = FZRT_NOERR;
	my_trans_data_td	*my_data = (my_trans_data_td *)data;
	char			name[256], unique_name[256];
	long			n_lights, j;
	fz_lite_ptr		light;

	err = fz_symb_def_get_name (windex, def_ptr, name);
	if(err == FZRT_NOERR)
	{
		sprintf(unique_name, "%s_%d ", name, _seed);
		err = my_write_symb_begin(my_data, unique_name);
		if(err == FZRT_NOERR)
		{
			err = fz_symb_lev_model_get_num_lights(windex, lev_ptr, &n_lights);
```

```
                    for(j = 0; j < n_lights && err == FZRT_NOERR; j++)
                    {
                          err = fz_symb_lev_get_light(windex, lev_ptr, j, &light);
                          if(err == FZRT_NOERR)
                          {
                                err = my_write_symb_light(my_data, light);
                          }
                    }
             }
      }

      return(err);
}
```

The data model translator export symbol definition end function (optional)

```
fzrt_error_td  fz_ffmt_cbak_data_model_write_symb_def_end (
                    long                windex,
                    fz_ffmt_ref_td             ffmt_id,
                    void                *data,
                    fz_symb_lib_ptr     lib_ptr,
                    fz_symb_def_ptr     def_ptr,
                    fz_symb_lev_ptr     lev_ptr
                    );
```

This function is called by **form•Z** to let a translator know that the export of a symbol definition has ended.

```
typedef struct
{
      my_file_td   file;
}  my_trans_data_td;

fzrt_error_td  my_write_symb_def_end (
                    long                windex,
                    fz_ffmt_ref_td             ffmt_id,
                    void                *data,
                    fz_symb_lib_ptr     lib_ptr,
                    fz_symb_def_ptr     def_ptr,
                    fz_symb_lev_ptr     lev_ptr
                    )
{
      fzrt_error_td       err = FZRT_NOERR;
      my_trans_data_td    *my_data = (my_trans_data_td *)data;

      err = my_write_symb_end(my_data);

      return(err);
}
```

2.8.4 Object types

In **form•Z**, there is a large number of object types, also called controlled objects. They are, for example, extrusions, enclosures, cubes, cones, cylinders, spheres, tori, sweeps, stairs etc. A controlled object stores its generation parameters in a data block that is maintained with the object. The parameters can be displayed in a dialog editing environment, which can be invoked form the Query dialog. The parameters of some controlled objects can also be edited graphically through the Edit Controls tool. It is possible to create custom object types in a plugin by registering a function set with a plugin class. The plugin with which the function set is registered is usually of type `FZ_OTYP_EXTS_TYPE`, but can be a different type as well. For example, a command or a file translator plugin may install the object type function set as part of the functionality added through the plugin. Multiple object type function sets may also be installed with a single plugin. This allows a plugin to offer a suite of object types, which logically belong together in a single package.

The function set which defines a custom object type is `fz_otyp_cbak_fset`. The example below shows the definition of a plugin of type `FZ_OTYP_EXTS_TYPE` and the registration of a single object type within that plugin. This object type defines star shaped objects. It will serve as an example throughout the remainder of this section. The source code for this plugin is available as an example as well. It is recommended to build this plugin with the respective compiler environment and trace the execution of the callback functions.

```
fzrt_error_td star_register_plugins ()
{
	fzrt_error_td	err = FZRT_NOERR;

	err = fzpl_glue->fzpl_plugin_register(
				STAR_OTYP_PLUGIN_UUID,
				"Star Object Type",
				STAR_OTYP_PLUGIN_VERSION,
				STAR_OTYP_PLUGIN_VENDOR,
				STAR_OTYP_PLUGIN_URL,
				FZ_OTYP_EXTS_TYPE,
				FZ_OTYP_EXTS_VERSION,
				star_error_str_func,
				0,
				NULL,
				&star_otyp_plugin_runtime_id);

	if ( err == FZRT_NOERR )
	{
		err = fzpl_glue->fzpl_plugin_add_fset(
				star_otyp_plugin_runtime_id,
				FZ_OTYP_CBAK_FSET_TYPE,
				FZ_OTYP_CBAK_FSET_VERSION,
				FZ_OTYP_CBAK_FSET_NAME,
				FZPL_TYPE_STRING(fz_otyp_cbak_fset),
				sizeof ( fz_otyp_cbak_fset ),
				star_otyp_fill_cbak_fset,
				FALSE);
	}


	return(err);
}
```

The function set registration passes a function to `fzpl_plugin_add_fset`, which is executed by **form•Z** at startup. In the example above, the registration of the object type passes the function `star_otyp_fill_cbak_fset`. This function must be defined by the plugin developer and must fill in the object type function set with the pointers of the callback functions which constitute the functionality of a custom object type. An example of this registration process is shown below. It assigns the callbacks of the star object type to the function set. It is quite possible to register more than one object type function set with a plugin. In this case the `fzpl_plugin_add_fset` call needs to be repeated for each function set, using the same runtime id, but a different callback function set fill function.

```
fzrt_error_td star_otyp_fill_cbak_fset (
             const fzpl_fset_def_ptr fset_def,
             fzpl_fset_td * const fset
             )
{
       fzrt_error_td              err = FZRT_NOERR;
       fz_otyp_cbak_fset          *otyp_fset;

       err = fzpl_glue->fzpl_fset_def_check (
             fset_def,
             FZ_OTYP_CBAK_FSET_VERSION,
             FZPL_TYPE_STRING(fz_otyp_cbak_fset),
             sizeof ( fz_otyp_cbak_fset ),
             FZPL_VERSION_OP_NEWER );

       if ( err == FZRT_NOERR )
       {
             otyp_fset = (fz_otyp_cbak_fset *)fset;

             otyp_fset->fz_otyp_cbak_uuid            = star_otyp_uuid;
             otyp_fset->fz_otyp_cbak_info            = star_otyp_info;
             otyp_fset->fz_otyp_cbak_init            = star_otyp_init;
             otyp_fset->fz_otyp_cbak_finit           = star_otyp_finit;
             otyp_fset->fz_otyp_cbak_name            = star_otyp_name;
             otyp_fset->fz_otyp_cbak_tform           = star_otyp_tform;
             otyp_fset->fz_otyp_cbak_geom            = star_otyp_geom;
             otyp_fset->fz_otyp_cbak_regen           = star_otyp_regen;
             otyp_fset->fz_otyp_cbak_iface_tmpl      = star_otyp_iface_tmpl;
             otyp_fset->fz_otyp_cbak_get_key_pnts    = star_otyp_get_key_pnts;
             otyp_fset->fz_otyp_cbak_io              = star_otyp_io;
             otyp_fset->fz_otyp_cbak_rvrs            = NULL;
             otyp_fset->fz_otyp_cbak_copy            = NULL;
             otyp_fset->fz_otyp_cbak_cvsl            = NULL;
             otyp_fset->fz_otyp_cbak_cvrt_ptch       = NULL;
             otyp_fset->fz_otyp_cbak_get_ncur        = NULL;
             otyp_fset->fz_otyp_cbak_get_nsrf        = NULL;
             otyp_fset->fz_otyp_cbak_cnstr_smod      = NULL;
             otyp_fset->fz_otyp_cbak_copy_cntl_objts = NULL;
             otyp_fset->fz_otyp_cbak_parm_count      = star_parm_count;
             otyp_fset->fz_otyp_cbak_parm_get_info2  = star_parm_get_info;
             otyp_fset->fz_otyp_cbak_parm_get_state_str = star_parm_get_state_str;
             otyp_fset->fz_otyp_cbak_parm_get        = star_parm_get;
             otyp_fset->fz_otyp_cbak_parm_set        = star_parm_set;


       }

       return err;
}
```

Of the all the callback functions of an object type function set, only some are required, while the others are optional. When an optional callback is not assigned to the function set, the respective functionality of the object type is disabled or performed by **form•Z** in a generic fashion. For example, if the `fz_otyp_cbak_iface_tmpl` function is not defined, the Edit button in the Query dialog is disabled when an object of this type is queried. The required functions are:

```
fz_otyp_cbak_name
fz_otyp_cbak_uuid
fz_otyp_cbak_info
fz_otyp_cbak_init
fz_otyp_cbak_regen
fz_otyp_cbak_io
```

`fz_otyp_cbak_parm_count` and `fz_otyp_cbak_parm_get_info2` are optional. However, if they are defined, `fz_otyp_cbak_io` is optional. That is, either `fz_otyp_cbak_io` or `fz_otyp_cbak_parm_count` and `fz_otyp_cbak_parm_get_info2` must be defined.

All others are optional. Note that there is no callback function to explicitly create an object of the given type. Usually, the object type plugin is not registered alone, but is paired with another plugin, such as a tool command. This is the case with the star example. The tool command plugin can be set up to define a new modeling tool, which manifests itself in form of an icon in the main tool palette. The tool command plugin can be written, so that selecting the tool and clicking in the modeling window creates a new object using the type defined by the object type plugin. In this section, only the object type plugin is described in more detail. The tool command plugin is described in more detail in section 2.6.3.

Object type function implementation

The following section gives a detailed description of each of the object type callback functions and what task each function is expected to perform.

The name function (required)

```
fzrt_error_td fz_otyp_cbak_name (
	long			windex,
	fz_objt_ptr		objt,
	fzrt_ptr		parm,
	char			*str,
	long			max_str
	);
```

The name function defines a name for the object type. This name will show up in the **form•Z** interface, whenever object types are listed. The name function must assign a string to the function's name argument. The length of the string assigned cannot exceed `max_len` characters. An example of a name function is shown below.

```
fzrt_error_td star_otyp_name(
	long			windex,
	fz_objt_ptr		objt,
	fzrt_ptr		parm,
	char			*str,
	long			max_len
	)
{

	strncpy(str,"Star",max_len);
```

```
        return(FZRT_NOERR);
}
```

The `objt` and `parm` parameters may be passed as `NULL`. In this case a name for all objects of this type should be returned. If `objt` and `parm` are passed in, a particular object of this type exists, and the type name may be further specified based on the parameters of the object. For example, the sweep object type in **form•Z** works this way. When its name function is called with `NULL`, it returns "Sweep". However, if it is called with a particular object as the parameter, the returned name contains which type of sweep it is, for example, "Axial Sweep", or "Two Source Sweep". Other object types do not make such a distinction and always return the same name, such as spheres, nurbz or symbols. It is recommended that the object type name is stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example above, this step is omitted for the purpose of simplicity.

The uuid function (required)

```
fzrt_error_td fz_otyp_cbak_uuid (
        fzrt_UUID_td                uuid
        );
```

Each object type is tagged with a unique identifier. This allows **form•Z** to distinguish objects of one type from all others. When a **form•Z** project file is written to disk, the uuid of the object type is saved with the object. When the project file is later opened again, **form•Z** will connect the loaded object type data with the installed plugin. If the plugin that created the object type is not installed, the object is automatically dropped to a plain object. The uuid function needs to assign this unique identifier string to the function's uuid argument. An example is shown below.

```
#define STAR_OTYP_UUID     \
"\x2d\xa8\x6d\xe1\xdb\xd3\x40\xc4\xa7\xb3\xd9\xe3\xd2\x73\x69\x75"

fzrt_error_td star_otyp_uuid (
        fzrt_UUID_td  uuid
        )
{
        fzrt_UUID_copy(STAR_OTYP_UUID, uuid);
        return(FZRT_NOERR);
}
```

The info function (required)

```
fzrt_error_td fz_otyp_cbak_info (
        long          *size,
        long          *flags
        );
```

form•Z manages the storage of each occurence of an object type. In order to do so, **form•Z** needs to know, what the data size (in # of bytes) of the object type parameters is. The info function is expected to return the number of bytes that the parameter storage requires. In most cases, a plugin developer will create a structure with fields which describe the object type parameters. The size returned to **form•Z** via this callback can be acquired with a `sizeof(structure_type)` call.

form•Z also needs to know some basic information about the object type, for example, whether the object type is always smooth, always facetted or both. This information is defined in the flags argument. This argument should be set with the bit encoded flags defined in the enum `fz_otyp_flags_enum`. Setting a bit in the flags argument of the function enables the functionality described by the bit. Setting a bit can be done with the FZ_SETBIT utility function. In case of the star object, it is defined to always generate facetted model type objects and also chooses to let **form•Z** handle the reversing of the object topology.

The info function for the star object type is shown below.

```
fzrt_error_td star_otyp_info (
       long          *size,
       long          *flags
       )
{
       *size = sizeof(star_otyp_td);

       *flags = 0;
       FZ_SETBIT(*flags,FZ_OTYP_ALWAYS_FACET);
       FZ_SETBIT(*flags,FZ_OTYP_HANDLE_RVRS);

       return(FZRT_NOERR);
}
```

A complete description of all object type flags follows:

`FZ_OTYP_NON_UNI_SCALE`
Certain parametric data cannot be scaled non uniformly. For example, local coordinate system with its own x, y and z axes would be distorted and even skewed with a non uniform scaling. In such a case, this bit should not be set. If a non uniform scale is applied to the object, the control parameters are automatically dropped by **form•Z**. Other parametric data can be scaled non uniformly. This is the case, for example, with nurbZ curves, which are defined by a set of control points. Scaling the control points also scales the evaluated shape of the curve. In this case, the bit should be set. The object can then be scaled non uniformly without loosing the parameters data.

`FZ_OTYP_NO_RENDER`
When this bit is set, the object will not be rendered in high end rendering modes, such as RenderZone. They will only be rendered in the interactive rendering modes. If the bit is not set, the object will always be rendered. This flag is expected to be used less frequently. It may be applied to object types, which are temporary in nature.

`FZ_OTYP_NO_SYS_FLIP`
When this bit is set, the object cannot be transformed so that a coordinate system changes from left hand to right hand without dropping the object to a plain object. Such a transformation occurs, for example, when mirroring about a plane or when scaling with one of the scale factors being negative and the other ones positive. If this bit is not set, such transformations are allowed and the object controls are not dropped.

`FZ_OTYP_ALWAYS_SMOOTH`
When this bit is set, the object is always a smooth object. In other words, its model type is always smooth. It is not possible to have both, `FZ_OTYP_ALWAYS_SMOOTH` and `FZ_OTYP_ALWAYS_FACET` set. However if none are set, the object may be smooth or facetted.

`FZ_OTYP_ALWAYS_FACET`

When this bit is set, the object is always a facetted object. In other words, it never has a smooth object representation. It is not possible to have both, `FZ_OTYP_ALWAYS_SMOOTH` and `FZ_OTYP_ALWAYS_FACET` set. However if none are set, the object may be smooth or facetted.

`FZ_OTYP_HANDLE_RVRS`
When this bit is set, the parametric representation of the object cannot be reversed in direction. In this case, **form•Z** will reverse the object facets after a reverse operation occurred. If this bit is not set, it is the responsibility of the object type to reverse its parametric data. This is usually done in the `fz_otyp_cbak_rvrs` callback function.

`FZ_OTYP_EXPL_PER_PART`
When this bit is set, the explode operation may yield multiple volumes for this object. When this bit is not set, the object is always represented by only one volume. In the Convert Options dialog, the Per Part check box will be added if this bit is set.

`FZ_OTYP_NESTED_CURVE_CNTRL`
When this bit is set, the object type is assumed to define an open or closed curve, which lends itself as the source for a number of other derivative objects, such as sweep, helix or revolved objects.

The init function (required)

```
fzrt_error_td fz_otyp_cbak_init (
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm
        );
```

form•Z calls this function to initialize the parameters of the object with default values. The storage for the parameters has already been allocated by **form•Z** and is passed in to this function as the `parm` parameter. The object to which the parameters belong and the project in which the object resides are passed in as well. The init function for the star object type is shown below.

```
fzrt_error_td star_otyp_init(
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm
        )
{
        star_otyp_parms_td  *star;
        short               err = FZRT_NOERR;

        if(parm != NULL)
        {
                star= (star_otyp_parms_td *)parm;

                star->base_type = _star_tool_opts->base_type;

                star->origin.x = 0.0;
                star->origin.y = 0.0;
                star->origin.z = 0.0;

                star->xaxis.x = 1.0;
                star->xaxis.y = 0.0;
                star->xaxis.z = 0.0;
                star->yaxis.x = 0.0;
                star->yaxis.y = 1.0;
                star->yaxis.z = 0.0;
```

```
        star->radius    = _star_tool_opts->radius;
        star->rad_ratio = _star_tool_opts->rad_ratio;
    }

    return(err);
}
```

Note that the `base_type`, `radius` and `rad_ratio` parameters are not set to fixed values, but are assigned from the tool option's current values. In the example provided, the star object type is combined with the star tool command plugin, which executes the creation of a star object.

The regeneration function (required)

```
fzrt_error_td fz_otyp_cbak_regen(
    long                windex,
    fz_objt_ptr         obj,
    fzrt_ptr            parm
    );
```

The regeneration function is called when **form•Z** needs to recreate the shape of the object based on the current settings of the object's parameters. This may be necessary, for example, after the display resolution attribute of the object was edited, or a parameter of the object was altered through the edit dialog, invoked from the Query dialog. This function constitutes the real essence of the object type, as it defines the steps necessary to create the final form of the object, executed by calling various **form•Z** API functions. There are a number of ways to create the object's shape. One would be to construct one face at a time, using the API `fz_objt_fact_create_face`. This process is illustrated in the regenerate function of the star object type shown below.

```
fzrt_error_td star_otyp_regen(
    long                windex,
    fz_objt_ptr         obj,
    fzrt_ptr            parm
    )
{
    fzrt_error_td               rv = FZRT_NOERR;
    fz_xyz_td                   rxyz,rot,pnt,vec;
    double                      radius;
    long                        i,n,ncord,nsegt,ncurv,nface,
                                ncord2,nsegt2,ncurv2,nface2;
    long                        sindx,shead,snext,pindx[3],lval;
    fz_map_plane_td             local_mplane;
    fz_objt_ptr                 temp_obj;
    fz_objt_spid_type_enum      spid_type;
    fz_objt_spid_cnstr_opts_ptr      spid_opts;
    fz_type_td                  data;
    fzrt_boolean                bval;

    star_otyp_parms_td  *star;

    if(parm != NULL)
    {
        star = (star_otyp_parms_td *)parm;

        star_otyp_get_mplane(star,&local_mplane);
        fz_objt_fact_reset(windex, obj);

        fz_math_3d_vec_rotation_xyz(&star->xaxis,&star->yaxis,&rot);

        radius = star->radius *
```

```
                (START_RATIO_MIN + star->rad_ratio *
                (START_RATIO_MAX - START_RATIO_MIN));
        rxyz.x = radius;
        rxyz.y = radius;
        rxyz.z = radius;
        spid_opts = NULL;

        /* SETUP OPTIONS FOR A TEMPORARY SPHEROID OBJECT */
        switch ( star->base_type )
        {
                case 0: spid_type = FZ_OBJT_SPID_TYPE_TETRA;   break;
                case 1: spid_type = FZ_OBJT_SPID_TYPE_HEXA;    break;
                case 2: spid_type = FZ_OBJT_SPID_TYPE_OCTA;    break;
                case 3: spid_type = FZ_OBJT_SPID_TYPE_DODECA;  break;
                case 4: spid_type = FZ_OBJT_SPID_TYPE_ICOSA;   break;
                case 5: spid_type = FZ_OBJT_SPID_TYPE_SOCCER;  break;
                case 6:
                case 7:
                        spid_type = FZ_OBJT_SPID_TYPE_GEO;
                        fz_objt_cnstr_spid_opts_init(windex,&spid_opts);
                        if ( star->base_type == 6 )       lval = 1;
                        else                              lval = 2;
                        fz_type_set_long(&lval, &data);
                        fz_objt_cnstr_spid_opts_set(windex,spid_opts,
                                FZ_OBJT_SPID_PARM_GEO_LEVEL,&data);
                        bval = TRUE;
                        fz_type_set_boolean(&bval, &data);

                        fz_objt_cnstr_spid_opts_set(windex,spid_opts,
                                FZ_OBJT_SPID_PARM_GEO_BY_LEVEL,&data);
                break;

        }

        /* CONSTRUCT A TEMPORARY SPHEROID OBJECT */
        if((rv = fz_objt_cnstr_spid(windex,
                                &rxyz,spid_type,
                                &star->origin,
                                &rot,spid_opts,&temp_obj)) == FZRT_NOERR)
        {

                /* COUNT HOW MUCH STORAGE IS NEEDED */
                fz_objt_get_face_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,&nface);
                fz_objt_get_curv_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,&ncurv);
                fz_objt_get_segt_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,&nsegt);
                fz_objt_get_point_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,&ncord);

                ncord2 = ncord + nface;
                ncurv2 = 0;
                nface2 = 0;
                nsegt2 = 0;
                for(i = 0; i < ncurv; i++)
                {

                        fz_objt_curv_get_segt_count(windex,temp_obj,
                                i,FZ_OBJT_MODEL_TYPE_FACT,&n);
                        ncurv2 +=  n;
                        nface2 +=  n;
                        nsegt2 +=  n * 3;
                }
```

```
			/* ALLOCATE STORAGE FOR FACES, CURVES, SEGMENTS AND POINTS */
			if((rv = fz_objt_fact_allocate(windex,obj,
					nface2,ncurv2,nsegt2,ncord2)) == FZRT_NOERR )
			{
				/* COPY SPHEROID POINTS */
				for(i = 0; i < ncord; i++)
				{
					fz_objt_point_get_xyz(windex,temp_obj,
						i,FZ_OBJT_MODEL_TYPE_FACT,&pnt);
					fz_objt_fact_add_pnts(windex,obj,&pnt,1);
				}

				/* CREATE STAR TIP POINTS */
				radius = star->radius - radius;
				for(i = 0; i < nface; i++)
				{
					fz_objt_alys_get_face_cog(windex,temp_obj,i,
							FZ_OBJT_MODEL_TYPE_FACT,&pnt);
					fz_math_3d_create_unit_vec(&star->origin,&pnt,&vec);
					pnt.x += vec.x * radius;
					pnt.y += vec.y * radius;
					pnt.z += vec.z * radius;
					fz_objt_fact_add_pnts(windex,obj,&pnt,1);
				}

				/* CREATE FACES */
				for(i = 0; i < ncurv; i++)
				{

					fz_objt_curv_get_segt_count(windex,temp_obj,
						i,FZ_OBJT_MODEL_TYPE_FACT,&n);
					fz_objt_curv_get_sindx(windex,temp_obj,
						i,FZ_OBJT_MODEL_TYPE_FACT,&shead);
					sindx = shead;
					do
					{

						fz_objt_segt_get_next(windex,temp_obj,
							sindx,FZ_OBJT_MODEL_TYPE_FACT,&snext);
						fz_objt_segt_get_start_pindx(windex,temp_obj,
							sindx,FZ_OBJT_MODEL_TYPE_FACT,&pindx[0]
						);
						fz_objt_segt_get_end_pindx(windex,temp_obj,
							sindx,FZ_OBJT_MODEL_TYPE_FACT,&pindx[1]
						);
						pindx[2] = ncord + i;
						fz_objt_fact_create_face(windex,obj,
							pindx,3,NULL);

					} while ((sindx = snext) != shead );
				}

				/* LINK FACES */
				fz_objt_fact_link_faces(windex,obj);
			}

			if (spid_opts) fz_objt_cnstr_spid_opts_finit(windex,&spid_opts);

			/* DELETE THE TEMPORARY SPHEROID OBJECT */
			fz_objt_edit_delete_objt(windex, temp_obj);
		}
	}

	return(rv);
}
```

Another method to create the object's shape would be to use a sequence of higher level API construction functions. These will create temporary objects, which can be combined using editing API function to yield the final object. The temporary objects used along the way need to be deleted and the content of the final object copied into the object passed into the regeneration function. For example, the star object could be constructed by creating a number of pyramids (the star's rays), transforming them to attach to the faces of a spheroid object and then using the boolean union tool to join the all together into the final shape. The intermediate objects all need to be deleted. In this case, the direct creation process clearly is the better approach.

The io stream function (required)

```
fzrt_error_td fz_otyp_cbak_io_func(
	long			windex,
	fz_objt_ptr		obj,
	fzrt_ptr		parm,
	fz_iost_ptr		iost,
	fz_iost_dir_td_enum	dir,
	fzpl_vers_td * const	version,
	unsigned long		size
	);
```

form•Z calls this function to write the parameters of an object to and read it from file. It is expected from the plugin to keep track of version changes of the object's parameters. For example, in its first release, the object parameters may consist of one long integer, three xyz and two double parameters (as it is the case with the sample star object type). When written, the version reported back to **form•Z** was 0. In a subsequent release, the plugin developer added a second long integer value. When writing these new object parameters, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the object parameters, **form•Z** will pass in the version number of the object parameters when they were written, in this case 0. This indicates to the plugin that only one long, three xyz and two doubles need to be read and the second long should be set to a default value. Likewise, it is possible that an older version of the plugin will be asked to read a newer version of the object parameters. This may be the case when backsaving a **form•Z** project file to an older version and then reading that file with and older version of **form•Z** that contains the older version of the object type plugin. In this case, the plugin may choose to read the data written by version 0. For safety it may also choose to skip any object type data that is written with a newer version, than the one it is currently set to. If the plugin decides to read a newer version of the data, it is important that additional values are written at the end, not in the middle of the original values. The io steam function of the star object type is shown below.

```
fzrt_error_td star_otyp_io(
	long				windex,
	fz_objt_ptr			obj,
	fzrt_ptr			parm,
	fz_iost_ptr			iost,
	fz_iost_dir_td_enum		dir,
	fzpl_vers_td * const		version,
	unsigned long			size
	)
{
	fzrt_error_td		err = FZRT_NOERR;
	star_otyp_parms_td	*star;

	star = (star_otyp_parms_td *)parm;

	if ( dir == FZ_IOST_WRITE )
	{	*version = 0;
```

```
    }

    if((err = fz_iost_long(iost,&star->base_type,1))      == FZRT_NOERR &&
       (err = fz_iost_xyz(iost,&star->origin,1))          == FZRT_NOERR &&
       (err = fz_iost_xyz(iost,&star->xaxis,1))           == FZRT_NOERR &&
       (err = fz_iost_xyz(iost,&star->yaxis,1))           == FZRT_NOERR &&
       (err = fz_iost_double(iost,&star->radius,1))       == FZRT_NOERR )
    {
        err = fz_iost_double(iost,&star->rad_ratio,1);
    }

    return(err);
}
```

The finit function (optional)

```
fzrt_error_td fz_otyp_cbak_finit (
    long                windex,
    fz_objt_ptr         objt,
    fzrt_ptr            parm
    );
```

form•Z calls the finit function whenever an object of the given type is deleted. The function is expected to free any dynamic memory or take whatever action is necessary, when an object of this type ceases to exist. Note that it is not necessary to delete the basic storage for the object's parameters, which is passed in this function as the `parm` argument. In case of the star object, the finit function is not necessary as no dynamic memory is used. A finit function of an arbitrary sample object type, which has an array is shown below.

```
fzrt_error_td my_otyp_finit (
    long                windex,
    fz_objt_ptr         objt,
    fzrt_ptr            parm
    )
{
    my_otyp_td          *my_otyp;
    fzrt_zone_ptr       zone_ptr;

    my_otyp = (my_otyp_td*) parm;

    if (my_otyp->array)
    {
        fz_objt_get_zone_ptr(windex,objt,&zone_ptr);
        fz_mem_zone_free(zone_ptr,(fzrt_ptr*)&my_otyp->array);
    }

    return(FZRT_NOERR);
}
```

In the example above, the array was allocated using a memory zone. It is important that the same zone is used for both, allocation and deallocation. In this case, the memory zone which is assigned to the object is used. It can be retrieved with the API call `fz_objt_get_zone_ptr`. A plugin may also define its own memory zone. This is discussed in more detail in section 1.4.4.

The transform function (optional)

```
fzrt_error_td fz_otyp_cbak_tform (
    long                windex,
    fz_objt_ptr         objt,
    fzrt_ptr            parm,
    fz_mat4x4_td        *tform
```

```
);
```

form•Z calls the transform function whenever an object is transformed (moved, rotated, scaled and/or mirrored). When an object contains positional geometric properties, such as an origin, 3d points or even a complete nested control object, they need to be transformed as well. Points can be transformed with the math API function `fz_math_4x4_multiply_mat_xyz`. If an object contains a linear dimension, such as a radius, only the scale factor of the matrix need to be applied. This scale factor can be extracted with the math API `fz_math_4x4_mat_to_trl_scl_rot`. If the parameters of the object type contain an entire nested object, it should be transformed with the API function `fz_objt_edit_transform`. The transform function for the star object type is listed below.

```
fzrt_error_td star_otyp_tform(
	long			windex,
	fz_objt_ptr		objt,
	fzrt_ptr		parm,
	fz_mat4x4_td		*tform
	)
{
	star_otyp_parms_td		*parms_ptr;
	fz_xyz_td			xaxis,yaxis,scl;
	fzrt_error_td			rv = FZRT_NOERR;

	if(parm != NULL)
	{
		parms_ptr = (star_otyp_parms_td *)parm;

		xaxis.x = parms_ptr->origin.x + parms_ptr->xaxis.x;
		xaxis.y = parms_ptr->origin.y + parms_ptr->xaxis.y;
		xaxis.z = parms_ptr->origin.z + parms_ptr->xaxis.z;

		yaxis.x = parms_ptr->origin.x + parms_ptr->yaxis.x;
		yaxis.y = parms_ptr->origin.y + parms_ptr->yaxis.y;
		yaxis.z = parms_ptr->origin.z + parms_ptr->yaxis.z;

		fz_math_4x4_multiply_mat_xyz(tform, &parms_ptr->origin);
		fz_math_4x4_multiply_mat_xyz(tform, &xaxis);
		fz_math_4x4_multiply_mat_xyz(tform, &yaxis);

		fz_math_3d_create_unit_vec(&parms_ptr->origin,
			&xaxis,
			&parms_ptr->xaxis);
		fz_math_3d_create_unit_vec(&parms_ptr->origin,
			&yaxis,
			&parms_ptr->yaxis);

		/* GET SCALE FROM MATRIX */
		fz_math_4x4_mat_to_trl_scl_rot(&tform,NULL,&scl,NULL);
		parms_ptr->radius *= ( scl.x + scl.y + scl.z) / 3.0;;
	}

	return(rv);
}
```

The copy function (optional)

```
fzrt_error_td fz_otyp_cbak_copy (
		long			src_windex,
		fz_objt_ptr		src_objt,
		fzrt_ptr		src_parm,
		long			dst_windex,
```

```
        fz_objt_ptr         dst_objt,
        fzrt_ptr            dst_parm
    );
```

form·Z calls the copy function whenever an object is copied. It should be implemented when the parametric data contains dynamically allocated arrays or nested control objects. To copy an array, the copy function must first allocate space in the destination parameter block and then transfer the array content from the source to the destination. Since the star object type does not contain any dynamic arrays, the copy function of an arbitrary object type which contains an array is shown below.

```
fzrt_error_td my_otyp_copy (
        long                src_windex,
        fz_objt_ptr         src_objt,
        fzrt_ptr            src_parm,
        long                dst_windex,
        fz_objt_ptr         dst_objt,
        fzrt_ptr            dst_parm
    )
{
    my_otyp_td          *src_my_otyp,*dst_my_otyp;
    fzrt_zone_ptr       zone_ptr;
    fzrt_error_td       err = FZRT_NOERR;

    src_my_otyp = (my_otyp_td *) src_parm;
    dst_my_otyp = (my_otyp_td *) dst_parm;

    fz_objt_get_zone_ptr(dst_windex,dst_objt,&zone_ptr);
    if((err = fz_mem_zone_alloc(zone_ptr,
                            sizeof(long) * src_my_otyp->n_array,
                            FALSE,
                            (fzrt_ptr*)&dst_my_otyp->array)
      ) == FZRT_NOERR )
    {

        fzrt_block_move(src_my_otyp->array,
                        dst_my_otyp->array,
                        sizeof(long) * src_my_otyp->n_array);
        dst_my_otyp->n_array = src_my_otyp->n_array;

        /* COPY REMAINING FIELDS */
        dst_my_otyp->value1 = src_my_otyp->value1;
        dst_my_otyp->value2 = src_my_otyp->value2;
        /* … ETC */
    }

    return(err);

}
```

To copy a nested control object, the copy function can use the API function `fz_objt_edit_copy_objt_geom`. For more information about nested control objects, see the details at the end of the section.

The reverse function (optional)

```
fzrt_error_td       fz_otyp_cbak_rvrs (
    long                windex,
    fz_objt_ptr         objt,
```

```
        fzrt_ptr            parm
        );
```

form•Z calls the reverse function of a controlled object, when the object's topology needs to be reversed in its direction. This is the case, for example, if the Reverse tool is applied, or if an object is mirrored. The reverse function gives the object type the opportunity to reverse its parametric data. If the parametric data is defined in a way that it cannot be reversed, the `FZ_OTYP_HANDLE_RVRS` the flags parameter of the `fz_otyp_cbak_info` function call should be set. **form•Z** will then handle the reversal of the objects facets when necessary. An example of an object type, whose parametric data cannot be reversed is the sphere object. It's shape is implicitly based on a right handed coordinate system. The sphere object type does not have a reverse function. The NurbZ object, on the other hand can be reversed. This is done by swapping all the control points of the nurbs surface on which the NurbZ obejct is build. This swapping is performed by the reverse function defined by the NurbZ object type. The reverse function of a sample object type, which is based on an array of xyz points is shown below.

```
fzrt_error_td       my_otyp_rvrs (
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm
        )
{
        fz_xyz_td     temp;
        long          nhalf,i,j;
        my_otyp_td    *my_otyp;


        my_otyp = (my_otyp_td*) parm;

        nhalf = my_otyp->npnts * 0.5;

        for( i = 0, j = my_otyp->npnts - 1; i < nhalf; i++, j--)
        {
                temp = my_otyp->pnts[i];
                my_otyp->pnts[i] = my_otyp->pnts[j];
                my_otyp->pnts[j] = temp;
        }

        return(FZRT_NOERR);

}
```

The geometry function (optional)

```
fzrt_error_td       fz_otyp_cbak_geom (
        long                windex,
        fz_objt_ptr         obj,
        fzrt_ptr            parm,
        fz_map_plane_td     *plane,
        fz_xyz_td           *center,
        fz_xyz_mm_td        *bbox
        );
```

form•Z calls the geometry function to retrieve basic geometric information about the object. It should be implemented if the object has its own, local coordinate system. For example, a sphere has its own x, y and z axis, which describe the location and orientation of the sphere in 3d space. The `plane` parameter returns the origin and rotation of the object's coordinate system in world space. This information is used, for example, to draw the object axes in wireframe. The `center` parameter returns the object's origin in the coordinate space of the object. Usually the center would be set to {0.0, 0.0, 0.0}, but may have different values, depending on the nature of the object. The `bbox` parameter returns the extent of the object along its x, y and z axis. If this function is not implemented by the plugin, the information is calculated from the facetted data of the object. For example, the center is computed as the average of all coordinate points of the object. The geometry function for the star object type is shown below.

```
fzrt_error_td star_otyp_geom(
	long			windex,
	fz_objt_ptr		obj,
	fzrt_ptr		parm,
	fz_map_plane_td		*plane,
	fz_xyz_td		*center,
	fz_xyz_mm_td		*bbox
	)
{
	star_otyp_parms_td	*star;
	fz_xyz_td		xaxis,yaxis;
	fzrt_error_td		err = FZRT_NOERR;

	if(parm != NULL)
	{
		star = (star_otyp_parms_td*) parm;

		if ( plane )
		{
			xaxis.x = star->origin.x + star->xaxis.x;
			xaxis.y = star->origin.y + star->xaxis.y;
			xaxis.z = star->origin.z + star->xaxis.z;

			yaxis.x = star->origin.x + star->yaxis.x;
			yaxis.y = star->origin.y + star->yaxis.y;
			yaxis.z = star->origin.z + star->yaxis.z;

			fz_math_3d_map_plane_from_pts(&xaxis,
				&star->origin,
				&yaxis, plane);
		}

		if(center)
		{
			center->x = 0.0;
			center->y = 0.0;
			center->z = 0.0;
		}

		if(bbox)
		{
			bbox->xmin = -star->radius;
			bbox->ymin = -star->radius;
			bbox->zmin = -star->radius;
			bbox->xmax =  star->radius;
			bbox->ymax =  star->radius;
			bbox->zmax =  star->radius;
		}
	}

	return(err);
}
```

The cvsl function (optional)

```
fzrt_error_td fz_otyp_cbak_cvsl (
	long			windex,
	fz_objt_ptr		obj,
	fzrt_ptr		parm,
	fz_xyz_td		*cog,
	double			*volume,
	double			*surf_area,
	double			*length,
	long			*result
	);
```

The cvsl function is called by **form•Z** to retrieve the center of gravity, volume, surface area and length (abbreviated cvsl) of an object. This function should be implemented, when the object type can provide more accurate values, than those computed from the facetted or smooth topology and geometry of the object. Since not all of these properties can be calculated for an object, the result parameter returned to **form•Z** tells which properties were computed by the function, by setting certain bits to on.

bit 0: center of gravity was calculated
bit 1: volume was calculated
bit 2: surface area was calculated
bit 3: perimeter length was calculated

For example, the perimeter length can only be calculated for curve like objects but not for solids. Therefore, for solids, bit #3 should not be set. The cvsl function for an object type which creates a regular sphere is shown below.

```
fzrt_error_td my_sphr_cvsl (
	long			windex,
	fz_objt_ptr		obj,
	fzrt_ptr		parm,
	fz_xyz_td		*cog,
	double			*volume,
	double			*surf_area,
	double			*length,
	long			*result
	)
{
	my_sphr_td	*sphr;
	short		rv = FZRT_NOERR;

	sphr = (my_sphr_td*)parm;

	if (cog)
	{	*cog = sphr->origin;
		FZ_SETBIT(*result,0);
	}

	if (volume)
	{	* volume = (	sphr->radius *
				sphr->radius *
				sphr->radius * 4.0 * FZ_PI) / 3.0;
		FZ_SETBIT(*result,1);
	}

	if (surf_area)
	{	*surf_area  = 4.0 * FZ_PI * sphr->radius * sphr->radius;
		FZ_SETBIT(*result,2);
	}

	return(rv);
}
```

The convert to patch function (optional)

```
fzrt_error_td fz_otyp_cbak_cvrt_ptch (
       long                windex,
       fz_objt_ptr         obj,
       fzrt_ptr            parm,
       long                action,
       fzrt_boolean        *can_cvrt
       );
```

This function is called by **form•Z** to convert the object to a parametric bezier or coons patch object. It should only be implemented if this can be done without changing the shape of the object. The function can get called in two ways. When the action argument is 0, the function only needs to check whether the particular object can be converted. Depending on the parametric data, it is possible that an object can or cannot be converted to patches. When the action parameter is 1, the object needs to be converted. Note that this function does not create a new object, but needs to change the object passed in. An example of an arbitrary object type's convert to patch function is shown below.

```
fzrt_error_td my_otyp_cvrt_ptch (
       long                       windex,
       fz_objt_ptr                obj,
       fzrt_ptr                   parm,
       long                       action,
       fzrt_boolean               *can_cvrt
       )
{
       fz_objt_ptr         temp_obj;
       fzrt_error_td       rv = FZRT_NOERR;

       if ( action == 0 )
       {
              *can_cvrt = TRUE;
       }
       else
       {
              *can_cvrt = TRUE;

              /* CODE TO CREATE A TEMPORARY PATCH OBJECT */
              .....

              /* COPY THE CONTENT OF THE TEMPORARY PATCH */
              /* OBJECT TO THE OBJECT PASSED IN */
              fz_objt_edit_copy_objt_data(windex,temp_obj,obj,FALSE);

              /* DELETE THE TEMPORARY PATCH OBJECT */
              fz_objt_edit_delete_objt(windex, temp_obj);
       }

       return(rv);

}
```

The nurbs curve function (optional)

```
fzrt_error_td fz_otyp_cbak_get_ncur(
       long                windex,
       fz_objt_ptr         obj,
```

```
        fzrt_ptr            parm,
        fzrt_boolean        clamp_closed,
        long                action,
        fzrt_boolean        *can_cvrt,
        fz_nurbs_cur_ptr    *ncur
        );
```

This function is called by **form•Z** to create a parametric nurbs curve entity from the object. It should only be implemented if the shape of the object can be represented by a single nurbs curve, without any sharp corners (discontinuities). Note that unlike the convert to patch function, the nurbs curve function does not convert the object passed in, but creates a new nurbs curve, which is passed back to **form•Z** through the last function argument. The function can get called in two ways. When the action argument is 0, the function only needs to check whether the nurbs curve can be created. Depending on the parametric data, it may or may not be possible to create a nurbs curve. When the action parameter is 1, the nurbs curve needs to be created. There are a number of **form•Z** API functions, which can be used to create nurbs curves. They can be found in the `fz_ncrv_fset` function set. The `clamp_closed` argument has a special meaning. If set to TRUE, a closed nurbs curve needs to be created with knot multiplicity at the start and end (clamped). Otherwise a closed curve needs to be created without knot multiplicity but with overlapping control points. An example of an arbitrary object type's nurbs curve function is shown below.

```
fzrt_error_td my_otyp_get_ncur(
        long                windex,
        fz_objt_ptr         obj,
        fzrt_ptr            parm,
        fzrt_boolean        clamp_closed,
        long                action,
        fzrt_boolean        *can_cvrt,
        fz_nurbs_cur_ptr    *ncur
        )
{
        fz_xyz_td           *cpts;
        long                npts;
        long                degree;
        double              *weights;
        double              *knots;
        fzrt_boolean        closed;
        fzrt_error_td       rv = FZRT_NOERR;

        if ( action == 0 )
        {
                *can_cvrt = TRUE;
        }
        else
        {
                *can_cvrt = TRUE;

                /* CODE TO SET THE NURBS CURVE  */
                /* PARAMETERS FROM THE OBJECT   */
                ...

                /* CREATE THE NURBS CURVE */
                fz_ncrv_create_nurbs_curve(cpts,npts,degree,
                        weights,knots,closed,ncur);

        }

        return(rv);

}
```

The nurbs surface function (optional)

```
fzrt_error_td fz_otyp_cbak_get_nsrf(
        long                windex,
        fz_objt_ptr         obj,
        fzrt_ptr            parm,
        long                action,
        fzrt_boolean        *can_cvrt,
        fz_nurbs_srf_ptr    *nsrf
        );
```

This function is called by **form•Z** to create a parametric nurbs surface entity from the object. It should only be implemented if the shape of the object can be represented by a single nurbs surface, without any sharp bends (discontinuities). Note that unlike the convert to patch function, the nurbs surface function does not convert the object passed in, but creates a new nurbs surface, which is passed back to **form•Z** through the last function argument. The function can get called in two ways. When the action argument is 0, the function only needs to check whether the nurbs surface can be created. Depending on the parametric data, it may or may not be possible to create a nurbs surface. When the action parameter is 1, the nurbs surface needs to be created. There are a number of **form•Z** API functions, which can be used to create nurbs surfaces. They can be found in the `fz_nsrf_fset` function set. An example of an arbitrary object type's nurbs surface function is shown below.

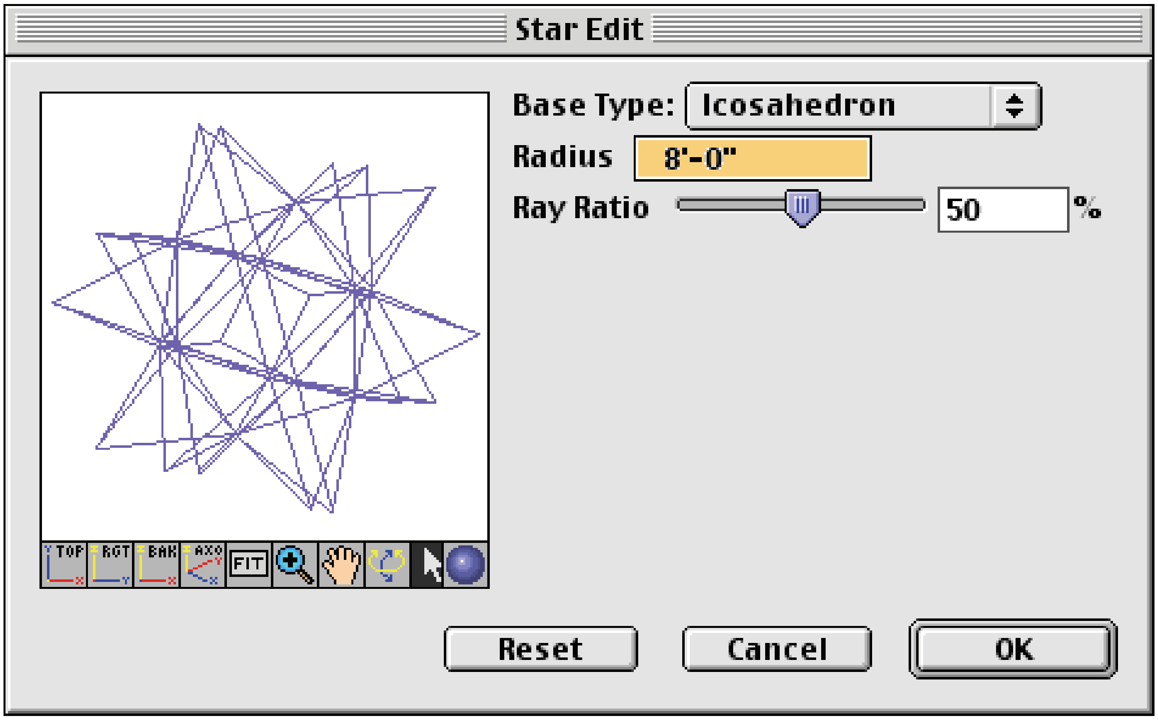
```
fzrt_error_td my_otyp_get_nsrf(
        long                windex,
        fz_objt_ptr         obj,
        fzrt_ptr            parm,
        long                action,
        fzrt_boolean        *can_cvrt,
        fz_nurbs_srf_ptr    *nsrf
        )
{
        fz_xyz_td           *cpts;
        double              *weights;
        long                u_npts;
        long                u_degree;
        double              *u_knots;
        fzrt_boolean        closed_u;
        long                v_npts;
        long                v_degree;
        double              *v_knots;
        fzrt_boolean        closed_v;
        fzrt_error_td       rv = FZRT_NOERR;
        fz_nurbs_srf_ptr    srf;

        if ( action == 0 )
        {
                *can_cvrt = TRUE;
        }
        else
        {
                *can_cvrt = TRUE;

                /* CODE TO SET THE NURBS SURFACE  */
                /* PARAMETERS FROM THE OBJECT     */
                ...

                /* CREATE THE NURBS SURFACE */
                fz_nsrf_create_nurbs_srf(cpts,weights,
```

```
                    u_npts,u_degree,u_knots,closed_u,
                    v_npts,v_degree,v_knots,closed_v,
                    &srf);

        }

        return(rv);

}
```

The dialog template function (optional)

```
fzrt_error_td fz_otyp_cbak_iface_tmpl(
        long                        windex,
        fz_fuim_tmpl_ptr            fuim_tmpl,
        fzrt_ptr                    obj_ptr
        );
```

form•Z calls this function, when the Edit button in the Query Object dialog is pressed. It is expected to initialize the dialog template and create all dialog items necessary to display the parameters of the object. This dialog may just present the parameters in simple text edit fields or go as far as offering a graphic preview window that shows the effects of the edited parameters. When this function is not implemented, **form•Z** will attempt to build the dialog automatically, based on the output of the parameter info function (see below). If the parameter info reports any parameters that can be displayed, **form•Z** will build the dialog with those parameters shown. If the parameter info function does not report any parameters and the dialog template function is also not implemented, the Edit button in the Query Object dialog is dimmed. The dialog template function for the star object is shown below together with the dialog invoked for a star object that was added to a project. For completion, the callback functions invoked from the dialog function are shown as well.



If a parameters of the object is reported by the parameter info function as an animatable parameter, different fuim item functions need to be called, so that the object parameter can be properly linked to the animation data. For example, the radius of the star sample object is an animatable parameter. Instead of using the `fz_fuim_new_text_static_edit` api function to build a text field with a title, the api `fz_fuim_new_text_static_edit_anim` needs to be used. The extra parameters passed to this function are :
• A flags parameter that is usually 0, unless a radio button item is created.
• The standard project window index parameter (windex).
• An identifier, that tells **form•Z**, that the interface item is constructed for an object type (instead of a light or view). It is an enum of type `fz_fuim_anim_item_type_enum` and needs to have the value `FZ_FUIM_ANIM_OBJECT_TYPE`.
The tag of the object being edited.
• A unique identifier for the object parameter. This uuid must be the same as the one reported by the parameter info function for that specific object parameter.

Note, that an object parameter is animatable if it is reported by the parameter info function and the parameter info function does not set the `FZ_OTYP_PARM_NO_ANIM_BIT` flag, but sets the `FZ_OTYP_PARM_ANIM_LEVEL1_BIT` or `FZ_OTYP_PARM_ANIM_LEVEL2_BIT` bit. If the parameter info function is not implemented or does not report a parameter, it is considered not animatable.

```
typedef struct star_otyp_pview_data_td
{
	int			src_windex;
	int			dst_windex;
	fz_objt_ptr		src_obj;
	fz_objt_ptr		dst_obj;
	star_otyp_parms_td	star_parms;
} star_otyp_pview_data_td;
```

```
static fzrt_error_td       star_otyp_iface_tmpl(
    long                   windex,
    fz_fuim_tmpl_ptr       fuim_tmpl,
    fzrt_ptr               objt_ptr
    )
{
    fzrt_error_td                  err = FZRT_NOERR;
    short                          g1,g2,g3;
    char                           str[256];
    fz_objt_ptr                    obj;
    star_otyp_parms_td             *star;
    fz_fuim_pview_opts_ptr         pview_opts;
    star_otyp_pview_data_td        pview_data,*pview_data_ptr;
    fzrt_menu_ptr                  menu;
    fz_tag_td                      obj_tag;

    obj = (fz_objt_ptr)objt_ptr;
    // GET THE OBJECT TAG
    fz_objt_ptr_to_tag(windex, obj,&obj_tag);

    // GET THE OBJECT PARAMETER DATA
    fz_objt_parm_get_data(windex,obj,(fzrt_ptr*)&star);

    fzrt_fzr_get_string(fz_rsrc_ref_func, STAR_STR_ID, STAR_STR_EDIT_TITLE, str);
    if((err = fz_fuim_tmpl_init(fuim_tmpl, str, 0, STAR_OTYP_ID, 0)) == FZRT_NOERR)
    {
        pview_data.src_obj = obj;
        pview_data.dst_obj = NULL;
        pview_data.src_windex = windex;
        pview_data.dst_windex = -1;
        pview_data.star_parms = *star;

        fz_fuim_tmpl_set_new_value_func(fuim_tmpl, star_otyp_fuim_newval, NULL);
        fz_fuim_tmpl_set_ok_func(fuim_tmpl,star_otyp_fuim_ok, NULL);
        fz_fuim_tmpl_set_reset_func(fuim_tmpl, star_otyp_fuim_reset, NULL);

        g1 = fz_fuim_new_group(fuim_tmpl, -1, FZ_FUIM_NONE,
               FZ_FUIM_FLAG_HORZ|FZ_FUIM_FLAG_GFLT, NULL);
            g2 = fz_fuim_new_group(fuim_tmpl, g1, FZ_FUIM_NONE,
                   FZ_FUIM_FLAG_NONE, NULL);

                // CREATE THE PREVIEW
                fz_fuim_pview_opts_init(&pview_opts,windex);
                fz_fuim_pview_opts_set_load_func(pview_opts, star_otyp_fuim_load_func);
                fz_fuim_pview_create(fuim_tmpl, g2, FZ_FUIM_NONE, pview_opts);
                fz_fuim_stack_put(fuim_tmpl, STAR_OTYP_STACK_PVIEW_DATA,
                    sizeof(pview_data), &pview_data);
                fz_fuim_stack_put(fuim_tmpl, STAR_OTYP_STACK_PVIEW_OPTS,
                    sizeof(pview_opts), &pview_opts);
                fz_fuim_stack_get_ptr(fuim_tmpl, STAR_OTYP_STACK_PVIEW_DATA,
                    (void**)&pview_data_ptr);
                star = &pview_data_ptr->star_parms;

            g2 = fz_fuim_new_group(fuim_tmpl, g1, FZ_FUIM_NONE,
                FZ_FUIM_FLAG_NONE, NULL);

            // CREATE THE TYPE MENU, AS AN ANIMATION ITEM
            fzrt_fzr_get_menu(fz_rsrc_ref_func, STAR_BASE_MENU_ID,&menu);
            fzrt_fzr_get_string(fz_rsrc_ref_func, STAR_STR_ID,
                STAR_STR_BASE_TYPE, str);
            g3 = fz_fuim_new_menu_anim(fuim_tmpl, g2, FZ_FUIM_NONE,
                FZ_FUIM_FLAG_HORZ, str, menu, FALSE, NULL, NULL,
                0,windex,FZ_FUIM_ANIM_OBJECT_TYPE,&obj_tag,STAR_PARM_BASE_TYPE_UUID);
            fz_fuim_item_range_long(fuim_tmpl,g3, &star->base_type, 0, 7,
```

```
                FZ_FUIM_FORMAT_INT_DEFAULT,
                FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MAX |
                FZ_FUIM_RANGE_MIN_INCL | FZ_FUIM_RANGE_MAX_INCL);


        // CREATE THE RADIUS TEXT EDIT FIELD, AS AN ANIMATION ITEM
        fzrt_fzr_get_string(fz_rsrc_ref_func, STAR_STR_ID,
            STAR_STR_RADIUS, str);

    fz_fuim_new_text_static_edit_anim(fuim_tmpl,g2,FZ_FUIM_NONE,str,FZ_FUIM_NONE,
        FZ_FUIM_FLAG_NONE, NULL, NULL,
        0,windex,FZ_FUIM_ANIM_OBJECT_TYPE,&obj_tag,
        STAR_PARM_RADIUS_UUID,&g3);
        fz_fuim_item_range_double(fuim_tmpl, g3, &star->radius, 0.0, 0.0,
            FZ_FUIM_FORMAT_FLOAT_DISTANCE, FZ_FUIM_RANGE_MIN);

        // CREATE THE RATIO SLIDER, AS AN ANIMATION ITEM
        fzrt_fzr_get_string(fz_rsrc_ref_func,
            STAR_STR_ID, STAR_STR_RAY_RATIO, str);
        fz_fuim_new_slider_edit_pcent_double_anim(
                fuim_tmpl,
                g2,
                str,
                FZ_FUIM_NONE,
                FZ_FUIM_NONE,
                0.0,
                1.0,
                0.0,
                100.0,
                0.0,
                100.0,
                FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
                FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL,
                NULL,
                NULL,
                &star->rad_ratio,
                0,windex,FZ_FUIM_ANIM_OBJECT_TYPE,&obj_tag,STAR_PARM_RAY_RATIO_UUID,
                NULL,
                NULL);

    }

    return (err);
}
```

The dialog shown includes the preview capability offered by **form•Z**. A number or callback functions used in the object type dialog template function are necessary for this functionality. They are discussed again briefly below. A complete description of the **form•Z** user interface API function can be found in section 1.4.6.

The preview load function

```
fzrt_error_td star_otyp_fuim_load_func(
        fz_fuim_tmpl_ptr fuim_tmpl,
        long src_windex,
        long dst_windex)
{
    long                    err=FZRT_NOERR;
    star_otyp_pview_data_td *pview_data;

    fz_fuim_stack_get_ptr(fuim_tmpl,STAR_OTYP_STACK_PVIEW_DATA,
        (void**)&pview_data);
```

```
    if(pview_data->src_obj != NULL)
    {
        pview_data->dst_windex = dst_windex;

        if((err = fz_objt_edit_copy_objt_to_windex(src_windex,
            pview_data->src_obj,dst_windex,
            TRUE, &pview_data->dst_obj)) == FZRT_NOERR )
        {
            err = fz_objt_add_objt_to_project(dst_windex,pview_data->dst_obj);
        }
    }

    return(err);
}
```

This function is invoked when the preview window is created. The preview window is a separate project. The load function is expected to copy the object to be preview to the preview project. This is achieved in this case via `fz_objt_edit_copy_objt_to_windex` and `fz_objt_add_objt_to_project`.

The new value function

```
fzrt_boolean star_otyp_fuim_newval(
        fz_fuim_tmpl_ptr    fuim_mngr,
        fzrt_ptr            data_ptr
        )
{
        star_otyp_pview_data_td    *pview_data;
        star_otyp_parms_td         *parm_data;

        fz_fuim_stack_get_ptr(fuim_mngr,STAR_OTYP_STACK_PVIEW_DATA,
              (void**)&pview_data);

        fz_objt_parm_get_data(pview_data->dst_windex,
              pview_data->dst_obj,(fzrt_ptr*)&parm_data);
        *parm_data = pview_data->star_parms;

        fz_objt_edit_parm_regen(pview_data->dst_windex,pview_data->dst_obj);

        return (TRUE);
}
```

This function is invoked, anytime a new value was entered in any of the dialog items. Once this happens, the shape of the object in the preview window needs to be generated. This is accomplishes via the API call `fz_objt_edit_parm_regen` in the new value function. Note that the function which sets up the dialog template uses a copy of the object's parameters to link the dialog items with parameter values. This is necessary, because the object in the preview window does not exist yet when the dialog edit function is invoked. In the new value function the copy of the parameters is copied into the parameter storage of the object in the preview window, which in return is regenerated.

The OK function

```
fzrt_boolean star_otyp_fuim_ok(
        fz_fuim_tmpl_ptr    fuim_mngr,
        fzrt_ptr            data_ptr
        )
{
        star_otyp_pview_data_td            *pview_data;
```

```
        fz_fuim_stack_get_ptr(fuim_mngr,STAR_OTYP_STACK_PVIEW_DATA,
                (void**)&pview_data);

        star_otyp_fuim_newval(fuim_mngr, data_ptr);

        fz_objt_edit_copy_objt_data_to_windex(pview_data->dst_windex,
                pview_data->dst_obj, pview_data->src_windex,
                pview_data->src_obj, TRUE);

        return(TRUE);
}
```

The OK function is called when the user presses OK to exit the dialog. In this case, the OK function performs the inverse of the load function. It copies the object from the preview window to the object that is actually edited.

The copy control objects function (optional)

```
fzrt_error_td fz_otyp_cbak_copy_cntl_objts(
        long                        windex,
        fz_objt_ptr                 obj,
        fzrt_ptr                    parm,
        long                        *nobj,
        fz_objt_ptr                 *cntrl_objs
        );
```

This function is called when **form•Z** needs to get a copy of the nested control objects from a parametric object. This is done, for example when executing the Extract tool. If an object type does not have any nested objects, this function does not need to be implemented. The function can be called in two modes. The `cntrl_objs` parameter may be passed in as NULL. In this case, the function only needs to determine how many nested control objects there are and pass that value back in the `nobj` parameter. If `cntrl_objs` is passed in, it is an array of pointers to already existing, empty objects, which are ready to be copied into. This can be done with the function call `fz_objt_parm_nested_extract`. This API function is specially designed to handle this copy operation. In addition to copying the content of the nested object, is also makes sure that any attributes, which may have existed when the nested control object was initially created, are set to the host objects attributes. This is necessary, since **form•Z** cannot maintain the attributes of nested control objects. Note that the empty objects are created by **form•Z** and the `cntrl_objs` array is also allocated by **form•Z**, based on the value of the `nobj` parameter when this function is called with cntrl_objs passed as NULL. The copy control objects function of an arbitrary sample object type is shown below.

```
fzrt_error_td my_cntl_objs_copy(
        long                        windex,
        fz_objt_ptr                 obj,
        fzrt_ptr                    parm,
        long                        *nobj,
        fz_objt_ptr                 *cntrl_objs
        )
{
        my_otyp_td          *my_otyp;
        fzrt_error_td       rv = FZRT_NOERR;

        *nobj = 1;
        if ( cntrl_objs != NULL )
        {
                my_otyp = (my_otyp_td *)parm;

                rv = fz_objt_parm_nested_extract(windex,obj,
                                my_otyp->cntl_obj,cntrl_objs[0]);
```

```
        }

        return(rv);
}
```

The key points function (optional)

form•Z calls the key points function to get important points from the object, which may not be part of the object's actual geometry. For example, the key points of an arc are its center, its start and end point. This function is called in two modes. If `pnts` is passed as NULL the function only needs to determine how many key points there are and pass that value back in the `knt` parameter. If `pnts` is passed in, it is an array, allocated by **form•Z**, ready to receive the key points. `knt` needs to be set in both cases. The key points function for the star object type is shown below.

```
fzrt_error_td star_otyp_get_key_pnts(
        long                windex,
        fz_objt_ptr         obj,
        fzrt_ptr            parm,
        long                *knt,
        fz_xyz_td           *pnts
        )
{
        fzrt_error_td       err = FZRT_NOERR;
        star_otyp_parms_td  *star;

        if(knt) *knt = 1;

        if(pnts)
        {
                star = (star_otyp_parms_td*) parm;
                pnts[0] = star->origin;
        }

        return(err);
}
```

The construct smooth function (required if object has both, a facetted and smooth representation, not required otherwise)

```
fzrt_error_td fz_otyp_cbak_cnstr_smod(
        long                        windex,
        fz_objt_ptr                 obj,
        fzrt_ptr                    parm
        );
```

This function is required, when the object has both, a smooth and a facetted representation. In **form•Z**, the revolved object would be of that kind. It is called, when the object is facetted, but **form•Z** needs the smooth equivalent. This is the case, for example, when exporting the object to a SAT file, where the smooth version of the object is much more meaningful, than the facetted. Other examples of the use of this function are derivative operations. The sweep operation, for example, when executed as a smooth sweep, will first try to convert the source and path objects from facetted to smooth, if possible. Assume, the user picked a facetted nurbs curve as the source. The smooth sweep operation will yield a much better result, if the facetted nurbs will be converted to a smooth object, before used in the sweep. The construct smooth function can be efficiently written, by executing the required regeneration function after setting the object's model type option to smooth. This is shown in an example below:

```
fzrt_error_td my_otyp_cnstr_smod(
        long                        windex,
        fz_objt_ptr                 obj,
        fzrt_ptr                    parm
        )
{
        my_otyp_td          *my_otyp;
        fzrt_error_td       rv;

        my_otyp = (my_otyp_td *)parm;

        my_otyp->do_smooth = TRUE;
        rv = my_otyp_regen(windex,obj,parm);

        return(rv);
}
```

The parameter count function (recommended)

```
fzrt_error_td fz_otyp_cbak_parm_count(
        long                *count
        );
```

The parameter count function tells **form•Z**, through how many parameters the object type is defined. This number may not only include the parameters exposed to the user in the dialog interface, but also hidden parameters that may be necessary to store additional information.

```
fzrt_error_td star_parm_count(
        long                *count
        )
{
        fzrt_error_td err = FZRT_NOERR;

        *count = 3;

        return err;
}
```

The parameter info function (recommended)

```
fzrt_error_td  fz_otyp_cbak_parm_get_info2 (
        long                            parm_indx,
        fzrt_UUID_td                    parm_uuid,
        fz_string_td                    parm_name,
        fz_type_enum                    *parm_type,
        fz_fuim_format_int_enum         *parm_format_int,
        fz_fuim_format_float_enum       *parm_format_float,
        fz_fuim_item_type_enum          *parm_fuim_item,
        long                            *parm_range,
        fz_type_td                      *parm_range_min,
        fz_type_td                      *parm_range_max,
        long                            *flags
        );
```

The parameter info function returns a number of informational values about a particular parameter. **form•Z** may invoke this function, for example, to automatically save a parameter's value to file, if the io function is not implemented. **form•Z** typically calls this function by looping over the number of parameters returned by the parameter count function (`fz_otyp_cbak_parm_count`). The only input argument to the info function is `parm_indx`. This is the nth parameter of the object relative to the parameter count. All other function arguments are output arguments. Each parameter needs to have a unique id. This id is returned by the `parm_uuid` argument. The name of the parameter, as it appears in a dialog is returned by `parm_name`. The data type of the parameter is defined by `parm_type`. The interface format for integer and floating point parameters are returned by `parm_format_int` and `parm_format_float`. The choice of dialog interface control by which the parameter is shown in a dialog is defined by `parm_fuim_item`. Whether or not the parameter value has lower and upper range limits is returned by `parm_range`. The min and max ranges are set in `parm_range_min` and `parm_range_max`. The flags argument defines additional attributes of the parameter. They are bit encoded. The allowable bits for the flags argument are :

`FZ_OTYP_PARM_NO_ANIM_BIT`
When this bit is set, **form•Z** cannot animate the parameter.

`FZ_OTYP_PARM_READ_ONLY_BIT`
When this bit is set, the parameter cannot be changed through the `fz_otyp_cbak_parm_set` function.

`FZ_OTYP_PARM_ANIM_LEVEL1_BIT`
When this bit is set, the parameter is considered a good parameter for animation. The parameter usually represents a fluid state. That is, a small change in the parameter causes a small change in the object. This makes it meaningful for animation. It is therefore added to the object's track list, by default, when keyframing the object. An example for such a parameter would be the radius of a sphere.

`FZ_OTYP_PARM_ANIM_LEVEL2_BIT`
When this bit is set, the parameter is considered a secondary parameter for animation. Usually, the parameter represents a state, that is not fluid. That is, a change in the parameter causes the object to take on a significantly different shape. While such a parameter can be animated, it is not added to the object's track list, by default, when keyframing the object. An example for such a parameter would be the type of a spherical object (tetrahedron, hexahedron, octahedron ...).

`FZ_OTYP_PARM_HIDDEN_BIT`
When this bit is set, the parameter is considered hidden, when an automatic dialog interface is build. This may be the case, for example, when a parameter is used for storage of data only, but not for modification by the user.

Note, that all return function arguments are optional. That is, any argument may be NULL, in which case the callback function is expected to ignore the argument. There is also a callback function called `fz_otyp_cbak_parm_get_info`. It is outdated and should not be used. The parameter info function for the star object type is shown below.

```
#define STAR_PARM_BASE_TYPE_UUID \
"\x6d\x8f\x6d\x80\x73\x37\x72\x4b\xbf\x02\x17\x90\xce\x44\x41\x59"
#define STAR_PARM_RADIUS_UUID \
"\x8a\x5e\x98\xfe\xf4\x56\x8c\x4a\xba\x5d\xca\x66\x88\xb2\x87\xd8"
#define STAR_PARM_RAY_RATIO_UUID \
"\x62\xd3\x3b\x97\xdb\x2a\x3f\x46\xaa\xc9\x03\x8d\xe8\x5d\x54\xc8"

enum
{
    STAR_PARM_BASE_TYPE = 0,
    STAR_PARM_RADIUS,
    STAR_PARM_RAY_RATIO,

    STAR_PARM_MAX
};
```

```
fzrt_error_td  star_parm_get_info(
   long                       parm_indx,
   fzrt_UUID_td               parm_uuid,
   fz_string_td               parm_name,
   fz_type_enum               *parm_type,
   fz_fuim_format_int_enum    *parm_format_int,
   fz_fuim_format_float_enum  *parm_format_float,
   fz_fuim_item_type_enum     *parm_fuim_item,
   long                       *parm_range,
   fz_type_td                 *parm_range_min,
   fz_type_td                 *parm_range_max,
   long                    *flags
   )
{
   fzrt_error_td err = FZRT_NOERR;
   char          str[256];
   long          lval;
   double        dval;

   switch(parm_indx)
   {
      case STAR_PARM_BASE_TYPE:
         if (parm_uuid)
            fzrt_UUID_copy(STAR_PARM_BASE_TYPE_UUID, parm_uuid);
         if (parm_name)
            strcpy(parm_name, "Base Type");
         if (parm_type)
            *parm_type = FZ_TYPE_LONG;
         if (parm_format_int)
            *parm_format_int = FZ_FUIM_FORMAT_INT_DEFAULT;
         if (parm_format_float)
            *parm_format_float = FZ_FUIM_FORMAT_FLOAT_DEFAULT;
         if (parm_range)
            *parm_range = FZ_FUIM_RANGE_MIN |
                          FZ_FUIM_RANGE_MIN_INCL |
                          FZ_FUIM_RANGE_MAX |
                          FZ_FUIM_RANGE_MAX_INCL;
         if (parm_range_min)
         {
            lval = 0;
            fz_type_set_long(&lval, parm_range_min);
         }
         if (parm_range_max)
         {
            lval = 7;
            fz_type_set_long(&lval, parm_range_max);
         }
         if (flags)
            *flags = 0;
         if ( parm_fuim_item )
            *parm_fuim_item = FZ_FUIM_ITEM_MENU;
      break;

      case STAR_PARM_RADIUS:
         if (parm_uuid)
            fzrt_UUID_copy(STAR_PARM_RADIUS_UUID, parm_uuid);
         if (parm_name)
            strcpy(parm_name, "Radius");
         if (parm_type)
            *parm_type = FZ_TYPE_DOUBLE;
         if (parm_format_int)
            *parm_format_int = FZ_FUIM_FORMAT_INT_DEFAULT;
         if (parm_format_float)
            *parm_format_float = FZ_FUIM_FORMAT_FLOAT_DISTANCE;
```

```
        if (parm_range)
            *parm_range = FZ_FUIM_RANGE_MIN;
        if (parm_range_min)
        {   dval = 0;
            fz_type_set_double(&dval, parm_range_min);
        }
        if (flags)
            *flags = 0;
        if ( parm_fuim_item )
            *parm_fuim_item = FZ_FUIM_ITEM_TEXT;
    break;

    case STAR_PARM_RAY_RATIO:
        if (parm_uuid)
            fzrt_UUID_copy(STAR_PARM_RAY_RATIO_UUID, parm_uuid);
        if (parm_name)
            strcpy(parm_name, "Ray Ratio");
        if (parm_type)
            *parm_type = FZ_TYPE_DOUBLE;
        if (parm_format_int)
            *parm_format_int = FZ_FUIM_FORMAT_INT_DEFAULT;
        if (parm_format_float)
            *parm_format_float = FZ_FUIM_FORMAT_FLOAT_PERCENT;
        if (parm_range)
            *parm_range = FZ_FUIM_RANGE_MIN |
                          FZ_FUIM_RANGE_MIN_INCL |
                          FZ_FUIM_RANGE_MAX |
                          FZ_FUIM_RANGE_MAX_INCL;
        if (parm_range_min)
        {   dval = 0;
            fz_type_set_double(&dval, parm_range_min);
        }
        if (parm_range_max)
        {   dval = 1;
            fz_type_set_double(&dval, parm_range_max);
        }
        if (flags)
            *flags = 0;
        if ( parm_fuim_item )
            *parm_fuim_item = FZ_FUIM_ITEM_SLIDER_TEXT;
    break;
  }

  return err;
}
```

The parameter get state name function (recommended)

```
fzrt_error_td fz_otyp_cbak_parm_get_state_str(
        fzrt_UUID_td        parm_uuid,
        long                which_state,
        fz_string_td        str
        );
```

This function should be implemented, if an integer or boolean parameter is displayed as a menu item in a dialog. Given the parameter's uuid, this function returns the nth string associated with the nth state of that parameter. This function may also be used if the parameter is shown through a set of radio buttons. The get state name function is mainly used when **form•Z** automatically builds a dialog interface and by the animation track editor interface.

```
fzrt_error_td  star_parm_get_state_str(
```

```
	fzrt_UUID_td		parm_uuid,
	long				which_state,
	fz_string_td		str
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	if (fzrt_UUID_is_equal(parm_uuid, STAR_PARM_BASE_TYPE_UUID))
	{
		switch ( which_state )
		{
			case 0 : strcpy(str,"Tetrahedron");		break;
			case 1 : strcpy(str,"Hexahedron");		break;
			case 2 : strcpy(str,"Octahedron");		break;
			case 3 : strcpy(str,"Dodecahedron");		break;
			case 4 : strcpy(str,"Icosahedron");		break;
			case 5 : strcpy(str,"Soccer Ball");		break;
			case 6 : strcpy(str,"Geodesic Level 1");	break;
			case 7 : strcpy(str,"Geodesic Level 2");	break;

		}
	}

	return err;
}
```

The get parameter function (recommended)

```
fzrt_error_td	fz_otyp_cbak_parm_get (
	long				windex,
	fz_objt_ptr			objt,
	fzrt_UUID_td			parm_uuid
	);
```

form•Z calls this function to get the value of a parameter, which is identified by the `parm_uuid` argument.

```
fzrt_error_td  star_parm_get(
	long				windex,
	fz_objt_ptr			objt,
	fzrt_UUID_td			parm_uuid,
	fz_type_td			*data
	)
{
	fzrt_error_td			err = FZRT_NOERR;
	star_otyp_parms_td		*star;
	long				lval;
	double				dval;

	fz_objt_parm_get_data(windex,objt,(fzrt_ptr*)&star);

	if ( fzrt_UUID_is_equal(parm_uuid, STAR_PARM_BASE_TYPE_UUID))
	{
		lval = star->base_type;
		fz_type_set_long(&lval, data);
	}
	else if ( fzrt_UUID_is_equal(parm_uuid, STAR_PARM_RADIUS_UUID))
	{
		dval = star->radius;
		fz_type_set_double(&dval, data);
	}
```

```
	else if ( fzrt_UUID_is_equal(parm_uuid, STAR_PARM_RAY_RATIO_UUID))
	{
		dval = star->rad_ratio;
		fz_type_set_double(&dval, data);
	}

	return err;
}
```

The set parameter function (recommended)

```
fzrt_error_td fz_otyp_cbak_parm_set (
	long				windex,
	fz_objt_ptr			objt,
	fzrt_UUID_td			parm_uuid
	);
```

form•Z calls this function to set the value of a parameter, which is identified by the `parm_uuid` argument.

```
fzrt_error_td  star_parm_set(
	long				windex,
	fz_objt_ptr			objt,
	fzrt_UUID_td			parm_uuid,
	fz_type_td			*data
	)
{
	fzrt_error_td			err = FZRT_NOERR;
	star_otyp_parms_td		*star;
	long				lval;
	double				dval;

	fz_objt_parm_get_data(windex,objt,(fzrt_ptr*)&star);

	if (fzrt_UUID_is_equal(parm_uuid, STAR_PARM_BASE_TYPE_UUID))
	{
		fz_type_get_long(data, &lval);
		if (lval >= 0 && lval <= 7)
			star->base_type = lval;
	}
	else if (fzrt_UUID_is_equal(parm_uuid, STAR_PARM_RADIUS_UUID))
	{
		fz_type_get_double(data, &dval);
		if (dval > 0)
			star->radius = dval;
	}
	else if (fzrt_UUID_is_equal(parm_uuid, STAR_PARM_RAY_RATIO_UUID))
	{
		fz_type_get_double(data, &dval);
		if (dval >= 0.0 && dval <= 1.0)
			star->rad_ratio = dval;
	}

	return err;
}
```

Note, that there is a close relationship between the size argument of the object type info function (`fz_otyp_cbak_info`) and the parameter count, parameter info, parameter get and parameter set functions. A plugin may return -1 as the size in `fz_otyp_cbak_info`. In this case, the parameter count and parameter info functions must be defined. **form•Z** will use those two function to calculate the size of the parameter block necessary to store an object's parameter data. If this is the case, the parameter get, parameter set io stream and dialog template functions should NOT be implemented. In other words, a -1 size tells **form•Z**, that this object type is a simple type, where the parameter access is automated as much as possible. If the plugin chooses to maintain its own data structure that represents the objects parameter data, such as the star example, it is recommended that all parameter functions are implemented. This ensures, that the object type is fully integrated in **form•Z**.

Working with nested objects

For some object types, it may be necessary to store one or more entire objects in the parameter data block of the object. This is the case, for example, with the **form•Z** sweep object type. It stores the sweep source and the sweep path as a nested control object in its parameter block. Some special rules apply for dealing with nested control objects.

Creating a nested control object

The object which becomes the nested control object is usually supplied to a function which creates an object of the given type. For example, a plugin may create an object type and a tool. Executing the tool may involve picking an object, which becomes the nested control object. The picked object cannot be stored in the object parameter block directly. A new, independent object must be created with the API function `fz_objt_indp_init` and the picked object must be copied into the independent object. Nested control objects must be independent objects. That is, they cannot not part of the project's main object list and it is the responsibility of the object type plugin to maintain the nested object. Below is the click function of the frame tool, which creates a frame object type. The frame object takes a base object, and constructs circular pipes along each segment, which meet at spheres, placed at each point of the base object. The click function performs the following main steps:

1. It constructs a new empty object. This will become the frame object.
2. It initializes the new object as a frame object. This allocates the parameter data block.
3. It constructs a new independent object and copies the picked object into it.
4. It sets the default parameters for the frame object and regenerates its shape.

Note that the structure used to store the parameters for a frame object contains a pointer to a modeling object. This is the nested control object.

```
fzrt_error_td frame_tool_click(
       long                        windex,
       fzrt_point                  *where,
       fz_xyz_td                          *where_3d,
       fz_map_plane_td             *map_plane,
       fz_fuim_click_enum          clicks,
       long                        click_count,
       fzrt_boolean                *click_handled,
       fz_fuim_click_wait_enum     *click_next,
```

```
    fzrt_boolean              *done)
{
    fzrt_error_td         err;
    fzrt_zone_ptr         zone_ptr;
    fz_objt_ptr           obj,pick_obj;
    frame_otyp_parms_td   *frame;
    fz_model_pick_enum    pkind;
    long                  i,npick;
    fzrt_boolean          pre_pick = FALSE;

    /* CHECK FOR PRE PICKED OBJECTS */
    fz_model_pick_get_count(windex,&npick);
    for(i = 0; i < npick; i++)
    {   fz_model_pick_get_data(windex,i,&pkind,NULL,NULL,NULL);
        if ( pkind == FZ_MODEL_PICK_OBJT )
        {
            pre_pick = TRUE;
            break;
        }
    }

    /* POST PICKING */
    if ( i >= npick )
    {   fz_model_pick(windex,where,FZ_MODEL_PICK_OBJT);

        fz_model_pick_get_count(windex, &npick);
        for(i = 0; i < npick; i++)
        {   fz_model_pick_get_data(windex,i,&pkind,NULL,NULL,NULL);
            if ( pkind == FZ_MODEL_PICK_OBJT )  break;
        }
        if ( i >= npick ) *done = TRUE;
    }

    if ( i < npick )
    {
        for(i = 0; i < npick; i++)
        {   fz_model_pick_get_data(windex,i,&pkind,NULL,&pick_obj,NULL);
            if ( pkind != FZ_MODEL_PICK_OBJT ) continue;

            /* CREATE A NEW OBJECT */
            if((err = fz_objt_cnstr_objt_new(windex,&obj)) == FZRT_NOERR )
            {
                /* INIT THE NEW OBJECT AS A FRAME OBJECT */
                if( (err = fz_objt_parm_init_data(windex,
                            obj,FRAME_OTYP_UUID,(fzrt_ptr*)&frame)) == FZRT_NOERR )
                {

                    /* CREATE AN INDEPENDENT OBJECT AND */
                    /* COPY THE PICKED OBJECT INTO IT */
                    fz_objt_get_zone_ptr(windex,obj,&zone_ptr);
                    if((err = fz_objt_indp_init(windex,zone_ptr,
                            &frame->base_obj)) == FZRT_NOERR )
                    {
                        if((err = fz_objt_edit_copy_objt_geom(windex,
                                pick_obj,frame->base_obj)) == FZRT_NOERR )
                        {
                            /* REGENERATE FRAME OBJECT AND ADD IT TO THE PROJECT */
                            frame->do_smooth       = _frame_tool_opts->do_smooth;
                            frame->radius          = _frame_tool_opts->radius;
                            if((err = fz_objt_edit_parm_regen(windex,obj)) == FZRT_NOERR )
                            {
                                err = fz_objt_add_objt_to_project(windex,obj);
                            }
                        }
                    }
```

```
            }

            if ( err != FZRT_NOERR )
            {   fz_objt_edit_delete_objt(windex,obj);
            }
        }
    }

    *done = TRUE;

    /* CLEAR PICK BUFFER */
    if ( pre_pick == FALSE ) fz_model_pick_clear(windex);
  }

  return FZRT_NOERR;
}
```

Deleting a nested control object

In the finit function of the object type, a nested control object must be deleted with the API call `fz_objt_indp_finit`. This is shown in the finit function of the frame object type.

```
fzrt_error_td frame_otyp_finit (
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm
        )
{
        frame_otyp_parms_td        *frame;
        frame = (frame_otyp_parms_td *)parm;

        if ( frame->base_obj )
        {       fz_objt_indp_finit(windex,&frame->base_obj);
        }

        return(FZRT_NOERR);
}
```

Nested control objects in the io stream function

When writing or reading a nested control object in the io stream callback function of the object type, the API function `fz_objt_io` must be called. When writing, the nested object can be passed directly to this API function. When reading, the io stream function must first create a new, independent object. It can be assumed that the object passed in the io stream function when reading, is an object, whose parameter data block has been initialized to default values. When a nested object is part of the parameter block, the object pointer is usually initialized to NULL. The io stream function for the frame object is shown below.

```
fzrt_error_td frame_otyp_iost(
    long                    windex,
    fz_objt_ptr             obj,
    fzrt_ptr                parm,
    fz_iost_ptr             iost,
    fz_iost_dir_td_enum     dir,
    fzpl_vers_td  *const    version,
    unsigned long           size
    )
```

```
{
   fzrt_error_td       err = FZRT_NOERR;
   fzrt_zone_ptr       zone_ptr;
   frame_otyp_parms_td    *frame;

   frame = (frame_otyp_parms_td *)parm;

   if ( dir == FZ_IOST_WRITE ) *version = 0;

   if((err = fz_iost_boolean(iost,&frame->do_smooth,1)) == FZRT_NOERR &&
      (err = fz_iost_double(iost,&frame->radius,1)) == FZRT_NOERR )
   {
      if ( dir == FZ_IOST_READ )
      {
         fz_objt_get_zone_ptr(windex,obj,&zone_ptr);
         err = fz_objt_indp_init(windex,zone_ptr,&frame->base_obj);
      }

      if ( err == FZRT_NOERR && frame->base_obj != NULL )
      {  err = fz_objt_io(windex,frame->base_obj,iost,dir);
      }
   }

   return(err);
}
```

2.8.5 Palette Plugins

A palette is a floating window that contains an interface for a feature or set of related features. The interface is composed of variety of interface elements (buttons, radio buttons, check boxes, lists etc.) provided by the **form•Z** interface manager (fuim). Palette plugins are extensions that complement the **form•Z** palettes and behave consistently with the **form•Z** palettes.

Palettes are available in ***system*** and ***project*** levels. System palettes are global in nature and do not require a project window index while project palettes require a project or window index and are expected to operate on project information for provided project, Palettes are flexible extensions as a lot of functionality can be included in a palette. The interface of the palette is defined by the extension through a fuim template. A description of fuim templates can be found in section 2.6 and in the **form•Z** API reference.

The names of palette plugins are added to a group near the bottom of the Palettes menu. As with all other palette names in this menu, selecting a palette name toggles the visibility of the palette. That is, if the palette is visible, then it is hidden and vice versa. Palettes that are visible are indicated by a check mark in the menu before the name. All palettes appear in the Key Shortcuts Manage dialog so that they may have key shortcuts assigned for them to open and close the palette. Note that if it is desirable to have the ability for the user to assign a key shortcut for individual items within the interface of the palette, then a separate palette plugin must be implemented for this action.

The Samples directory in the **form•Z** SDK folder contains a folder named Palettes that contains an example of a palette plugin named my_view_palette. This example creates a project palette with buttons for selecting a standard view type. This sample can be very valuable as both starting points for development as well as examples of how the functions work.

Palette plugin type and registration

Palette plugins are registered with the plugin type identifier `FZ_PALT_EXTS_TYPE` and version of `FZ_PALT_EXTS_VERSION`. System palette plugins must implement the function set `fz_palt_cbak_syst_fset` and project palette plugins must implement the function set `fz_palt_cbak_proj_fset`.

The following example shows the registration of a palette plugin and the binding of a system palette and project palette function sets to the plugin. This registration is performed in the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3. Note that the normal usage is to register a system palette or a project palette (not both). Palette plugins may also provide the `fz_notf_cbak_fset` function set to be notified when changes occur within **form•Z**.

```
fzrt_error_td my_palt_register_plugins()
{
	fzrt_error_td			err = FZRT_NOERR;
	char				my_name[256];

	/* Get the title string from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_name)) ==
FZRT_NOERR)
	{
		/* register the palette plugin */
		err = fzpl_glue->fzpl_plugin_register(
			MY_PLUGIN_UUID,
			my_name,
			MY_PLUGIN_VERSION,
```

```
                    MY_PLUGIN_VENDOR,
                    MY_PLUGIN_URL,
                    FZ_PALT_EXTS_TYPE,
                    FZ_PALT_EXTS_VERSION,
                    my_plugin_error_string_func,
                    0,
                    NULL,
                    &my_plugin_runtime_id);

            /* add a system palette callback function set */
            if ( err == FZRT_NOERR )
            {
                    err = fzpl_glue->fzpl_plugin_add_fset(
                            my_plugin_runtime_id,
                            FZ_PALT_CBAK_SYST_FSET_TYPE,
                            FZ_PALT_CBAK_SYST_FSET_VERSION,
                            FZ_PALT_CBAK_SYST_FSET_NAME,
                            FZPL_TYPE_STRING(fz_palt_cbak_syst_fset),
                            sizeof (fz_palt_cbak_syst_fset),
                            my_palt_cbak_syst_fill_fset, FALSE);
            }
            /* add a project palette callback function set */
            if ( err == FZRT_NOERR )
            {
                    err = fzpl_glue->fzpl_plugin_add_fset(
                            my_plugin_runtime_id,
                            FZ_PALT_CBAK_PROJ_FSET_TYPE,
                            FZ_PALT_CBAK_PROJ_FSET_VERSION,
                            FZ_PALT_CBAK_PROJ_FSET_NAME,
                            FZPL_TYPE_STRING(fz_palt_cbak_proj_fset),
                            sizeof (fz_palt_cbak_proj_fset),
                            my_palt_cbak_proj_fill_fset, FALSE);
            }
    }
    return (err);
}
```

2.8.5.1 System Palette

System palette plugins are implemented by the plugin by providing the call back function set `fz_palt_cbak_syst_fset`. There are seven functions in this function set. The following example shows the assignment of the plugins defined functions into the function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above. Note that some of these functions are optional hence a plugin would rarely implement all functions.

```
fzrt_error_td my_palt_cbak_syst_fill_fset (
            const fzpl_fset_def_ptr fset_def,
            fzpl_fset_td * const fset )
{
    fzrt_error_td                   err = FZRT_NOERR;
    fz_palt_cbak_syst_fset          *palt_syst;

    /* check that the provided function set is of the expected version */
    err = fzpl_glue->fzpl_fset_def_check ( fset_def,
            FZ_PALT_CBAK_SYST_FSET_VERSION,
            FZPL_TYPE_STRING(fz_palt_cbak_syst_fset),
            sizeof ( fz_palt_cbak_syst_fset ),
            FZPL_VERSION_OP_NEWER );

    if ( err == FZRT_NOERR )
    {
            /* fill function set structure with local plugins functions */
```

```
		palt_syst = (fz_palt_cbak_syst_fset *)fset;

		palt_syst->fz_palt_cbak_syst_init       = my_palt_syst_init;
		palt_syst->fz_palt_cbak_syst_finit      = my_palt_syst_finit;
		palt_syst->fz_palt_cbak_syst_name       = my_palt_syst_name;
		palt_syst->fz_palt_cbak_syst_uuid       = my_palt_syst_uuid;
		palt_syst->fz_palt_cbak_syst_help       = my_palt_syst_help;
		palt_syst->fz_palt_cbak_syst_iface_tmpl = my_palt_syst_iface_tmpl;
		palt_syst->fz_palt_cbak_syst_pref_io    = my_palt_syst_pref_io;
	}

	return err;
}
```

The initialization function (optional)

```
fzrt_error_td fz_palt_cbak_syst_init (
	void
	);
```

This function is called by **form•Z** once when the plugin is successfully loaded and registered. The initialization function is where the plugin should initialize any data that may be needed by the other functions in the function set.

```
fzrt_error_td my_palt_syst_init(
	void
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/** Do initialization here **/

	return(err);
}
```

The finalization function (optional)

```
fzrt_error_td fz_palt_cbak_syst_finit(
	void
	);
```

This function is called by **form•Z** once when the plugin is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to free and memory allocated in the initialization function or during the life of the palette.

```
fzrt_error_td my_palt_syst_finit(
	void
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/** Free any initalized data here **/

	return(err);
}
```

The name function (recommended)

```
fzrt_error_td fz_palt_cbak_syst_name (
	char		*name,
```

```
        long          max_len
        );
```

This function is called by **form•Z** at various times to get the name of the palette. It is recommended that the name is stored in a .fzr file so that it is localizable. The name is the name that is added to the palette menu and is used as the tittle for the palette.

```
fzrt_error_td my_palt_syst_name(
        char          *name,
        long          max_len
        )
{
        fzrt_error_td       err = FZRT_NOERR;
        char                my_str[256];

        /* Get the title string “My Palette” from the plugin’s resource file */
        if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_str)) ==
FZRT_NOERR)
        {
               /* copy the string to the name parameter */
               strncpy(name,  my_str, max_len);
        }
        return(err);
}
```

The uuid function (recommended)

```
fzrt_error_td fz_palt_cbak_syst_uuid
        fzrt_UUID_td  uuid
        );
```

This function is called by **form•Z** at various times to get the uuid of the palette. This unique id is used by **form•Z** to distinguish the palette from other palettes.

```
#define MY_PALT_UUID
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td  my_palt_syst_uuid (
        fzrt_UUID_td  uuid
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        /* copy constant UUID to into the uuid parameter */
        fzrt_UUID_copy(MY_PALT_UUID, uuid);

        return(err);
}
```

The help function (recommended)

```
fzrt_error_td fz_palt_cbak_syst_help (
        char          *help,
        long          max_len,
        );
```

This function is called by **form•Z** to display a help string that describes the detail of what the palette does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` bytes of data. It is recommended that the palette name is stored in .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 512 bytes (characters).

```
fzrt_error_td my_palt_syst_help(
      char          *help,
      long          max_len
      )
{
      fzrt_error_td       err = FZRT_NOERR;
      char                my_str[512];

      /* Get the help string from the plugin's resource file */
      if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) ==
FZRT_NOERR)
      {
            /* copy the string to the help parameter */
            strncpy(help,  my_str, max_len);
      }
      return(err);
}
```

The interface template function (required)

```
fzrt_error_td fz_palt_cbak_syst_iface_tmpl (
      fz_fuim_tmpl_ptr          tmpl_ptr,
      fzrt_ptr                  tmpl_data
      )
```

This function is called by **form•Z** when the interface for the palette is needed. The **form•Z** interface template functions should be called to construct the interface of the palette in this function. Please see section 2.6 for more details on the fuim template functions. The full fuim template documentation can be found in the API reference.

The following sample is a template for 3 buttons grouped inside a border with a title.

```
#define MY_STRINGS         1

enum
{     MY_STRING_NAME = 1,
      MY_STRING_TYPE,
      MY_STRING_1,
      MY_STRING_2,
      MY_STRING_3
};

enum
{     MY_BUTTON1=1,
      MY_BUTTON2,
      MY_BUTTON3
};

fzrt_error_td my_palt_syst_iface_tmpl (
      fz_fuim_tmpl_ptr          tmpl_ptr,
      fzrt_ptr                  tmpl_data
      )
{
      fzrt_error_td       err;
      short               gindx;
      char                str[256];

      /* get the options title from plugin's resource file */
      fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS, MY_STRING_NAME, str);
```

```
        if((err = fz_fuim_tmpl_init(tmpl_ptr, str,
                FZ_FUIM_NONE, MY_PALT_OPTS_UUID, 0)) == FZRT_NOERR)
        {
                /* create a static text item */
                fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS, MY_STRING_TYPE,
                                                str);
                gindx = fz_fuim_new_text_static(tmpl_ptr, -1, FZ_FUIM_NONE,
                                FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_EQSZ, str, NULL,
                        NULL);

                        /* create a button */
                        fzrt_fzr_get_string(my_rfzr_refid,
                                MY_STRINGS, MY_STRING_1, str);
                        fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON1,
                                FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);

                        /* create a button */
                        fzrt_fzr_get_string(my_rfzr_refid,
                                MY_STRINGS, MY_STRING_2, str);
                        fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON2,
                                FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);

                        /* create a button */
                        fzrt_fzr_get_string(my_rfzr_refid,
                                MY_STRINGS, MY_STRING_3, str);
                        fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON3,
                                FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);
        }

        return (err);
}
```

The preferences IO function (optional)

```
fzrt_error_td fz_palt_cbak_syst_pref_io (
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum                     dir,
        fzpl_vers_td * const            version,
        unsigned long                   size
        );
```

form•Z calls this function to read and write any palette specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a palette's data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
typedef struct my_palette_td /* my palette's global pref data */
{
	long value1,value2,value3,value4,value5;
	...
}my_palette_td;
my_palette_td*		my_palette;

fzrt_error_td my_palt_syst_pref_io(
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum				dir,
	fzpl_vers_td * const			version,
	unsigned long				size
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	if ( dir == FZ_IOST_WRITE ) *version = 1;

	err = fz_iost_one_long(iost,&my_palette->value1);
	if(err == FZRT_NOERR)
	{	err = fz_iost_one_long(iost,&my_palette->value2);
		if(err == FZRT_NOERR)
		{	err = fz_iost_one_long(iost,&my_palette->value3);
			if(err == FZRT_NOERR)
			{	err = fz_iost_one_long(iost,&my_palette->value4);

				if(*version >= 1)
				{	err = fz_iost_one_long(iost,
							&my_palette->value5);
				}
			}
		}
	}

	return(err);
}
```

2.8.5.2 Project Palette

Project palette plugins are implemented by the call back function set `fz_palt_cbak_proj_fset`. There are seven functions in this function set. The following example shows the assignment of the plugins defined functions into the function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above. Note that some of these functions are optional hence a plugin would never implement all of these functions.

```
fzrt_error_td my_palt_cbak_proj_fill_fset (
		const fzpl_fset_def_ptr fset_def,
		fzpl_fset_td * const fset )
{
	fzrt_error_td			err = FZRT_NOERR;
	fz_palt_cbak_proj_fset		*palt_proj;

	/* check that the provided function set is of the expected version */
	err = fzpl_glue->fzpl_fset_def_check ( fset_def,
```

```
		FZ_PALT_CBAK_PROJ_FSET_VERSION,
		FZPL_TYPE_STRING(fz_palt_cbak_proj_fset),
		sizeof ( fz_palt_cbak_proj_fset ),
		FZPL_VERSION_OP_NEWER );

	if ( err == FZRT_NOERR )
	{
		/* fill function set structure with local plugins functions */
		palt_proj = (fz_palt_cbak_proj_fset *)fset;

		palt_proj->fz_palt_cbak_proj_init		= my_palt_proj_init;
		palt_proj->fz_palt_cbak_proj_finit		= my_palt_proj_finit;
		palt_proj->fz_palt_cbak_proj_info		= my_palt_proj_info;
		palt_proj->fz_palt_cbak_proj_name		= my_palt_proj_name;
		palt_proj->fz_palt_cbak_proj_uuid		= my_palt_proj_uuid;
		palt_proj->fz_palt_cbak_proj_help		= my_palt_proj_help;
		palt_proj->fz_palt_cbak_proj_iface_tmpl	= my_palt_proj_iface_tmpl;
		palt_proj->fz_palt_cbak_proj_pref_io		= my_palt_proj_pref_io;
		palt_proj->fz_palt_cbak_proj_data_io		= my_palt_proj_data_io;
		palt_proj->fz_palt_cbak_proj_wind_data_io	= my_palt_proj_wind_data_io;
	}

	return err;
}
```

The initialization function (optional)

```
fzrt_error_td fz_palt_cbak_proj_init (
	void
	);
```

This function is called by **form•Z** once when the plugin is successfully loaded and registered. The initialization function is where the plugin should initialize any data that may be needed by the other functions in the function set.

```
fzrt_error_td my_palt_proj_init(
	void
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/** Do initialization here **/

	return(err);
}
```

The finalization function (optional)

```
fzrt_error_td fz_palt_cbak_proj_finit(
	void
	);
```

This function is called by **form•Z** once when the plugin is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to free and memory allocated in the initialization function or during the life of the palette.

```
fzrt_error_td my_palt_proj_finit(
	void
	)
{
	fzrt_error_td		err = FZRT_NOERR;
```

```
        /** Free any initalized data here **/

        return(err);
}
```

The information function (required)

```
fzrt_error_td fz_palt_cbak_proj_info (
        fz_proj_level_enum          *level
        );
```

This function is called by **form•Z** once when the plugin is successfully loaded and registered immedialty after the initialization function (if provided).

The `level` parameter indicates the context of the tool. **form•Z** uses the value in this parameter to determine when the palette should be shown and when it should be updated. The following are the available values:

`FZ_PROJ_LEVEL_MODEL`: Indicates that the tool operates on the projects modeling content (objects for example).
`FZ_PROJ_LEVEL_MODEL_WIND`: Indicates that the tool operates on modeling window specific content (views for example) of modeling windows.
`FZ_PROJ_LEVEL_DRAFT`: Indicates that the tool operates on the projects drafting content (elements for example).
`FZ_PROJ_LEVEL_DRAFT_WIND`: Indicates that the tool operates on drafting window specific content (views for example) of drafting windows.

```
fzrt_error_td my_palt_proj_info(
        fz_proj_level_enum          *level
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        *level = FZ_PROJ_LEVEL_MODEL;

        return(err);
}
```

The name function (recommended)

```
fzrt_error_td fz_palt_cbak_proj_name (
        char          *name,
        long          max_len
        );
```

This function is called by **form•Z** at various times to get the name of the palette. It is recomended that the name is stored in a .fzr file so that it is localizable. The name is the name that is added to the palette menu and is used as the tittle for the palette.

```
fzrt_error_td my_palt_proj_name(
        char          *name,
        long          max_len
        )
{
        fzrt_error_td       err = FZRT_NOERR;
        char                my_str[256];
```

```
        /* Get the title string from the plugin's resource file */
        if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_str)
           ) == FZRT_NOERR)
        {
                /* copy the string to the name parameter */
                strncpy(name,  my_str, max_len);
        }
        return(err);
}
```

The uuid function (recommended)

```
fzrt_error_td fz_palt_cbak_proj_uuid
        fzrt_UUID_td  uuid
        );
```

This function is called by **form•Z** at various times to get the uuid of the palette. This unique id is used by formZ to distinguish the palette from other palettes.

```
#define MY_PALT_UUID
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td  my_palt_proj_uuid (
        fzrt_UUID_td  uuid
        )
{
        fzrt_error_td        err = FZRT_NOERR;

        /* copy constant UUID to into the uuid parameter */
        fzrt_UUID_copy(MY_PALT_UUID, uuid);

        return(err);
}
```

The help function (recommended)

```
fzrt_error_td fz_palt_cbak_proj_help (
        char          *help,
        long          max_len
        );
```

This function is called by **form•Z** to display a help string that describes the detail of what the palette does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` bytes of data. It is recommended that the palette name is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 512 bytes (characters).

```
fzrt_error_td my_palt_proj_help(
        char          *help,
        long          max_len
        )

{
        fzrt_error_td        err = FZRT_NOERR;
        char                 my_str[512];

        /* Get the help string from the plugin's resource file */
        if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) ==
FZRT_NOERR)
```

```
        {
                /* copy the string to the help parameter */
                strncpy(help,  my_str, max_len);
        }
        return(err);
}
```

The interface template function (required)

```
fzrt_error_td fz_palt_cbak_proj_iface_tmpl (
        long                        windex,
        fz_fuim_tmpl_ptr            tmpl_ptr,
        fzrt_ptr                    tmpl_data
        )
```

This function is called by **form•Z** when the interface for the palette is needed. The **form•Z** interface template functions should be called to construct the interface of the palette in this function. Please see section 2.6 for more details on the fuim template functions. The full fuim template documentation can be found in the API reference.

The following sample is a template for 3 buttons grouped inside a border with a title.

```
#define MY_STRINGS          1

enum
{       MY_STRING_NAME = 1,
        MY_STRING_TYPE,
        MY_STRING_1,
        MY_STRING_2,
        MY_STRING_3
};

enum
{       MY_BUTTON1=1,
        MY_BUTTON2,
        MY_BUTTON3
};

fzrt_error_td my_palt_proj_iface_tmpl (
        long                        windex,
        fz_fuim_tmpl_ptr            tmpl_ptr,
        fzrt_ptr                    tmpl_data
        )
{
        fzrt_error_td       err;
        short               gindx;
        char                str[256];

        /* get the options title from plugin's resource file */
        fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS, MY_STRING_NAME, str);
        if((err = fz_fuim_tmpl_init(tmpl_ptr, str,
                FZ_FUIM_NONE, MY_PALT_OPTS_UUID, 0)) == FZRT_NOERR)
        {
                /* create a static text item */
                fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS, MY_STRING_TYPE,
str);
                gindx = fz_fuim_new_text_static(tmpl_ptr, -1, FZ_FUIM_NONE,
                            FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_EQSZ, str, NULL,
                      NULL);

                      /* create a button */
                      fzrt_fzr_get_string(my_rfzr_refid,
                            MY_STRINGS, MY_STRING_1, str);
```

```
                    fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON1,
                          FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);

                    /* create a button */
                    fzrt_fzr_get_string(my_rfzr_refid
                          MY_STRINGS, MY_STRING_2, str);
                    fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON2,
                          FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);

                    /* create a button */
                    fzrt_fzr_get_string(my_rfzr_refid,
                          MY_STRINGS, MY_STRING_3, str);
                    fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON3,
                          FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);
       }

       return (err);
}
```

The preferences IO function (optional)

```
fzrt_error_td fz_palt_cbak_proj_pref_io (
       fz_iost_ptr                    iost,
       fz_iost_dir_td_enum                   dir,
       fzpl_vers_td * const           version,
       unsigned long                  size
       );
```

form•Z calls this function to read and write any palette specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a palettes data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value..

```
typedef struct my_palette_td /* my palette's global pref data */
{
       long value1,value2,value3,value4,value5;
```

```
	...
}my_palette_td;
my_palette_td*		my_palette;

fzrt_error_td my_palt_proj_pref_io(
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum			dir,
	fzpl_vers_td * const			version,
	unsigned long				size
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	if ( dir == FZ_IOST_WRITE ) *version = 1;

	err = fz_iost_one_long(iost,&my_palette->value1);
	if(err == FZRT_NOERR)
	{	err = fz_iost_one_long(iost,&my_palette->value2);
		if(err == FZRT_NOERR)
		{	err = fz_iost_one_long(iost,&my_palette->value3);
			if(err == FZRT_NOERR)
			{	err = fz_iost_one_long(iost,&my_palette->value4);

				if(*version >= 1)
				{	err = fz_iost_one_long(iost,
						&my_palette->value5);
				}
			}
		}
	}

	return(err);
}
```

The project data IO function (optional)

```
fzrt_error_td fz_palt_cbak_proj_data_io (
	long					windex,
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum			dir,
	fzpl_vers_td * const			version,
	unsigned long				size
	);
```

form•Z calls this function to read and write any palette specific project data to a **form•Z** project file. This function is called once when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** project file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a palettes data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_palt_proj_data_io (
       long                              windex,
       fz_iost_ptr                       iost,
       fz_iost_dir_td_enum                      dir,
       fzpl_vers_td * const              version,
       unsigned long                     size
       )
{
       fzrt_error_td       err = FZRT_NOERR;

       if ( dir == FZ_IOST_WRITE ) *version = 1;

       err = fz_iost_one_long(iost,&my_palette->value1);
       if(err == FZRT_NOERR)
       {      err = fz_iost_one_long(iost,&my_palette->value2);
              if(err == FZRT_NOERR)
              {      err = fz_iost_one_long(iost,&my_palette->value3);
                     if(err == FZRT_NOERR)
                     {      err = fz_iost_one_long(iost,&my_palette->value4);

                            if(*version >= 1)
                            {      err = fz_iost_one_long(iost,
                                          &my_palette->value5);
                            }
                     }
              }
       }

       return(err);
}
```

The project window data IO function (optional)

```
fzrt_error_td fz_palt_cbak_proj_wind_data_io (
       long                              windex,
       fz_iost_ptr                       iost,
       fz_iost_dir_td_enum                      dir,
       fzpl_vers_td * const              version,
       unsigned long                     size
       );
```

form•Z calls this function to read and write any palette specific project window data to a **form•Z** project file. This function is called once for each window in the project when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** Project file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a palettes data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_palt_proj_wind_data_io (
	long					windex,
	fz_iost_ptr				iost,
	fz_iost_dir_td_enum			dir,
	fzpl_vers_td * const			version,
	unsigned long				size
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	if ( dir == FZ_IOST_WRITE ) *version = 1;

	err = fz_iost_one_long(iost,&my_palette->value1);
	if(err == FZRT_NOERR)
	{	err = fz_iost_one_long(iost,&my_palette->value2);
		if(err == FZRT_NOERR)
		{	err = fz_iost_one_long(iost,&my_palette->value3);
			if(err == FZRT_NOERR)
			{	err = fz_iost_one_long(iost,&my_palette->value4);

				if(*version >= 1)
				{	err = fz_iost_one_long(iost,
						&my_palette->value5);
				}
			}
		}
	}

	return(err);
}
```

2.8.6 Renderer

In **form•Z**, there are seven default rendering types: Wire Frame, Interactive Shaded, Quick Paint, Surface Render, Hidden Line, Shaded Render and RenderZone. Additional rendering types may be added by creating a renderer plugin and registering a callback function set, which provides functions called by **form•Z** to render the modeling scene on the screen. Each plugin renderer will automatically be added to the Display menu, below the standard **form•Z** renderers. In general, a plugin renderer will fall into one of three categories: vector line or polygonal drawing, such as Wire Frame, Hidden Line or Quick Paint, interactive rendering such as Interactive Shaded or static pixel based rendering, such as Shaded Render or RenderZone. Depending on the category, different screen drawing methods need to be used. Vector and polygonal renderers may draw directly to the screen. Interactive renderers usually employ some kind of hardware assisted display. Static pixel based renderers must use the **form•Z** supplied image buffer to store and display the image. A renderer should represent the modeling scene in a faithful manner. That is, the projections of the objects on the screen should match those of the other rendering types. **form•Z** offers utility functions, which facilitate the transformation of a 3d point through the display pipeline to the screen space. If a renderer provides a shaded display of the scene, surface colors and lighting should be taken into account. While it is not possible to execute the RenderZone shaders of a surface style in a plugin, the renderer is free to create its own artistic or realistic shading of surfaces based on the color of a surface style or based on its own surface style attribute. Lighting effects can be incorporated in a renderer anywhere from simple to accurate illumination. All parameters of the lights in a scene can be retrieved by a renderer through **form•Z** API functions. It is up to the renderer to use this information to create the illumination of the shaded surfaces. If a renderer is pixel based, the image is automatically exported to a 2d file format, when the Export Image command is chosen from the File menu. Vector and polygonal renderers should implement an export callback function, which writes out the rendered graphics to a 2d file.

The function set which defines a renderer is `fz_rndr_cbak_fset` and must be registered with a plugin of type `FZ_RNDR_EXTS_TYPE`. The example below shows the definition of a plugin of type `FZ_RNDR_EXTS_TYPE` and the registration of a single renderer within that plugin.

```
fzrt_error_td my_rndr_register_plugin ()
{
	fzrt_error_td	err = FZRT_NOERR;

	/* REGISTER THE RENDERER PLUGIN */
	err = fzpl_glue->fzpl_plugin_register(
		MY_RNDR_PLUGIN_UUID,
		MY_RNDR_PLUGIN_NAME,
		MY_RNDR_PLUGIN_VERSION,
		MY_RNDR_PLUGIN_VENDOR,
		MY_RNDR_PLUGIN_URL,
		FZ_RNDR_EXTS_TYPE,
		FZ_RNDR_EXTS_VERSION,
		NULL,
		0,
		NULL,
		&my_rndr_plugin_runtime_id);

	if ( err == FZRT_NOERR )
	{
		/* REGISTER THE RENDERER CALLBACK FUNCTION SET */
		err = fzpl_glue->fzpl_plugin_add_fset(
				my_rndr_plugin_runtime_id,
				FZ_RNDR_CBAK_FSET_TYPE,
				FZ_RNDR_CBAK_FSET_VERSION,
				FZ_RNDR_CBAK_FSET_NAME,
```

```
                            FZPL_TYPE_STRING(fz_rndr_cbak_fset),
                            sizeof (fz_rndr_cbak_fset),
                            my_fill_rndr_cbak_fset,
                            FALSE);
        }

        return(err);
}
```

The function set registration passes a function to `fzpl_plugin_add_fset`, which is executed by **form•Z** at startup. In the example above, the registration of the renderer passes the function `my_fill_rndr_cbak_fset`. This function must be defined by the plugin developer and must fill in the renderer function set with the pointers of the callback functions which constitute the functionality of a custom renderer. An example of this registration process is shown below. It assigns the callbacks of the sample renderer type to the function set. It is possible to register more than one renderer function set with a plugin. In this case the `fzpl_plugin_add_fset` call needs to be repeated for each function set, using the same plugin id, but a different callback function set fill function. Given the complexity of a renderer plugin, it is recommended that only one function set is registered with a renderer plugin.

```
fzrt_error_td my_rndr_callback_fset (
        const fzpl_fset_def_ptr fset_def,
        fzpl_fset_td * const fset )
{
    fzrt_error_td       err = FZRT_NOERR;
    fz_rndr_cbak_fset   *rndr_fset;

    err = fzpl_glue->fzpl_fset_def_check ( fset_def,
                    FZ_RNDR_EXTS_VERSION,
                    FZPL_TYPE_STRING(fz_rndr_cbak_fset),
                    sizeof (fz_rndr_cbak_fset),
                    FZPL_VERSION_OP_NEWER);

    if ( err == FZRT_NOERR )
    {
        rndr_fset = (fz_rndr_cbak_fset *)fset;

        /* RENDERER LEVEL */
        rndr_fset->fz_rndr_cbak_init            = my_rndr_init;
        rndr_fset->fz_rndr_cbak_info            = my_rndr_info;
        rndr_fset->fz_rndr_cbak_finit           = my_rndr_finit;
        rndr_fset->fz_rndr_cbak_name            = my_rndr_name;
        rndr_fset->fz_rndr_cbak_uuid            = my_rndr_uuid;
        rndr_fset->fz_rndr_cbak_attr            = my_rndr_attr;
        rndr_fset->fz_rndr_cbak_disp_attr       = my_rndr_disp_attr;
        rndr_fset->fz_rndr_cbak_handle_view     = my_rndr_view_type;
        rndr_fset->fz_rndr_cbak_activate        = my_rndr_activate;
        rndr_fset->fz_rndr_cbak_deactivate      = my_rndr_deactivate;

        /* PROJECT LEVEL */
        rndr_fset->fz_rndr_cbak_proj_data_init  = my_rndr_proj_init;
        rndr_fset->fz_rndr_cbak_proj_data_finit = my_rndr_proj_finit;
        rndr_fset->fz_rndr_cbak_proj_clear_mem  = my_rndr_clear_mem;

        /* WINDOW LEVEL */
        rndr_fset->fz_rndr_cbak_wind_data_init  = my_rndr_wind_init;
        rndr_fset->fz_rndr_cbak_wind_data_finit = my_rndr_wind_finit;
        rndr_fset->fz_rndr_cbak_wind_opts_init  = my_rndr_wind_opts_init;
        rndr_fset->fz_rndr_cbak_wind_opts_default = my_rndr_wind_opts_defaults;
        rndr_fset->fz_rndr_cbak_wind_opts_io    = my_rndr_wind_opts_io;
        rndr_fset->fz_rndr_cbak_wind_opts_copy  = my_rndr_wind_opts_copy;
        rndr_fset->fz_rndr_cbak_wind_opts_are_equal= my_rndr_wind_opts_cmp;
```

```
        rndr_fset->fz_rndr_cbak_wind_opts_finit = my_rndr_wind_opts_finit;

        /* IMAGE RELATED */
        rndr_fset->fz_rndr_cbak_image_init      = my_rndr_image_prep;
        rndr_fset->fz_rndr_cbak_image_disp      = my_rndr_image_disp;
        rndr_fset->fz_rndr_cbak_image_finit     = my_rndr_image_finit;
        rndr_fset->fz_rndr_cbak_image_inval     = my_rndr_image_inval;
        rndr_fset->fz_rndr_cbak_image_dirty     = my_rndr_image_dirty;

        /* VECTOR/POLYGON IMAGE EXPORT */
        rndr_fset->fz_rndr_cbak_expt_vect_out            = NULL;
        rndr_fset->fz_rndr_cbak_expt_vect_check_feature  = NULL;

        /* INTERFACE RELATED */
        rndr_fset->fz_rndr_cbak_iface_tmpl      = my_rndr_iface_tmpl;
        rndr_fset->fz_rndr_cbak_get_parm        = my_rndr_get_parm;
        rndr_fset->fz_rndr_cbak_set_parm        = my_rndr_set_parm;

        /* DISTRIBUTED RENDERING */
        rndr_fset->fz_rndr_cbak_proj_linked_files = my_rndr_linked_files;

        /* BACKGOUND IMAGE DONE NOTIFIACTION */
        rndr_fset->fz_rndr_cbak_notify_user     = my_rndr_notify_user;

    }

    return err;
}
```

Of all the callback functions of a renderer, only some are required, while others are recommended and others are purely optional. The callback functions of a renderer are grouped in a number of categories: Renderer level, project level, window level, image display, image export, interface and interactive.

Renderer level functions

The init function (optional)

```
fzrt_error_td  fz_rndr_cbak_init (
        void
        );
```

form•Z calls this function once at system startup. It allows a plugin renderer to perform one time initializations.

```
fzrt_error_td  my_rndr_init (
        void
        )
{
        fzrt_error_td rv = FZRT_NOERR;

        /* INIT CODE GOES HERE */
        ...

        return(rv);
}
```

The finit function (optional)

```
fzrt_error_td  fz_rndr_cbak_finit (
       void
       );
```

form•Z calls this function once, when the user quits **form•Z**. It allows a plugin renderer to perform final cleanup.

```
fzrt_error_td  my_rndr_finit (
       void
       )
{
       fzrt_error_td rv = FZRT_NOERR;

       /* FINIT CODE GOES HERE */
       ...

       return(rv);
}
```

The info function (required)

```
fzrt_error_td  fz_rndr_cbak_info (
       fz_rndr_type_enum   *type,
       fz_rndr_behave_enum *behave,
       long                *proj_data_size,
       long                *wind_data_size,
       long                *wind_opts_size
       );
```

This function is called by **form•Z** to retrieve basic information about the renderer. The first function argument sets the type of renderer. Three choices are available: pixel, vector or polygonal. A pixel renderer is expected to create a pixel based image, using techniques such as scanline z-buffering or raytracing. The Shaded Render and RenderZone display modes are examples of a pixel renderer. A vector renderer is expected to draw lines to the screen. The basic vector renderer provided by **form•Z** is Wire Frame. A vector renderer is also expected to provide a callback function which exports the lines to a 2d vector format. This callback function is described in more detail below. A polygonal renderer draws the faces of objects as closed and color filled polygons on the screen. The faces are usually sorted in the viewing direction, to provide proper depth display. It is also expected to provide the 2d export callback function. The Quick Paint and Surface Render display modes are examples of a polygonal renderer.

The behave argument tells **form•Z**, whether the render is fast enough to be executed as an interactive renderer or not. This will allow a user to create and edit objects and manipulate views in real time. Two return values are possible: `FZ_RNDR_BEHAVE_INTERACT` and `FZ_RNDR_BEHAVE_STATIC`. If the renderer is interactive, additional callback functions must be supplied, which is described in more detail further below.

Depending on the combination of type and behavior, different drawing methods need to be used. If a renderer is static and the type is pixel, it must store one horizontal scanline of the rendered image at a time in an image buffer, which is provided by **form•Z**. This is done with the API call `fz_rndr_ibuf_add_scanline`. If a renderer is interactive, it may use simple screen drawing commands, such as the **form•Z** API functions `fzrt_move_to` and `fzrt_line_to`, but must do so fast enough to be reasonably interactive. The **form•Z** Wire Frame drawing mode, for example, uses this technique. An interactive renderer may also use hardware assisted drawing, such as OpenGL. This is done in the Interactive Shaded display mode. Static vector and polygonal renderers may use any drawing method to put the image on the screen.

form•Z manages the storage of the options, window level and project level data for a plugin renderer. To allocate the proper amount of memory, the plugin renderer needs to tell **form•Z** how many bytes are needed for each data block. This is done with the last three function arguments. Typically, a renderer has the options stored in a structure, whose size can be inquired with a `sizeof(structure_type)` call. If 0 is returned for any of the sizes, no memory will be allocated for the respective data block. The options data holds the parameters for a renderer that can be set by a user. They are also displayed in the corresponding options dialog, which can be accessed through the Display menu. The project data is information that may be needed by a renderer on a per project level. For example, the renderer may need to keep a copy of the geometry to be rendered. This information would be stored in the project data block. Likewise, the renderer may need to keep information on a per window basis. For example, it may be necessary to keep track of whether the rendering in a window needs to be regenerated since last rendered, or whether the previously rendered image can be displayed from a buffer. This information can be stored in the window data block.

```
fzrt_error_td  my_rndr_info (
       fz_rndr_type_enum   *type,
       fz_rndr_behave_enum *behave,
       long                *proj_data_size,
       long                *wind_data_size,
       long                *wind_opts_size
       )
{
       fzrt_error_td rv = FZRT_NOERR;

       *type = FZ_RNDR_TYPE_PIXEL;
       *behave = FZ_RNDR_BEHAVE_STATIC;

       *proj_data_size = sizeof(my_rndr_proj_data_td);
       *wind_data_size = sizeof(my_rndr_wind_data_td);
       *wind_opts_size = sizeof(my_rndr_wind_opts_td);

       return(rv);
}
```

The name function (required)

```
fzrt_error_td  fz_rndr_cbak_name)(
       char   *name,
       long   max_len       );
```

This function is called by **form•Z** to get the name of the renderer. This name shows up in the **form•Z** interface, whenever the title of the renderer is shown. The name function must assign a string to the function's name argument. The length of the string assigned cannot exceed `max_len` characters. It is recommended that the renderer's name is stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example below, this step is omitted for the purpose of simplicity.

```
fzrt_error_td my_rndr_name (
       char          *name,
       long          max_name
       )
{
       strncpy(name,"My renderer",max_name);
       return(FZRT_NOERR);
}
```

The uuid function (required)

```
fzrt_error_td fz_rndr_cbak_uuid (
	fzrt_UUID_td			uuid
	);
```

This function is called by **form•Z** to get the uuid of the renderer. This unique id is used by **form•Z** to distinguish the renderer from other renderers. The uuid function needs to assign the unique identifier to the function's uuid argument. An example is shown below.

```
#define MY_RNDR_UUID       \
"\xec\x4c\x18\x73\xc4\x48\x48\x6a\x99\xa5\x35\xc8\xf9\xc4\x35\xd7"

fzrt_error_td my_rndr_uuid (
	fzrt_UUID_td  uuid
	)
{
	fzrt_UUID_copy(MY_RNDR_UUID, uuid);
	return(FZRT_NOERR);
}
```

The options uuid function (required, if the renderer has user options)

```
fzrt_error_td fz_rndr_cbak_opts_uuid (
	fzrt_UUID_td			uuid
	);
```

This function is called by **form•Z** to get the options uuid of the renderer. This unique id is used by **form•Z** to distinguish the renderer's options dialog from that of other renderers. The uuid function needs to assign the unique identifier to the function's uuid argument. An example is shown below.

```
#define MY_RNDR_OPTS_UUID  \
"\xc3\x63\xd8\xf5\x81\xd2\x4a\x47\x8a\xc2\xd4\xe0\xf8\x63\x56\x70"

fzrt_error_td my_rndr_opts_uuid (
	fzrt_UUID_td  uuid
	)
{
	fzrt_UUID_copy(MY_RNDR_OPTS_UUID, uuid);
	return(FZRT_NOERR);
}
```

The options name function (required, if the renderer has user options)

```
fzrt_error_td fz_rndr_cbak_opts_name (
	char   *name,
	long   max_len
	);
```

This function is called by **form•Z** to get the title of the options dialog for a renderer. The options name function must assign a string to the function's name argument. The length of the string assigned cannot exceed `max_len` characters. It is recommended that the options name is stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example below, this step is omitted for the purpose of simplicity.

```
fzrt_error_td my_rndr_opts_name (
```

```
        char           *name,
        long           max_name
        )
{
        strncpy(name,"My Renderer Options",max_name);
        return(FZRT_NOERR);
}
```

The attribute function (recommended)

```
fzrt_boolean  fz_rndr_cbak_attr (
        long                windex,
        fz_rndr_attr_enum   rndr_attr
        );
```

The attribute function is intended to tell **form•Z** more detailed information about the renderer. The function receives a question in the form of an enum, and the function needs to return TRUE or FALSE as an answer to that question. Based on the answer, **form•Z** will take the appropriate action with that renderer in various settings. For example, **form•Z** may call this function with the FZ_RNDR_ATTR_IN_PVIEW enum. If the function returns TRUE, the rendering mode will be offered in dialogs, which offer object preview renderings, such as the Sweep Edit dialog. If the answer is FALSE, the rendering in not offered.

The enums that **form•Z** could pass in the attribute function are:

FZ_RNDR_ATTR_RADIOS

If the renderer is capable to display the illumination of a radiosity solution, it should return TRUE and FALSE otherwise.

FZ_RNDR_ATTR_RADIOS_PVIEW

If the renderer is capable to display the illumination of a radiosity solution and it is fast enough to do this while generating the solution, it should return TRUE and FALSE otherwise.

FZ_RNDR_ATTR_ALPHA_CHANNEL

If the renderer is a static pixel renderer and it supports the alpha channel and the current options have this setting turned on, the function should return TRUE and FALSE otherwise.

FZ_RNDR_ATTR_SUN_ONLY

If the renderer perfroms only simple illumination and only uses one light, the sun light, it should return TRUE and FALSE otherwise.

FZ_RNDR_ATTR_CAN_SS

If the renderer's image can be supersampled, either by a built in method of by drawing the image larger, it should return TRUE and FALSE otherwise

FZ_RNDR_ATTR_IN_PVIEW

If the renderer should be offered in a dialog with an object preview window, such as the Sweep Edit dialog, the function should return TRUE.

FZ_RNDR_ATTR_STAY_IN_IACT_MODE

If the function returns TRUE to this question, the rendering mode is switched from the current rendering mode to an interactive rendering mode, after performing an object creation or editing operation or a view manipulation. TRUE should only be returned if the renderer itself is non interactive. For example, when creating a new object and the current rendering mode is RenderZone, **form•Z** will temporarily switch to Wire Frame of Interactive Shaded while the object is rubberbanded. After that, the rendering mode returns to RenderZone. This is because, the RenderZone attribute function return FALSE to the FZ_RNDR_ATTR_STAY_IN_IACT_MODE question. If it would return TRUE, the objects would still be displayed in Wire Frame or Interactive Shaded after the creation is finished.

FZ_RNDR_ATTR_USE_PARTIAL_IMAGE

If the renderer returns TRUE to this question, the standard set image size option will automatically be added at the bottom of the renderers options dialog. TRUE should only be returned for static pixel renderers.

FZ_RNDR_ATTR_USE_SAVE_IMAGE

If the renderer returns TRUE to this question, the standard save image option will automatically be added at the bottom of the renderers options dialog. TRUE should only be returned for non interactive renderers, which provide higher quality images, which may take a longer time to generate.

FZ_RNDR_ATTR_USE_SHAD_PROJ_OPTS

If the renderer returns TRUE to this question, the standard Smooth Shading options tab in the Project Rendering Options dialog will be offered. The renderer is expected to use these settings when creating smooth shaded images in a pixel based rendering. The smooth shading settings of a project can be acquired with the API call `fz_proj_rndr_opts_sshd_get`.

FZ_RNDR_ATTR_USE_GEOM_PROJ_OPTS

If the renderer returns TRUE to this question, the standard Geometry options tab in the Project Rendering Options dialog will be offered. The renderer is expected to use these settings when rendering parametric and smooth objects using the high level surfaces of the smooth and parametric objets. FALSE should be returned, if the facets of smooth objects are always used to render the scene. The geometry settings of a project can be acquired with the API call `fz_proj_rndr_opts_geom_get`.

FZ_RNDR_ATTR_USE_TCTL_PROJ_OPTS

If the renderer returns TRUE to this question, the standard Texture Map Control options tab in the Project Rendering Options dialog will be offered. The renderer is expected to use these settings when rendering textures. The renderer should use the global texture map copntrol settings for objects which do not have a texture map control attribute. FALSE should be returned, if the renderer does not deal with textures. The texture map control settings of a project can be acquired with the API call `fz_proj_rndr_opts_tctl_get`.

FZ_RNDR_ATTR_HANDLE_PIXBUFF

Under certain circumstances, a rendering is not intended to be displayed on the screen but needs to be stored in a pixel buffer. **Form•Z** will automatically handle this if the renderer is a static pixel renderer or uses the fzrt drawing commands. In this case, FALSE should be answered to this question. However, if the renderer does not use standard drawing command, but uses hardware assistance, it must fill in the pixel buffer and must answer TRUE to this question. Pixel buffer related API functions can be found in the fzrt (**form•Z** runtime) function set.

FZ_RNDR_ATTR_HANDLE_PANORAMIC

This should only be answered TRUE if panoramic views are supported (the fz_rndr_cbak_handle_view callback must return TRUE for FZ_VIEW_TYPE_PANORAMIC) and the renderer is a vector or polygon renderer. In this case, drawing of panoramic images is solely the responsibility of the renderer. If panoramic views are supported, but FALSE is answererd to this question, **form•Z** will handle the drawing of the panorama for the renderer. This is also automatically the case, if the renderer is a static pixel renderer. This is done by asking the renderer to create a set of narrow perspectives, which are turned 90 degrees. The width of these strips is the Smoothness parameter of the panoramic view type.

FZ_RNDR_ATTR_IS_BACKGROUND

This should be answered TRUE, if the renderer calculates the image in a separate background process. See the section titled "Background renderers" at the end of this chapter for more details.

FZ_RNDR_ATTR_USES_TMAPS

This should be answered TRUE, if the renderer is using the texture maps of a surface style. For example, if a surface style has the Color Image Map shader and the renderer uses this texture map when rendering a surface, TRUE should be returned. Note, that not all possible texture maps of a surface style need to be supported (such as bump, ambient, diffuse etc). However, at least color and transparency maps should when answering TRUE.

```
fzrt_boolean  my_rndr_attr (
       long                windex,
       fz_rndr_attr_enum   rndr_attr
       )
{
       fzrt_boolean  rv = FALSE;

       switch (rndr_attr)
       {
              case FZ_RNDR_ATTR_STAY_IN_IACT_MODE      :
              case FZ_RNDR_ATTR_USE_PARTIAL_IMAGE      :
              case FZ_RNDR_ATTR_USE_SAVE_IMAGE         :
              case FZ_RNDR_ATTR_USE_TCTL_PROJ_OPTS     :
                     rv = TRUE;
              break;
       }

       return(rv);
}
```

The display attribute function (optional)

```
fzrt_boolean  fz_rndr_cbak_disp_attr (
        long                        windex,
        fz_rndr_disp_attr_enum      rndr_disp_attr
        );
```

The display attribute function is similar to the attribute callback function.
It is intended to tell **form•Z** which standard graphics are drawn by the renderer. The function receives a question in form of an enum, and the function needs to return TRUE or FALSE as an answer to that question. Based on the answer, **form•Z** will enable or disable the respective item in the Windows menu. It is still the responsibility of the renderer to draw the actual graphics, such as the grid. **form•Z** provides API functions to perform this task. If this function is not implemented, the answer is assumed to be FALSE.

FZ_RNDR_DISP_ATTR_GRID

If the renderer chooses to display the grid in the background, it should return TRUE.

FZ_RNDR_DISP_ATTR_AXIS

If the renderer chooses to display the world and reference plane axes in the background, it should return TRUE.

FZ_RNDR_DISP_ATTR_ULAY

If the renderer chooses to display the underlay image, if it exists, in the background, it should return TRUE.

FZ_RNDR_DISP_ATTR_LITE

If the renderer chooses to display the Wire Frame graphics of lights it should return TRUE.

FZ_RNDR_DISP_ATTR_MARQUEE

If the renderer chooses to display the axis markers it should return TRUE.

```
fzrt_boolean  my_rndr_disp_attr (
        long                        windex,
        fz_rndr_disp_attr_enum      rndr_disp_attr
        )

        fzrt_boolean  rv = FALSE;

        switch (rndr_disp_attr)
        {
                case FZ_RNDR_DISP_ATTR_GRID       :
                case FZ_RNDR_DISP_ATTR_AXIS       :
                        rv = TRUE;
                break;
        }

        return(rv);
}
```

The handle view function (required)

```
fzrt_boolean fz_rndr_cbak_handle_view (
       long                windex,
       fz_view_ptr         view
       );
```

This function needs to tell **form•Z**, whether the renderer can handle a particular view type. The window for the current rendering and the view are passed in. The function should use the view API functions to retrieve the necessary parameters of the view to determine, whether is can faithfully display the geometry in the scene with the view's settings. For example, a renderer may be able to display in a standard perspective, but when the Keep Vertical Lines Straight option is selected, the renderer may not be able to handle the view. In this case the function should return TRUE, if the option is off and FALSE if it is on. Other, more exotic view types such as panoramic, may not be handled at all, in which case FALSE will be returned for all views of that type. The sample function below shows how to properly tell **form•Z** that straight up perspectives and panoramic views are not handled.

```
fzrt_boolean my_rndr_handle_view (
   long          windex,
   fz_view_ptr   view
   )
{
   fz_type_td              data;
   fz_view_type_enum   view_type;
   fzrt_boolean        persp_straight, rv = TRUE;

   fz_view_get_parm_data(windex, view, FZ_VIEW_PARM_TYPE,&data);
   fz_type_get_enum(&data,&view_type);
   if (view_type == FZ_VIEW_TYPE_PANORAMIC )
   {
      rv = FALSE;
   }
   else if (view_type == FZ_VIEW_TYPE_PERSPECTIVE)
   {
      fz_view_get_parm_data(windex,view,
         FZ_VIEW_PARM_PERSPECTIVE_STRAIGHT,&data);
      fz_type_get_boolean(&data,&persp_straight);
      if (persp_straight == TRUE) rv = FALSE;
   }
   return(rv);
}
```

The activate function (optional)

```
fzrt_error_td fz_rndr_cbak_activate (
       long          windex
       );
```

The activate function is called by **form•Z**, when the rendering mode is switched from another mode to the rendering mode defined by this plugin. It gives the renderer the opportunity to perform operations which need to be executed once, when a renderer is selected and before the rendering is executed.

```
fzrt_error_td my_rndr_activate (
       long          windex
       )
{
       fzrt_error_td rv = FZRT_NOERR;
```

```
        /* ACTIVATION CODE GOES HERE */
        ...

        return(rv);
}
```

The deactivate function (optional)

```
fzrt_error_td fz_rndr_cbak_deactivate (
        long            windex
        );
```

The deactivate function is called by **form•Z**, when the rendering mode is switched from the rendering mode defined by this plugin to another mode. It gives the renderer the opportunity to perform operations which need to be executed once, when a renderer is deselected.

```
fzrt_error_td my_rndr_deactivate (
        long            windex
        )
{
        fzrt_error_td rv = FZRT_NOERR;

        /* DEACTIVATION CODE GOES HERE */
        ...

        return(rv);
}
```

Project level functions

The project data init function (required, if project data exists)

```
fzrt_error_td  fz_rndr_cbak_proj_data_init (
        long            windex,
        fzrt_ptr        proj_data
        );
```

The project data init function needs to be implemented, if the info function returns a size other than 0 for the project data size argument (see above). It is called once, when a new project is created. **form•Z** will allocate a data block of the given size. The project data init function is then called with the pointer to the data and is expected to initialize the data. The project data block is intended to store any runtime data a renderer may need on a per project basis. For example, a renderer may need to create a copy of the geometry to be rendered. This would be stored in the project data.

```
fzrt_error_td  my_rndr_proj_data_init(
        long            windex,
        fzrt_ptr        proj_data
        )
{
        my_proj_data_td     *my_proj_data;

        my_proj_data = (my_proj_data_td*) proj_data;

        /* PROJECT INIT CODE GOES HERE. FOR EXAMPLE */
        /* A HYPOTHETICAL POLYGON ARRAY */
        my_proj_data->num_polys = 0;
        my_proj_data->polys = NULL;
        ...
```

```
        return (FZRT_NOERR);
}
```

The project data finit function (recommended, if project data exists)

```
fzrt_error_td  fz_rndr_cbak_proj_data_finit (
        long           windex,
        fzrt_ptr       proj_data
        );
```

The project data finit function is complementary to the project data init function. If implemented, it is called when a project is closed. It gives the renderer the opportunity to finit any dynamic data that was created since the project was opened. If the info function returned 0 for the project data size, this function does not need to be implemented.

```
fzrt_error_td  my_rndr_proj_data_finit (
        long           windex,
        fzrt_ptr       proj_data
        )
{
        my_proj_data_td     *my_proj_data;

        my_proj_data = (my_proj_data_td*) proj_data;

        /* PROJECT FINIT CODE GOES HERE. FOR EXAMPLE */
        /* A HYPOTHETICAL POLYGON ARRAY */
        if (my_proj_data->polys != NULL)
        {
               ...
        }

        return (FZRT_NOERR);
}
```

The clear memory function (required, if memory is allocated)

```
fzrt_error_td  fz_rndr_cbak_proj_clear_mem (
        long           windex
        );
```

This function is called, when the user selects the Clear Rendering Memory item in the Display menu. It is expected to reset all memory that was allocated for the renderer since the creation of the project. As renderers tend to allocate and store large amounts of data, the Clear Rendering Memory command is intended to give memory back to the user for modeling operations. The clear memory function should leave the dynamic data of a renderer in the same state, as when a project is first created.

```
fzrt_error_td  my_rndr_proj_clear_mem (
        long           windex
        )
{
        my_proj_data_td     *my_proj_data;


        fz_rndr_proj_data_get(windex,my_rndr_indx,(fzrt_ptr*)&my_proj_data);

        /* CLEAR MEMORY CODE GOES HERE. FOR EXAMPLE */
```

```
        /* A HYPOTHETICAL POLYGON ARRAY */
        if (my_proj_data->polys != NULL)
        {
        .../* deallocate */
        }
        my_proj_data->polys        = NULL;
        my_proj_data->num_polys    = 0;
        ...

        return (FZRT_NOERR);
}
```

Window level functions

The window data init function (required, if window data exists)

```
fzrt_error_td  fz_rndr_cbak_wind_data_init (
        long          windex,
        fzrt_ptr      wind_data
        );
```

The window data init function needs to be implemented, if the info function returns a size other than 0 for the window data size argument (see above). It is called once, when a new window is created. **form•Z** will allocate a data block of the given size. The window data init function is then called with the pointer to the data and is expected to initialize the data. The window data block is intended to store any runtime data a renderer may need on a per window basis. For example, a renderer may need to keep track whether the last image rendered is still valid or whether any changes made by the user in the meantime have made the image out of date. Such a marker would be stored in the window data.

```
fzrt_error_td  my_rndr_wind_data_init(
        long          windex,
        fzrt_ptr      wind_data
        )
{
        my_wind_data_td     *my_wind_data;

        my_wind_data = (my_wind_data_td *) wind_data;

        /* SET THE IMAGE DIRTY MARKER TO TRUE, SINCE */
        /* NO IMAGE HAD BEED RENDERED YET */
        my_wind_data->image_dirty = TRUE;

        ...

        return (FZRT_NOERR);
}
```

The window data finit function (recommended, if window data exists)

```
fzrt_error_td  fz_rndr_cbak_wind_data_finit (
        long          windex,
        fzrt_ptr      wind_data
        );
```

The window data finit function is complementary to the window data init function. If implemented, it is called when a window is closed. It gives the renderer the opportunity to finit any dynamic data that was created since the window was opened. If the info function returned 0 for the window data size, this function does not need to be implemented.

```
fzrt_error_td  my_rndr_wind_data_finit (
       long          windex,
       fzrt_ptr      wind_data
       )
{
       my_wind_data_td     *my_wind_data;

       my_wind_data = (my_wind_data_td*) wind_data;

       /* WINDOW FINIT CODE GOES HERE. */
       ...

       return (FZRT_NOERR);
}
```

The window options init function (required, if window options exist)

```
fzrt_error_td  fz_rndr_cbak_wind_opts_init (
       long          windex,
       fzrt_ptr      wind_opts
       );
```

The window options init function needs to be implemented, if the info function returns a size other than 0 for the window options size argument. It is called once, when a new window is created. **form•Z** will allocate a data block of the given size. The window options init function is then called with the pointer to the data and is expected to initialize the data. The window options block is intended to store the options a renderer will expose to the user. For example, a vector renderer may offer a line width parameter.

```
fzrt_error_td  my_rndr_cbak_wind_init (
       long          windex,
       fzrt_ptr      wind_opts
       )
{
       my_wind_opts_td     *my_wind_opts;

       my_wind_opts = (my_wind_opts_td *) wind_opts;

       /* WINDOW OPTIONS INIT CODE GOES HERE. FOR EXAMPLE */
       /* A LINE THICKNESS PARAMETER */
       my_wind_opts->line_width = 1;

       ...


       return(FZRT_NOERR);
}
```

The window options default function (required, if window options exist)

```
fzrt_error_td  fz_rndr_cbak_wind_opts_default (
       long          windex,
       fzrt_ptr      wind_opts
       );
```

The window options default function needs to be implemented, if the info function returns a size other than 0 for the window options size argument. It is invoked, whenever default values need to be set for the window options of a renderer. The window options defaults function is called with the pointer to the window options data and is expected to set the data to default values. This function may, in essence, be the same as the window options init function. The difference is that the init function is called only once, when the window is created, whereas the defaults function may be called multiple times.

The window options io function (required, if window options exist)

```
fzrt_error_td fz_rndr_cbak_wind_opts_io(
       long                        windex,
       fz_iost_ptr                 iost,
       fz_iost_dir_td_enum         dir,
       fzpl_vers_td * const        version,
       fzrt_ptr                    data
);
```

The window options io function needs to be implemented, if the info function returns a size other than 0 for the window options size argument. It is called, whenever the options are written to or read from a file. It is expected from the plugin to keep track of version changes. When writing, the function needs to return a version number back to **form•Z**. When reading, the version of the window options data when written will be passed into this function by **form•Z**. When the options are changed by a plugin, the version number should be increased. Thus, when reading older versions, they can be handled accordingly.

```
fzrt_error_td my_rndr_wind_opts_io(
       long                        windex,
       fz_iost_ptr                 iost,
       fz_iost_dir_td_enum         dir,
       fzpl_vers_td * const        version,
       fzrt_ptr                    data
       )
{
       my_wind_opts_td     *my_wind_opts;
       fzrt_error_td       rv = FZRT_NOERR;

       my_wind_opts = (my_wind_opts_td *) wind_opts;

       if ( dir == FZ_IOST_WRITE ) *version = 1;

       rv = fz_iost_long(iost,&wind_opts->line_width,1);
       ...

       return(rv);
}
```

The window options copy function (required, if dynamic data is allocated in the window options)

```
fzrt_error_td  fz_rndr_cbak_wind_opts_copy(
       long          src_windex,
       fzrt_ptr      src_opts_data,
       long          dst_windex,
       fzrt_ptr      dst_opts_data
       );
```

The window options copy function allows a renderer to copy any dynamic data that may be contained in the window options. If this function is not implemented and the window options are copied, **form•Z** performs a byte by byte copy of the data. This will work well, as long as there are no dynamically allocated arrays in the window options. If this is the case, the copy function must be implemented and must copy the arrays from the source to the destination storage.

```
fzrt_error_td my_rndr_wind_opts_copy(
	long		src_windex,
	fzrt_ptr	src_opts_data,
	long		dst_windex,
	fzrt_ptr	dst_opts_data
	)
{
	my_wind_opts_td		*src_wind_opts,*dst_wind_opts;
	fzrt_error_td		rv = FZRT_NOERR;

	src_wind_opts = (my_wind_opts_td*) src_opts_data;
	dst_wind_opts = (my_wind_opts_td*) dst_opts_data;

	if (src_wind_opts->array != NULL )
	{
		if((dst_wind_opts->array = (long*)fzrt_mem_zone_alloc(
					plugin_zone_ptr,
					sizeof(long) * src_wind_opts->n_array,
					FALSE)) != NULL )
		{

			fzrt_block_move(src_wind_opts->array,
					dst_wind_opts->array,
					sizeof(long) * src_wind_opts->n_array);
		}
		else
		{
			err = fzrt_error_set (
				FZ_MALLOC_ERROR,
				FZRT_ERROR_SEVERITY_ERROR,
				FZRT_ERROR_CONTEXT_APP, 0 );
		}
	}
	else
	{
		dst_wind_opts->array = NULL;
	}
	dst_wind_opts->n_array = src_wind_opts->n_array;

	/* COPY REMAINING FIELDS */
	...

	return(err);

}
```

The window options compare function (required, if dynamic data is allocated in the window options)

```
fzrt_boolean  fz_rndr_cbak_wind_opts_are_equal(
	fzrt_ptr	wind_opts_data1,
	fzrt_ptr	wind_opts_data2
	);
```

The window options compare function needs to be implemented if the window options contain dynamically allocated data. It is expected to tell **form•Z**, whether two sets of options are the same. If this function is not implemented, **form•Z** performs a byte by byte comparison of the options.

```
fzrt_boolean  my_rndr_wind_opts_are_equal(
```

```
	fzrt_ptr	wind_opts_data1,
	fzrt_ptr	wind_opts_data2
	)
{
	my_rndr_wind_opts_td	*my_wind_opts1,*my_wind_opts2;
	fzrt_boolean		are_equal;

	are_equal = TRUE;

	my_wind_opts1 = (my_rndr_wind_opts_td*) wind_opts_data1;
	my_wind_opts2 = (my_rndr_wind_opts_td*) wind_opts_data2;

	/* COMPARE ARRAY SIZE */
	if (my_wind_opts1->n_array == my_wind_opts2->n_array )
	{
		/* COMPARE ARRAY CONTENT */
		for(i = 0; i < my_wind_opts1->n_array; i++)
		{
			if (my_wind_opts1[i] != my_wind_opts2->array[i] ) break;
		}

		if ( i < my_wind_opts1->n_array)
		{
			are_equal = FALSE;
		}
		else
		{	/* COMPARE REMAINING FIELDS */
			...
		}
	}
	else
	{	are_equal = FALSE;
	}

	return(are_equal);
}
```

The window options finit function (required, if dynamic data is allocated in the window options)

```
fzrt_error_td  fz_rndr_cbak_wind_opts_finit (
	long		windex,
	fzrt_ptr	wind_opts
	);
```

The window options finit function needs to be implemented, if the info function returns a size other than 0 for the window options size argument (see above) and if dynamic data exists in the window options. It is called once, when a window is closed. The window options finit function is then called with the pointer to the data and is expected to free any dynamic data.

```
fzrt_error_td  my_rndr_cbak_wind_finit (
	long		windex,
	fzrt_ptr	wind_opts
	)
{
	my_wind_opts_td		*my_wind_opts;

	my_wind_opts = (my_wind_opts_td *) wind_opts;

	/* WINDOW OPTIONS FINIT CODE GOES HERE. FOR EXAMPLE */
	/* A DYNAMIC ARRAY */
	if ( my_wind_opts->array != NULL )
	{
```

```
		fzrt_mem_zone_free( plugin_zone_ptr, &my_wind_opts->array);
	}

	...

	return(FZRT_NOERR);
}
```

Image related functions

The image init function (optional)

```
fzrt_error_td  fz_rndr_cbak_image_init (
	long          windex
	);
```

This function is called by **form•Z** right before an image is about to be rendered. It gives the plugin the opportunity to perform the setup of image related data. For example, if the renderer needs to make a copy of the geometry to be rendered, it should be done in the image init function. The image init function is also allowed to bring up a progress bar to, for example, inform the user of the progress of creating the rendering data. The actual image display function (see below) which is called right after the image init function, is not allowd to show the progress bar or any dialogs, which obscure the image, while it is generated. If any errors occurred during the image init phase, the function should return an error code to **form•Z**.

```
fzrt_error_td  my_rndr_image_init (
	long          windex
	)
{
	my_wind_opts_td     *my_wind_opts;
	fzrt_error_td       err = FZRT_NOERR;

	/* GET THE RENDERER'S WINDOW OPTIONS */
	fz_rndr_wind_opts_get(windex,my_rndr_indx,(fzrt_ptr*)&my_wind_opts);

	/* IMAGE SETUP CODE GOES HERE */
	...

	return(err);
}
```

The image display function (required)

```
fzrt_error_td fz_rndr_cbak_image_disp (
	long                      windex,
	fzrt_error_td             prep_err,
	fzrt_rect                 *sub_image
	);
```

The image display function is the main function of a renderer. It is called by **form•Z** anytime the rendering on the screen needs to be refreshed. It is always called after the image init function (see above) and before the image finit function (see below). The sub_image argument is only passed as non NULL for static pixel renderers. If passed as NULL, the renderer is expexted to render the entire image. If it is non NULL, the sub_image rectangle outlines a rectangular portion of the image to be rendered. It is the resposibility of the renderer to not pass more than (sub_image->bottom - sub_image->top) number of scanlines to fz_rndr_ibuf_add_scanline and to make sure that each scanline is exactly (sub_image->right - sub_image->left) pixels wide. The sub_image rectangle is, for example, passed in if the Set Image Size option is checked by the user, or if the renderer is used by the network rendering environment and is asked to render one or more bands of an image.

If the image init function generated an error, it is passed into the display function. This gives the display function the opportunity to perform any necessary cleanup because of the error. The display function is expected to NOT render the image, if an error is passed in.

```
fzrt_error_td my_rndr_image_disp (
	long			windex,
	fzrt_error_td		prep_err,
	fzrt_rect		*sub_image
	)
{
	my_wind_data_td	*my_wind_data;
	fzrt_error_td err = FZRT_NOERR;

	if ( prep_err == FZRT_NOERR )
	{
		/* RENDER THE IMAGE */
		...
	}
	else
	{
		/* PERFORM ANY CLEANUP DUTIES HERE */
		...
	}

	return(err);
}
```

Note that the image display function is not allowed to post any dialogs, error message or progress bars. This should all be done in the image init or in the image finit function.

The image finit function (optional)

```
fzrt_error_td  fz_rndr_cbak_image_finit (
	long		windex
	);
```

This function is called by **form•Z** right after the image display function. If implemented, it should perform cleanup duties, which need to be done after the image display has been completed.

```
fzrt_error_td  my_rndr_image_finit (
	long		windex
	)
{
	fzrt_error_td		err = FZRT_NOERR;


	/* IMAGE CLEANUP CODE GOES HERE */
	...

	return(err);
}
```

The image inval function (optional)

```
fzrt_error_td  fz_rndr_cbak_image_inval(
       long                windex,
       fzrt_rect           *rect,
       fzrt_rgn_ptr        rgn
       );
```

The inval function is called when an area on the screen becomes invalid (i.e. needs to be redrawn because the graphics in that area are no longer uptodate). A renderer may need to know when this happens, in which case this function should be implemented. The invalidated area could either be a rectangle or an arbitrary shape. If the `rect` argument is not NULL, the area is a rectangle. Otherwise the `rgn` argument will be non NULL and the area is an arbitrary region.

```
fzrt_error_td  fz_rndr_cbak_image_inval(
       long                windex,
       fzrt_rect           *rect,
       fzrt_rgn_ptr        rgn
       )
{
       fzrt_error_td       rv = FZRT_NOERR;

       /* HANDLE THE INVAL HERE */
       ...

       return(rv);
}
```

The image dirty function (recommended)

```
fzrt_boolean  fz_rndr_cbak_image_dirty(
       long                windex
       );
```

form•Z calls this function to find out, whether any changes made by the user since the image was rendered last have made the image invalid. This function is especially useful, when the renderer is a static pixel renderer. As long as the image is valid and the screen needs to be redrawn **form•Z** will automatically draw the image buffer, instead of asking the renderer to re-render the scene. In order to accomplish this, the renderer needs to tell **form•Z** that nothing has occurred in the meantime that would invalidate the image. The recommended mechanism for this is for the renderer to install a notification function set. In this function set, the three callback functions `fz_notf_cbak_proj`, `fz_notf_cbak_wind` and `fz_notf_cbak_objt` should be defined. The `fz_notf_cbak_wind` function, for example, is called each time an aspect of the window in which a rendering takes place changes. What kind of change occurred is identified by an enum argument to the function. The renderer's window notification callback should then set a "dirty" marker in its window data block. When the `fz_rndr_cbak_image_dirty` function is invoked by **form•Z**, the renderer then passes back the value of the dirty marker. An example of a notification and image dirty mechanism is shown below.

In the plugin's registration function, a notification function set is registered:

```
       err = fzpl_glue->fzpl_plugin_add_fset(
              my_rndr_plugin_runtime_id,
              FZ_NOTF_CBAK_FSET_TYPE,
              FZ_NOTF_CBAK_FSET_VERSION,
              FZ_NOTF_CBAK_FSET_NAME,
              FZPL_TYPE_STRING(fz_notf_cbak_fset),
              sizeof(fz_notf_cbak_fset),
              my_rndr_fill_notf_fset,
              FALSE);
```

The fill function assigns the three relevant notification callbacks (more may need to be assigned, depending on the complexity of the renderer).

```
fzrt_error_td my_rndr_fill_notf_fset(
	const fzpl_fset_def_ptr fset_def,
	fzpl_fset_td* const fset)
{
	fzrt_error_td		err = FZRT_NOERR;
	fz_notf_cbak_fset* notf_fset;

	err = fzpl_glue->fzpl_fset_def_check(
		fset_def,
		FZ_NOTF_CBAK_FSET_VERSION,

		FZPL_TYPE_STRING(fz_notf_cbak_fset),
		sizeof(fz_notf_cbak_fset),
		FZPL_VERSION_OP_NEWER );

	if (err == FZRT_NOERR)
	{
		notf_fset = (fz_notf_cbak_fset*)fset;

		notf_fset->fz_notf_cbak_proj = my_rndr_notf_proj;
		notf_fset->fz_notf_cbak_wind = my_rndr_notf_wind;
		notf_fset->fz_notf_cbak_objt = my_rndr_notf_objt;
	}

	return err;
}
```

The three notification callbacks are implemented as follows:

```
fzrt_error_td my_rndr_notf_wind(
	long			windex,
	fz_notf_wind_enum	wind_notf,
	fz_notf_proj_enum	proj_notf
	)
{
	my_rndr_wind_data_td_ptr wind_data;

	fz_rndr_wind_data_get(windex, mt_rndr_index,(fzrt_ptr*)&wind_data);

	switch (wind_notf)
	{
		// THESE ACTIONS MAY THE IMAGE DIRTY
		case FZ_NOTF_WIND_DIRTY:
		case FZ_NOTF_WIND_RESIZE:
		case FZ_NOTF_WIND_ISPEC:
		case FZ_NOTF_WIND_VIEW:
		case FZ_NOTF_WIND_OPTS:
			wind_data->dirty = TRUE;
		break;

		// WINDOW IS DIRTY BECAUSE THE PROJECT BECAME DIRTY
		case FZ_NOTF_WIND_PROJ:
			my_rndr_notf_proj(windex,proj_notf);
		break;

	}

	return FZRT_NOERR;
}
```

```
fzrt_error_td my_rndr_notf_proj(
	long				windex,
	fz_notf_proj_enum	proj_notf
	)
{
	my_rndr_wind_data_td_ptr wind_data;

	fz_rndr_wind_data_get(windex,my_rndr_index,(fzrt_ptr*)&wind_data);

	switch(proj_notf)
	{
		case FZ_NOTF_PROJ_DIRTY:
		case FZ_NOTF_PROJ_LIGHTS:
		case FZ_NOTF_PROJ_CAMERA:
		case FZ_NOTF_PROJ_COLORS:
		case FZ_NOTF_PROJ_SURF:
		default:
			wind_data->dirty = TRUE;
		break;
	}

	return FZRT_NOERR;
}

fzrt_error_td my_rndr_notf_objt(
	long				windex,
	fz_notf_objt_enum	objt_notf,
	fz_objt_ptr			objt
	)
{
	my_rndr_wind_data_td_ptr wind_data;

	fz_rndr_wind_data_get(windex,my_rndr_index,(fzrt_ptr*)&wind_data);

	if (objt_notf != FZ_NOTF_OBJT_NONE)
	{
		wind_data->dirty = TRUE;
	}

	return FZRT_NOERR;
}
```

And finally, the image dirty callback function simply returns the dirty marker. Note that the image display function, after sucessfully rendering the scene, needs to set the dirty marker to FALSE.

```
fzrt_boolean my_rndr_image_dirty(long windex)
{
	my_rndr_wind_data_td_ptr wind_data;

	fz_rndr_wind_data_get(windex,my_rndr_index,(fzrt_ptr*)&wind_data);

	return (wind_data->dirty);
}

fzrt_error_td my_rndr_image_disp (
	long					windex,
	fzrt_error_td			prep_err,
	fzrt_rect				*sub_image
	)
{
	my_wind_data_td		*wind_data;
	fzrt_error_td err = FZRT_NOERR;
```

```
        fz_rndr_wind_data_get(windex,my_rndr_index,(fzrt_ptr*)&wind_data);

        if ( prep_err == FZRT_NOERR )
        {
                /* RENDER THE IMAGE */
                ...
                if ( err == FZRT_NOERR )
                {       wind_data->dirty = FALSE;
                }
        }


        return(err);
}
```

The 2d vector/polygon export function (recommended, if the renderer is vector or polygonal. Not used for pixel based renderers).

```
fzrt_error_td  fz_rndr_cbak_expt_vect_out(
        long                        windex,
        char                        *prg_str,
        char                        *fname,
        fzrt_boolean                do_back,
        fz_expt_vect_ptr            expt_vect
        );
```

This function is called only for vector and polygon based renderers. It is called when the user exports the rendered image to a 2d image file that supports vector graphics, such as Adobe Illustrator or HPGL. The function needs to call the expt_vect API functions, which are equivalent to drawing the image on the screen. These functions usually take the expt_vect argument passed into this function as an argument as well. If this function is not implemented, the rendered image cannot be exported to those image file formats.

```
fzrt_error_td  my_rndr_expt_vect_out(
        long                        windex,
        char                        *prg_str,
        char                        *fname,
        fzrt_boolean                do_back,
        fz_expt_vect_ptr            expt_vect
        )
{
        fzrt_error_td err = FZRT_NOERR;

        /* IMAGE EXPORT CODE GOES HERE */
        ...

        return(err);
}
```

The 2d export check feature function (recommended, if the renderer is vector or polygonal. Not used for pixel based renderers).

```
fzrt_error_td  fz_rndr_cbak_expt_vect_check_feature (
        long                              windex,
        fz_rndr_expt_vect_feature_enum    rndr_expt_vect_feature,
        fzrt_boolean                      *check
        );
```

This function is called only for vector and polygon based renderers. It is called when the user exports the rendered image to a 2d image file that supports vector graphics, such as Adobe Illustrator or HPGL. Different file format types support different features of data that can be exported, and so this function informs **form•Z** which features this exporter is exporting. The different features that **form•Z** cares about exporting are contained in the enum `fz_rndr_expt_vect_feature_enum`.

```
fzrt_error_td my_expt_check_feature(
	long                            windex,
	fz_rndr_expt_vect_feature_enum  rndr_expt_vect_feature,
	fzrt_boolean                    *check
	)
{
	if(check != NULL)
	{
		switch(rndr_expt_vect_feature)
		{
			case FZ_RNDR_EXPT_VECT_FEATURE_EXPT_LAYER:
			case FZ_RNDR_EXPT_VECT_FEATURE_EXPT_OBJ:
			case FZ_RNDR_EXPT_VECT_FEATURE_EXPT_FACE:
			case FZ_RNDR_EXPT_VECT_FEATURE_EXPT_COLOR:
				*check = TRUE;
			break;

			case FZ_RNDR_EXPT_VECT_FEATURE_EXPT_FILL_POLY:
				*check = FALSE;
			break;
		}
	}
	return(FZRT_NOERR);
}
```

The dialog template function (recommended, if the renderer has window options).

```
fzrt_error_td fz_rndr_cbak_iface_tmpl (
	long                    windex,
	fz_fuim_tmpl_ptr        tmpl_mngr,
	fzrt_ptr                wind_opts
	);
```

form•Z calls this function to create a dialog template to display the window options of the renderer. This can be done using the functions in the fz_fuim_fset function set defined in "fz_fuim_API.h".

```
fzrt_error_td my_rndr_iface_tmpl (
	long                    windex,
	fz_fuim_tmpl_ptr        tmpl_mngr,
	fzrt_ptr                wind_opts
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	/* TEMPLATE SETUP CODE GOES HERE */

	return(err);
}
```

The options get parameter function (recommended, if the renderer has window options which will be accessed via other extensions).

```
fzrt_error_td fz_rndr_cbak_get_parm (
	long            windex,
	long            parm_indx,
	fz_type_td      *data
```

```
        );
```

Other extensions call this function to get the parameter options that this renderer chooses to expose for potential interaction. **form•Z** does not call this function. A plugin developer would expose the range of possible indices for the `parm_indx` parameter, the types they correspond to, the possible range of values, and the default values of the data they represent. The `fz_type_set_` functions are used to pass the data values from the plugin's representation to the generic variable `data` of type `fz_type_td`, so that whoever calls this function can retrieve the value of the parameter index they specify.

```
fzrt_error_td my_rndr_get_parm (
        long                windex,
        long                parm_indx,
        fz_type_td          *data
        )
{
        fzrt_error_td err = FZRT_NOERR;
        my_opts_td* my_opts;

        err = fz_rndr_wind_opts_get(windex, MY_PLUGIN_UUID, (fzrt_ptr*)&my_opts);

        if (err == FZRT_NOERR &&
                data != NULL && parm_indx >= MY_PARM_1 && parm_indx < MY_PARM_MAX)
        {
                switch(parm_indx)
                {
                        case MY_PARM_1:
                                fz_type_set_boolean(&my_opts->parm1, data);
                                break;
                        case MY_PARM_2:
                                fz_type_set_double(&my_opts->parm2, data);
                                break;
                        case MY_PARM_3:
                                fz_type_set_boolean(&my_opts->parm3, data);
                                break;
                        ...
                }
        }
        else
        {
                err = _base_funcs.fzrt_error_set(FZRT_BAD_PARAM_ERROR,
                        FZRT_ERROR_SEVERITY_ERROR, FZRT_ERROR_CONTEXT_FZRT, 0);
        }
        return err;
}
```

The options set parameter function (recommended, if the renderer has window options which will be accessed via other extensions).

```
fzrt_error_td fz_rndr_cbak_set_parm (
        long                windex,
        long                parm_indx,
        fz_type_td          *data
        );
```

Other extensions call this function to set the parameter options that this renderer chooses to expose for potential interaction. **form•Z** does not call this function. A plugin developer would expose the range of possible indices for the `parm_indx` parameter, the types they correspond to, the possible range of values, and the default values of the data they represent. The `fz_type_get_` functions are used to pass the data values from the generic variable `data` of type `fz_type_td` to the plugin's representation, so that whoever calls this function can set the value of the parameter index they specify. Note that when allowing someone to set a parameter, it is up to the plugin to enforce any range constraints of the variables (see in the example that follows).

```
fzrt_error_td my_rndr_set_parm (
	long			windex,
	long			parm_indx,
	fz_type_td		*data
	)
{
	fzrt_error_td err = FZRT_NOERR;
	my_opts_td* my_opts;
	double tmp_double;

	err = fz_rndr_wind_opts_get(windex, MY_PLUGIN_UUID, (fzrt_ptr*)&my_opts);

	if (err == FZRT_NOERR &&
		data != NULL && parm_indx >= MY_PARM_1 && parm_indx < MY_PARM_MAX)
	{
		switch(parm_indx)
		{
			case MY_PARM_1:
				fz_type_get_boolean(data, &my_opts->parm1);
				break;
			case MY_PARM_2:
				fz_type_get_double(data, &tmp_double);
				if (tmp_double >= MY_PARM_2_MIN_VALUE &&
				    tmp_double <= MY_PARM_2_MAX_VALUE)
				{
					my_opts->parm2 = tmp_double;
				}
				else
					.../* set error */
				break;
			case MY_PARM_3:
				fz_type_get_boolean(data, &my_opts->parm3);
				break;
			...
		}
	}
	else
	{
		err = _base_funcs.fzrt_error_set(FZRT_BAD_PARAM_ERROR,
			FZRT_ERROR_SEVERITY_ERROR, FZRT_ERROR_CONTEXT_FZRT, 0);
	}
	return err;
}
```

Background renderers

It is possible, that a plugin renderer is set up, so that the actual rendering calculation occurs in a background process. In that case, a user may select the rendering mode, but can keep on working on the scene without having to wait until the rendering is completed. For a plugin to function as a background renderer only two actions need to be taken. First, the `fz_rndr_cbak_attr` callback function must return TRUE for the `FZ_RNDR_ATTR_IS_BACKGROUND` question. Second, the plugin may optionally implement the `fz_rndr_cbak_notify_user` callback (see below). It gives the plugin the chance to notify a user, that a previously started rendering is now complete. The plugin may post a dialog to let the user know, or may even display the image by using the `fz_file_display_in_viewer()` api function. It is important, that this callback function does not spend a lot of time determining that a rendering is complete. It is called quite frequently in the main event loop and would slow form•Z down significantly if it does not execute fast.

When the user selects a background renderer from the Display menu, first the `fz_rndr_cbak_image_init` callback is invoked. The renderer may prepare any data necessary for the rendering. This should not be done in a background process. Next, the `fz_rndr_cbak_image_disp` is invoked. The renderer is expected to fire off the main rendering in a background process and return back to form•Z immediately. Finally `fz_rndr_cbak_image_finit` is invoked to allow the renderer to clean up. Note, that the active rendering mode does not change to the renderer selected by the user. For example, if wireframe is active when the user selects the background renderer, wireframe will remain the active rendering mode.

The user notification function (optional)

```
void   fz_rndr_cbak_notify_user(
       void
       );
```

This function is invoked, if the plugin is a background renderer. It is called at regular intervals and gives the plugin the chance to let the user know, that an image that was rendered in the background is now complete. See the section titled "Background renderers" above.

Using **form•Z** API functions to support a plugin renderer

Surface styles

A vector renderer usually displays an object using the simple color representation of the surface style assigned to the object. The rgb values of the color can be retrieved with the following API function calls.

```
fz_objt_attr_get_objt_rmtl(windex, obj, TRUE,&rmtl_tag);
fz_rmtl_tag_to_ptr(windex,rmtl_tag,&rmtl);
fz_rmtl_get_parm(windex,rmtl,FZ_RMTL_PARM_SIMPLE_COLOR,&data);
fz_type_get_rgb_float(&data, &rgb);
```

The sample code above first retrieves the tag of the surface style (abbreviated rmtl for "render material") assigned to the object. Then the tag is converted to a pointer. From the surface style pointer, the simple color parameter is acquired.

A pixel renderer may need to use more than a simple color to render the object. For example the rendere may want to use textures and patterns. Currently it is not possible for a plugin renderer to use the shaders in a Surface Style for that purpose. They are exclusively reserved for the RenderZone rendering command. A plugin renderer may however, extract generic material properties from a surface style. For example, the surface style function set allows a plugin renderer to extract the diffuse factor of a surface style's reflection shader with the API call `fz_rmtl_get_diffuse_factor`. A plugin may also extract the color shader from the surface style and then check whether the color shader uses a texture map. If this is the case, the plugin renderer may acquire the texture map and use it for its one rendering display. This can be done with the following API calls:

```
fz_rmtl_get_parm(windex,rmtl,FZ_RMTL_PARM_COL_SHADER,&data);
fz_type_get_ptr(&data, &shdr_ptr);
fz_shdr_get_parm_type(windex,shdr_ptr,1,&type);
if ( type == FZ_TYPE_PTR )
{
	fz_shdr_get_parm(windex,shdr_ptr,1,&data);
	fz_type_get_ptr(&data, &tmap_ptr);
}
```

The call `fz_rmtl_get_parm(windex,rmtl,FZ_RMTL_PARM_COL_SHADER,&data);` returns the color shader pointer of the surface style. From the color shader, the type of the second parameter is retrieved with the API call `fz_shdr_get_parm_type`. If it is a generic pointer (`FZ_TYPE_PTR`), the shader is a texture map based shader. This is a convention for all shaders in **form•Z**. The actual texture map pointer is acquired with a call to `shdr_param_get_param`. In the same manner, transparency and bump maps may be retrieved from a surface style.

It is also possible for a plugin renderer to create its own surface style definitions. This must be done using a second plugin, which creates a custom attribute (see section 2.8.1 for more details about plugin attributes). The attribute should be setup in such a way that it is tagged as an object and face level attribute. An additional tool command plugin may be added, which creates a tool icon. When used, this tool should assign the custom surface style attribute to a selected object or face. Finally, the custom surface style attributes can be displayed in a palette using a project level palette plugin. It is up to the plugin developer to coordinate the different plugins to work in a coherent fashion.

Lights

Information about the lights in the scene can be acquired using the `fz_lite_fset` function set. It allows a rendering plugin the loop through all lights and extract the parameter of each lite. Using these parameters, the plugin renderer can compute surface illumination. Note that shadow calculation must be performed by the plugin renderer. The shadow type attribute of a light (Mapped or Raytraced) is currently only applied to the Shaded Render and RenderZone rendering modes. If possible, a pixel based plugin renderer should implement its own raytraced and shadow map algorithms.

A sample loop which iterates through all lights in a project and extracts the light type is shown below:

```
fz_lite_ptr lite_ptr;
fz_lite_get_next_light(windex,NULL,&lite_ptr);
while ( lite_ptr != NULL )
{
	fz_lite_get_parm_common(windex,lite_ptr,FZ_LITE_PARM_TYPE,&data);
	fz_type_get_enum(&data, &lite_type);

	...

	fz_lite_get_next_light(windex,lite_ptr,&lite_ptr);
}
```

Texture map controls and Decals

If a pixel based plugin renderer offers texture mapped surface rendering, the texture map control attribute of an object should be taken into account. Attribute API functions supplied by **form•Z** can be used to extract the parameters of texture map control that may have been assigned to an object. The API functions can be found in the function set `fz_model_attr_fset`. If an object does not have a texture map control attribute, the plugin renderer should use the global texture map control, which is stored with the project rendering options. These settings can be retrieved with the API function call `fz_proj_rndr_opts_tctl_get`. Likewise, object decals can be extracted from an object with the respective attribute API call, which are also located in the `fz_model_attr_fset` function set.

An example of how to extract the origin of the object level texture group of a texture map control attribute assigned to an object is shown below.

```
fz_objt_attr_objt_has_tctl(windex,obj,&has_tcntl);
if ( has_tcntl == TRUE )
{
    /* OBJECT HAS A TEXTURE MAP CONTROL ATTRIBUTE */
    fz_objt_attr_get_objt_tctl_parm(windex,obj,0,
        FZ_ATTR_TCTL_PARM_ORIGIN,&data);
    fz_type_get_xyz(&data, &origin);
    ...
}
else
{
    /* OBJECT DOES NOT HAVE A TEXTURE MAP CONTROL ATTRIBUTE */
    /* GET THE ORIGIN FROM THE PROJECT TCNTRL SETTINGS */
    ...
}
```

Geometry

Vector and polygon based renderers can extract the geometry of an object by traversing the structure of the object. For example, the end points of the edges of a facetted object can be found with the following loop:

```
fz_objt_get_segt_count(windex,obj,FZ_OBJT_MODEL_TYPE_FACT,&nsegt);
for(i = 0; i < nsegt; i++)
{
        fz_objt_segt_get_reverse(windex,obj,i,FZ_OBJT_MODEL_TYPE_FACT,&rvrs);
        fz_objt_segt_get_next(windex,obj,i,FZ_OBJT_MODEL_TYPE_FACT,&next);
        if ( i > rvrs  && next ! -1)
        {
            fz_objt_segt_get_start_pindx(windex,obj,i,FZ_OBJT_MODEL_TYPE_FACT,&pindx1);
            fz_objt_segt_get_end_pindx(windex,obj,i,FZ_OBJT_MODEL_TYPE_FACT,&pindx2);

            fz_objt_point_get_xyz(windex,obj,pindx1,FZ_OBJT_MODEL_TYPE_FACT,&xyz1);
            fz_objt_point_get_xyz(windex,obj,pindx2,FZ_OBJT_MODEL_TYPE_FACT,&xyz2);

            ...
        }

}
```

Recall that in a solid object there are two segments for each edge (reversely coincident). By getting the reverse segment index with `fz_objt_segt_get_reverse`, and only using the segment whose index is larger, one can ensure that each edge is accessed only once. This algorithm also works for segments that don't have a reverse segment (open edges of surfaces). In addition, it is necessary to check that a segment is not the dummy end segment of an open wire. In this case, the segment does not have an end point.

The sample loop above only accesses the edges of a facetted object. if a vector renderer wants to draw the edges of smooth objects, a different loop needs to be constructed. The edges of smooth object may be curved. In this case, sample points along a curved edge may need to be extracted which represent the smooth edge. These points are already stored with the object and are also used by the Wire Frame rendering mode. A loop for smooth objects is shown below.

```
fz_objt_get_segt_count(windex,obj,FZ_OBJT_MODEL_TYPE_SMOD,&nsegt);
for(i = 0; i < nsegt; i++)
{
	fz_objt_segt_get_reverse(windex,obj,i,FZ_OBJT_MODEL_TYPE_SMOD,&rvrs);
	fz_objt_segt_get_next(windex,obj,i,FZ_OBJT_MODEL_TYPE_SMOD,&next);
	if ( i > rvrs  && next ! -1)
	{
	    fz_objt_segt_get_num_wire_pnts(windex,obj,i,&npnts);
	    for(j = 0; j < npnts; j++)
	    {
	        fz_objt_segt_get_wire_pnt(windex,obj,i,j,&pt_xyz);
	        ...
	    }
	}

}
```

The code above only works though if the object is a smooth object. It is up to the plugin to check for the correct object model type.

A vector renderer may also draw iso lines across the faces of smooth objects, as it is done by the Wire Frame rendering mode. Sample code which extracts the points of iso line edges of smooth objects is shown below.

```
fz_objt_get_face_count(windex,obj,FZ_OBJT_MODEL_TYPE_SMOD,&nface);
for(i = 0; i < nface; i++)
{
    model_face_get_num_iso_lines(windex,obj,i,&niso);
    for(j = 0; j < niso; j++)
    {
        model_face_get_iso_pnt(windex,obj,i,j,&pt_xyz);
        ...
    }
}
```

Finally, a vector renderer may display the facetted faces of smooth objects different than the faces of facetted objects. In this case, the renderer needs to loop through the smooth faces and extract the facetted faces that belong to each smooth face. This is shon in the sample code below:

```
fz_objt_get_face_count(windex,obj,FZ_OBJT_MODEL_TYPE_SMOD,&nface);

/* LOOP FOR ALL SMOOTH FACES OF AN OBJECT */
for(i = 0; i < nface; i++)
{
    fz_objt_face_smod_get_fact_faces(windex,obj,i,&fstart,&nface);

    for(j = fstart; j <= fstart + nface; j++ )
    {
        fz_objt_face_get_cindx(windex,obj,j,FZ_OBJT_MODEL_TYPE_FACT,&cindx);
        chead = cindx;

        /* LOOP FOR ALL CURVES OF A FACE */
        do
        {
```

```
            fz_objt_curv_get_sindx(windex,obj,cindx,FZ_OBJT_MODEL_TYPE_FACT,&sindx);
            shead = sindx;

            /* LOOP FOR ALL SEGMENTS OF A CURVE */
            do
            {
                /* COLLECT THE START POINTS OF THE SEGMENTS */
                fz_objt_segt_get_start_pindx(windex,obj,i,
                        FZ_OBJT_MODEL_TYPE_FACT,&pindx1);
                fz_objt_point_get_xyz(windex,obj,pindx1,
                        FZ_OBJT_MODEL_TYPE_FACT,&xyz1);
                ...

                fz_objt_segt_get_next(windex,obj,sindx,FZ_OBJT_MODEL_TYPE_FACT,&sindx);
            } while ( sindx != shead && sindx != -1 );

            fz_objt_curv_get_next(windex,obj,cindx,FZ_OBJT_MODEL_TYPE_FACT,&cindx);
        } while (cindx != chead);
    }
}
```

Pixel based renderers often need to decompose the faces of an object into a set of triangles. This can be done with the API call:

```
fz_type_list_init(FZ_TYPE_LONG,&pindx_list);
fz_type_list_init(FZ_TYPE_LONG,&findx_list);

fz_objt_decompose_simple(windex,obj,FZ_DECOMP_SIMPLE_ALL_TRIA,
    num_triang,pindx_list,findx_list);

for(i = 0, j = 0; i < num_triang; i++, j+= 4)
{
    for(k = 1; k < 4; k++)
    {
        fz_type_list_get_item(pindx_list,j+k,&data);
        fz_type_get_long(&data,&pindx);

        fz_objt_point_get_xyz(windex,obj,pindx,FZ_OBJT_MODEL_TYPE_FACT,&pts[k]);
        ...
    }

    fz_type_list_get_item(findx_list,i,&data);
    fz_type_get_long(&data,&findx);

    /* pts NOW CONTAINS THE THREE POINTS OF THE TRIANGLE */
    /* findx IS THE INDEX OF THE ORIGINAL FACETTED FACE */
    ...
}

fz_type_list_finit(&pindx_list);
fz_type_list_finit(&findx_list);
```

The data returned is a list of point indices, which, three at a time, define a triangle. The point indices refer to the coordinate point of the facetted representation of the original object. In addition, this function also returns which original face a triangle belongs to. The indices of the original facetted faces are stored in the face index list. This list contains as many entries, as there are triangles.

2.8.7 RenderZone Shaders

The shader pipeline

When a pixel in an image is rendered, the shaders needed to compute the final pixel color are executed in a specific order. This order is referred to as the shader pipeline. The sequence of the shader pipeline for each pixel is as follows:
1. The color shader of the material assigned to the surface on which the pixel lies is executed. This defines the unshaded pixel color.
2. The bump shader of the material assigned to the surface on which the pixel lies is executed. This defines a new normal direction at the pixel, which is important for the reflection calculation which comes next.
3. The reflection shader of the material assigned to the surface on which the pixel lies is executed. The unshaded pixel color, generated by the color shader is augmented with shading information from all lights in the scene. If a bump shader other than None was used, the altered surface normal direction will be used to create bump patterns from the shading calculation. The shaded color is returned by the reflection shader.
4. The transparency shader of the material assigned to the surface on which the pixel lies is executed. The transparency of the pixel is returned by the shader and retained by **form•Z**.
5. If the transparency value from step 4 is more than 0.0 (i.e. there is some level of transparency) the background shader is executed. The color from the background shader and the shaded color from step 3 are mixed using the transparency value and returned by the shader.
6. The depth effect shader is executed. It uses the color from step 5. A new color is calculated using the depth information of the current pixel. This color is returned and becomes the final pixel color in the image.

Any of the six shaders contained in the shader pipeline can be extended through a plugin. Color, reflection, transparency and bump extension shaders are added to the respective menus in the Surface Style Parameters dialog. Background and Depth Effect plugin shaders are added in the RenderZone Options dialog. A Background plugin shader also becomes available as an Environment shader.

Shader plugin type and registration

An shader plugin is identified with the plugin type `FZ_SHDR_EXTS_TYPE` and must implement one of the 6 shader call back function sets, `fz_shdr_cbak_colr_fset`, `fz_shdr_cbak_refl_fset`, `fz_shdr_cbak_trns_fset`, `fz_shdr_cbak_bump_fset`, `fz_shdr_cbak_bgnd_fset`, or `fz_shdr_cbak_fgnd_fset`. Multiple shader function sets may be registered with the plugin. This allows for the creation of a whole suite of shaders in a single plugin. For example a developer may create a family of wall-paper shaders and offer those as a single plugin to **form•Z** users. Most of the color shaders already available in a **form•Z** surface style also have transparency and bump shader equivalents. When creating a new color shader, offering the transparency and bump shader twins may be another method to register multiple shaders with the same plugin. The example below show the definition of a plugin of type `FZ_SHDR_EXTS_TYPE` and the registration of a color, transparency and bump shader with that plugin. This is done from the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3.

```
	fzrt_error_td sinewave_register_plugins()
{
	fzrt_error_td		err = FZRT_NOERR;

	/* REGISTER THE PLUGIN */
```

```
	err = fzpl_glue->fzpl_plugin_register(
		SINEWAVE_PLUGIN_UUID,
		SINEWAVE_PLUGIN_NAME,
		SINEWAVE_PLUGIN_VERSION,
		SINEWAVE_PLUGIN_VENDOR,
		SINEWAVE_PLUGIN_URL,
		FZ_SHDR_EXTS_TYPE,
		FZ_SHDR_EXTS_VERSION,
		NULL /*error string function*/,
		0,
		NULL,
		&sinewave_plugin_runtime_id);

	if ( err == FZRT_NOERR )
	{
		/* REGISTER THREE SHADER CALLBACK FUNCTION SETS */
		err = fzpl_glue->fzpl_plugin_add_fset(
				sinewave_plugin_runtime_id,
				FZ_SHDR_CBAK_COLR_FSET_TYPE,
				FZ_SHDR_CBAK_COLR_FSET_VERSION,
				FZ_SHDR_CBAK_COLR_FSET_NAME,
				FZPL_TYPE_STRING(fz_shdr_cbak_colr_fset),
				sizeof (fz_shdr_cbak_colr_fset),
				colr_sinewave_cbak_fill_fset, FALSE);

		if ( err == FZRT_NOERR )
		{
			err = fzpl_glue->fzpl_plugin_add_fset(
					sinewave_plugin_runtime_id,
					FZ_SHDR_CBAK_TRNS_FSET_TYPE,
					FZ_SHDR_CBAK_TRNS_FSET_VERSION,
					FZ_SHDR_CBAK_TRNS_FSET_NAME,
					FZPL_TYPE_STRING(fz_shdr_cbak_trns_fset),
					sizeof (fz_shdr_cbak_trns_fset),
					trns_sinewave_cbak_fill_fset, FALSE);
		}

		if ( err == FZRT_NOERR )
		{
			err = fzpl_glue->fzpl_plugin_add_fset(
					sinewave_plugin_runtime_id,
					FZ_SHDR_CBAK_BUMP_FSET_TYPE,
					FZ_SHDR_CBAK_BUMP_FSET_VERSION,
					FZ_SHDR_CBAK_BUMP_FSET_NAME,
					FZPL_TYPE_STRING(fz_shdr_cbak_bump_fset),
					sizeof (fz_shdr_cbak_bump_fset),
					bump_sinewave_cbak_fill_fset, FALSE);
		}
	}

	return (err);
}
```

Shader call back function sets

Shader plugins are implemented by defining one of the shader call back function sets. The plugin developer must pass a fill function to `fzpl_plugin_add_fset` which assigns the pointers of the functions which define the plugins functionality to an instance of the shader callback function set. An example of the fill function for the color shader "Sine Wave" is shown below. The function sets for the other shader types are very similar to the color shader.

```
fzrt_error_td colr_sinewave_cbak_fill_fset (
      const fzpl_fset_def_ptr fset_def,
      fzpl_fset_td * const fset )
{
      fzrt_error_td                  err = FZRT_NOERR;
      fz_shdr_cbak_bump_fset         * colr_fset;

      err = fzpl_glue->fzpl_fset_def_check ( fset_def,
                  FZ_SHDR_CBAK_COLR_FSET_VERSION,
                  FZPL_TYPE_STRING(fz_shdr_cbak_bump_fset),
                  sizeof (fz_shdr_cbak_bump_fset),
                  FZPL_VERSION_OP_NEWER );

      if ( err == FZRT_NOERR )
      {
            colr_fset = (fz_rzne_colr_shdr_cb_fset *)fset;

            colr_fset->fz_shdr_cbak_colr_uuid = colr_sinewave_uuid;
            colr_fset->fz_shdr_cbak_colr_name = colr_sinewave_name;
            colr_fset->fz_shdr_cbak_colr_version = colr_sinewave_get_version;
            colr_fset->fz_shdr_cbak_colr_set_parameters =
                                          colr_sinewave_set_parameters;
            colr_fset->fz_shdr_cbak_colr_pre_render = colr_sinewave_pre_render;
            colr_fset->fz_shdr_cbak_colr_post_render = colr_sinewave_post_render;
            colr_fset->fz_shdr_cbak_colr_pixel = colr_sinewave_colr_shade_pixel;
            colr_fset->fz_shdr_cbak_colr_avg = colr_sinewave_avg_color;
      }

      return err;
}
```

Of the eight callback functions of a color shader, only some are required, while others are optional. When an optional callback is not assigned to the function set, the respective functionality of the shader is disabled. For example, if the `fz_shdr_cbak_colr_avg` callback function is not provided, **form•Z** will substitute a 50% gray for the color, whenever a single solid color is used, such as in wireframe drawing. The required callback functions for a color shader are:

```
fz_shdr_cbak_colr_name
fz_shdr_cbak_colr_pixel
```

Optional functions are:

```
fz_shdr_cbak_colr_set_parameters
fz_shdr_cbak_colr_pre_render
fz_shdr_cbak_colr_post_render
fz_shdr_cbak_colr_get_avg
```

The functions shown below are taken from the Sine Wave color shader, which is available as sample source code with **form•Z**. It is recommended to build this shader with the respective development environment on Mac or Windows and trace the execution of the function with the debugging environments of the compiler.

The following section gives a detailed description of each of the shader functions and what task each function is expected to perform. The functions are explained in detail for the color shader. Any differences for the equivalent function of the other shaders are noted where necessary.

The uuid function (recommended)

```
fzrt_error_td  fz_shdr_cbak_colr_uuid(
       fzrt_UUID_td  uuid
       );
```

This function defines a unique identifier. **form•Z** uses the UUID to distinguish this shader from other shaders.

```
fzrt_error_td sinewave_colr_uuid(fzrt_UUID_td uuid)
{
       fzrt_UUID_copy(SINEWAVE_COLR_PLUGIN_UUID, uuid);
       return FZRT_NOERR;
}
```

The version function (recommended)

```
fzrt_error_td  fz_shdr_cbak_colr_version(
       fzpl_vers_td *version
       );
```

If this function is implemented, it needs to return the version of the shader. It is up to the developer to assign a version number to the shader. When a **form•Z** project file is saved with a plugin shader, the version of the shader is saved as well. If the project is opened later and a newer version of the shader exists at that time, **form•Z** will reset the parameters of the shader to default values. A shader developer must increase the version number when, during ongoing development of the shader, the parameters of the older shader do not match the parameters of the newer shader. If the shader is changed, so that saved shader parameters are still meaningful and are aligned with the current shader parameters, the version does not need to be changed. Assume, for example, that a shader is originally defined with 2 color and 2 integer parameters. The version assigned to the shader initially was 0. In the second release of the shader, the developer adds a 5^{th} parameter. This requires, that the version is increased to 1. In a third release of the shader, the first integer parameter, which originally could take on values between 0 and 10, can now take on values from 0 to 20. This does not require a version change.

```
fzrt_error_td sinewave_version(fzpl_vers_td* version)
{
       *version = 0;
       return FZRT_NOERR;
}
```

The name function (required)

```
fzrt_error_td      fz_shdr_cbak_colr_name(
       char        *name,
       long        max_len
       );
```

The name function must assign a string to the name argument. The length of the string assigned cannot exceed max_len characters. This string appears as the shader's name in the respective menu. It is recommended, that the name is stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example below, this step is omitted for the purpose of simplicity. A plugin name function would look like this:

```
fzrt_error_td sinewave_get_name(char *name, long max_len)
{
	strncpy(name,"Sine Wave",max_len);
	return(FZRT_NOERR);
}
```

The set parameters function (optional)

```
fzrt_error_td	fz_shdr_cbak_colr_set_parameters(
	void
	);
```

The set parameters function is called once at startup. It needs to establish the number and types of parameters for the shader. Based on the parameters set up in this function, **form•Z** automatically builds the content of the shader's option dialog, which can be invoked by clicking on the Options... botton next to the shader menu, as usual. The content of a shader option dialog can also be created by implementing the optional dialog callback function. If provided, **form•Z** will not automatically create the content of the dialog, but the callback function is invoked and expected to create the template interface items to correctly display the shader parameters. This is explained in more detail in a following section.

Setting the shader's parameters is accomplished with a number of **form•Z** API function calls. There are standard parameters which can be set up automatically, such as scale or noise. Custom parameters can be created individually, such as colors, floating point values with sliders or check boxes. If the shader is a color, transparency or bump shader, fhe first **form•Z** API call in the set parameters function should identify the shader as a 2d (wrapped) or 3d (solid) shader. This is done with the API call:

```
	fz_shdr_set_wrapped(TRUE);
```

if the shader is 2d, and

```
	fz_shdr_set_solid(TRUE);
```

if the shader is 3d. Note, not calling these functions is equivalent to calling either function with the argument set to `FALSE`. It is also possible to call both function with `TRUE`, in which case the shader would be labeled as a 2d and 3d shader. While this is rarely the case, it is conceivable, that a shader creates a pattern based on 2d and 3d texture space mapping. Mirror, background and depth effect shaders do not need to call this API function.

Shaders which create a pattern should present the standard scale parameter to a user. This parameter is set up with the API call:

```
	fz_shdr_set_scale_parm (1.0);
```

The function argument 1.0 sets the default value of the scale parameter to 100%. This function call will automatically add the Scale field in the shader options dialog. **form•Z** will apply the current scale factor to the 2d or 3d texture space coordinate, which is used in the pixel function to calculate the shader's pattern.

If a shader uses any of the noise functions, which create random patterns, the standard noise parameters can be added ot the shader with the API call:

```
	fz_shdr_set_noise_parm(1,3);
```

This will add the Noise menu and # of Impulses field to the shader option dialog. The current setting of these parameters may be retrieved in the pre_render function and used in a call to any of the noise functions in the shader's pixel function.

Most procedural shaders, which create some kind of pattern suffer from strong moire artifacts, when the pattern becomes very small. With an area sampling technique, these artifacts can be avoided. Automatic area sampling can be added to a color, transparency or bump shader by adding the standard shader parameter with the API function call:

```
	fz_shdr_set_area_sample_parm(FALSE);
```

The argument in the API function call sets the default value of area sampling to TRUE or FALSE (FALSE should be the default). The standard "Area Sampling" check box will be added by **form•Z** in the shader dialog. If this API call is not made in the set parameters callback, the shader will not have area sampling. Note, that his call only applies in the set parameters function of color, transparency and bump shaders. For all other shader types, this API call is ignored.

If the shader is a reflection shader, additional standard parameters can be set up. They define the six shading parameters: ambient, diffuse, specular, mirror, transmission and glow:

```
	fz_shdr_set_ambient_parm (1.0);
	fz_shdr_set_diffuse_parm (0.75);
	fz_shdr_set_specular_parm (0.5,0.1);
	fz_shdr_set_specular_color_parm (col);
	fz_shdr_set_mirror_parm (0.5);
	fz_shdr_set_transmission_parm (0.5,1.0);
	fz_shdr_set_glow_parm (0.0);
```

When the respective setup call is made, the shader options dialog will add the Factor field, Map menu and map Options button. Not all six reflection parameters need to be offered. Any combination of the six can be selected and mixed with custom parameters.

Custom parameters are created with the API calls:

```
	fz_shdr_set_pct_parm("Value 1", 0.5, 1, 1, SHDR_VAL1_ID);
	fz_shdr_set_col_parm("Color 1", col, SHDR_COL_ID);
	fz_shdr_set_sld_flt_parm("Value 2", 0.5,1,1, SHDR_VAL2_ID);
	fz_shdr_set_sld_int_parm("Value 3", 5,1,10,1,1, SHDR_VAL3_ID);
	fz_shdr_set_flt_parm("Value 4", 0.5, 0.0, 1.0, 1, 1, SHDR_VAL4_ID);

	fz_shdr_set_int_parm("Value 5", 5, 1, 10, 1, 1, SHDR_VAL5_ID);
	fz_shdr_set_bool_parm("Boolean", TRUE, SHDR_BOOL_ID);
```

Each of these calls creates a shader parameter of the respective type, with the given title, default values, allowable range and range checking. The last parameter to each function is an integer id, which must be unique. This id is used when retrieving the current value of a parameter in the pre render function. It is possible to pass a value of -1 for the id argument. In this case **form•Z** will generate a unique id and pass it back through the function's return value. For example:

```
	id = fz_shdr_set_col_parm("Color 1", col, -1);
```

Since the **form•Z** generated id must be used to retrieve the parameter value in the pre render function, it must be a global variable.

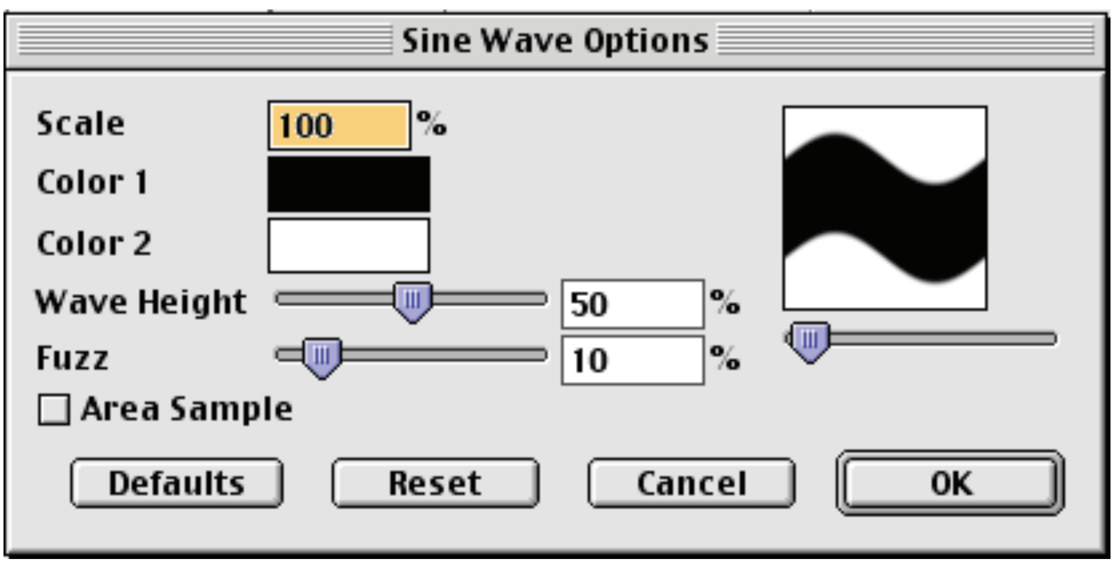
A user may edit the preset and custom values in the options dialog. In the pre render function the current values of the custom parameters should be retrieved and passed on to the pixel function, where they are used to compute the shader pattern.

The set parameters function for the Sine Wave color shader in a plugin is:

```
enum
{
	PARAM_ID_COLOR1 = 0,
	PARAM_ID_COLOR2,
	PARAM_ID_HEIGHT,
	PARAM_ID_FUZZ
```

```
};

fzrt_error_td     sinewave_colr_set_parameters(void)
{
      fz_rgb_float_td def_col1;
      fz_rgb_float_td def_col2;

      def_col1.red   = 0.0;
      def_col1.green = 0.0;
      def_col1.blue  = 0.0;
      def_col2.red   = 1.0;
      def_col2.green = 1.0;
      def_col2.blue  = 1.0;
      fz_shdr_set_wrapped(TRUE);
      fz_shdr_set_scale_parm(1.0);
      fz_shdr_set_area_sample_parm(TRUE);

      fz_shdr_set_col_parm("Color 1",&def_col1, PARAM_ID_COLOR1);
      fz_shdr_set_col_parm("Color 2",&def_col2, PARAM_ID_COLOR2);
      fz_shdr_set_sld_flt_parm("Wave Height",0.5,1,1, PARAM_ID_HEIGHT);
      fz_shdr_set_sld_flt_parm("Fuzz",0.1,1,1, PARAM_ID_FUZZ);

      return(FZRT_NOERR);
}
```

The dialog resulting from these shader parameters is shown below:



There is one important detail to the use of the custom parameters API functions, such as `fz_shdr_set_sld_flt_parm`. The first parameter to this API is the name of the parameter as it will appear in the shader dialog. A transparency shader is also used to define an equivalent shader in the reflection map menus of a reflection shader, which uses any of the six standard reflection parameters. When the dialog for this shader (if used as a diffuse map for eample) is invoked, the parameter, which would be called, for example, "Background Transparency" in the transparency shader options dialog, is called "Background Diffuse" in the diffuse map options dialog. This automatic adjustment of the parameter name can be achieved by substituting %s in the name parameter of the API, for those calls of the API which would use the word "Transparency" in the dialog. The same mechanism also works for color and bump shaders, although they are not used in any other context. For example the API call to define a color in the set parameters function of a color shader can be written in two different ways:

```
fz_shdr_set_col_parm("Color 1",&def_col1, PARAM_ID_COLOR1);
```

or

```
fz_shdr_set_col_parm("%s 1",&def_col1, PARAM_ID_COLOR1);
```

While it is not necessary to substitute the %s in color and bump shaders, it is necessary to do so in transparency shaders, in order to get the correct parameter title, when the transparency shader is also used in the context of a reflection map shader.

The dialog function (optional)

```
fzrt_error_td fz_shdr_colr_cbak_iface_tmpl(
        long                windex,
        fz_fuim_tmpl_ptr    fuim_tmpl,
        fzrt_shdr_ptr       shdr_ptr
        long                parent_group
);
```

If this function is supplied, **form•Z** will not automatically build the content of the shader options dialog, as described in the "The set parameters function" section. If this function is implemented and the Options… button next to the shader menu is pressed, **form•Z** will invoke this callback function. It is expected to create the dialog items, with which the shader parameters are displayed. This is achieved by using the template interface api function supplied by **form•Z**. This gives the developer the flexibility to create interface items, that are not created by **form•Z** through the automatic method. For example, the interface callback function may create groups with borders or tabs to allow for a complete customized interface. Note, that the preview window, common to all shaders does not need to be created by the dialog callback function, as **form•Z** provides it automatically. The function argument `parent_group`, which is supplied by **form•Z** when the dialog callback is invoked, should be used as the parent group for all dialog items created. This ensures, that the dialog items and the preview window are properly aligned. In order to facilitate the creation of the dialog, **form•Z** also supplies a few utility functions, that create the standard shader dialog interface items. They are :

`fz_shdr_fuim_create_scale_items` : This api function creates the standard Scale items with a percent text edit field. It may be called for any of the shader types. Note, that in the set parameters callback function, the api call fz_shdr_set_scale_parm should be made to establish, that the shader uses the Scale parameter.

`fz_shdr_fuim_create_noise_items` : This api function creates the standard Noise items with a pop up menu for the noise type and a text edit field for the number of impulses. It may be called for any of the shader types. Note, that in the set parameters callback function, the api call `fz_shdr_set_noise_parm` should be made to establish, that the shader uses the Noise parameter.

`fz_shdr_fuim_create_area_sample_items` : This api function creates the standard Area Sample check box. It may be called for any of the shader type, except for reflection shaders. Note, that in the set parameters callback function, the api call `fz_shdr_set_area_sample_parm` should be made to establish, that the shader uses the Area Sample parameter.

```
fz_shdr_fuim_create_ambient_items
fz_shdr_fuim_create_diffuse_items
```

`fz_shdr_fuim_create_specular_items`
`fz_shdr_fuim_create_mirror_items`
`fz_shdr_fuim_create_transmission_items`
`fz_shdr_fuim_create_glow_items` : These api functions create the standard reflection items with a slider and percent text edit field, the Map menu and an Options… button. They can only be called for reflections shaders. Note, that is is necessary to declare in the set parameters callback function, that a particular reflection parameter is used. For example, in order to create the diffuse reflection items in the dialog callback function via `fz_shdr_fuim_create_diffuse_items`, it is necessary to establish that the shader uses diffuse reflection in the `fz_shdr_cbak_refl_set_parameters` callback function through the api call `fz_shdr_set_diffuse_parm`.

`fz_shdr_fuim_create_specular_roughness_items` : This api function creates the standard specular Roughness items with a slider and percent text edit field. It can only be called for reflections shaders. Note, that in the set parameters callback function, the api call `fz_shdr_set_specular_parm` should be made to establish, that the shader uses the Specular Reflection parameter.

`fz_shdr_fuim_create_trans_refraction_items`: This api function creates the standard Refraction text edit field and pop up menu for shader that use the transmission parameter . It can only be called for reflections shaders. Note, that in the set parameters callback function, the api call `fz_shdr_set_transmission_parm` should be made to establish, that the shader uses the Transmission Reflection parameter.

In order to link a particular shader parameter to a **form•Z** interface item, such as a text edit field, it is necessary to extract a pointer to the shader parameter. This is accomplished with the api call `fz_shdr_get_parm_ptr`. As input, this function receives the shader, which is supplied to the callback function by **form•Z**, and an id, which is the same id established for a specific parameter in the set parameters callback function. If the dialog of the sine wave sample shader would be created with the optional dialog callback function it would look as in the following code example. For completeness, the set parameters callback is repeated to show the connection between the declaration of a parameters and its creation in the interface.

```
enum
{
	PARAM_ID_COLOR1 = 0,
	PARAM_ID_COLOR2,
	PARAM_ID_HEIGHT,
	PARAM_ID_FUZZ
};

fzrt_error_td sinewave_colr_set_parameters(void)
{
	fz_rgb_float_td def_col1;
	fz_rgb_float_td def_col2;

	def_col1.green = 0.0;
	def_col1.blue = 0.0;
	def_col2.red = 1.0;
	def_col2.green = 1.0;
	def_col2.blue = 1.0;
	fz_shdr_set_wrapped(TRUE);
	fz_shdr_set_scale_parm(1.0);
	fz_shdr_set_area_sample_parm(TRUE);
	fz_shdr_set_col_parm("Color 1",&def_col1, PARAM_ID_COLOR1);
	fz_shdr_set_col_parm("Color 2",&def_col2, PARAM_ID_COLOR2);
	fz_shdr_set_sld_flt_parm("Wave Height",0.5,1,1, PARAM_ID_HEIGHT);
	fz_shdr_set_sld_flt_parm("Fuzz",0.1,1,1, PARAM_ID_FUZZ);
```

```
	return(FZRT_NOERR);
}

fzrt_error_td sinewave_colr_iface_tmpl(
	long			windex,
	fz_fuim_tmpl_ptr	fuim_tmpl,
	fz_shdr_ptr		shdr_ptr,
	long			parent_group
	)
{
	short		g1;
	fzrt_ptr	parm_ptr;

	/* STANDARD SCALE*/
	fz_shdr_fuim_create_scale_items(fuim_tmpl,shdr_ptr,parent_group,NULL,NULL
);

	/* COLOR 1 */
	fz_shdr_get_parm_ptr(shdr_ptr,PARAM_ID_COLOR1,&parm_ptr);
	g1 = fz_fuim_new_text_static(fuim_tmpl,parent_group, FZ_FUIM_NONE,
	FZ_FUIM_FLAG_HORZ | FZ_FUIM_FLAG_GFLT , "Color 1", NULL,NULL);
	fz_fuim_new_color_box(fuim_tmpl, g1, FZ_FUIM_NONE, FZ_FUIM_FLAG_NONE,
		NULL, (float *)parm_ptr);

	/* COLOR 2 */
	fz_shdr_get_parm_ptr(shdr_ptr,PARAM_ID_COLOR2,&parm_ptr);
	g1 = fz_fuim_new_text_static(fuim_tmpl,parent_group, FZ_FUIM_NONE,
	FZ_FUIM_FLAG_HORZ | FZ_FUIM_FLAG_GFLT , "Color 2", NULL,NULL);
	fz_fuim_new_color_box(fuim_tmpl, g1, FZ_FUIM_NONE, FZ_FUIM_FLAG_NONE,
		NULL, (float *)parm_ptr);

	/* HEIGHT */
	fz_shdr_get_parm_ptr(shdr_ptr,PARAM_ID_HEIGHT,&parm_ptr);
	fz_fuim_new_slid_edit_pcent_float(fuim_tmpl,parent_group,"Height",FZ_FUIM
_NONE,
		FZ_FUIM_NONE,0.0,1.0,0.0,100.0,FZ_FUIM_RANGE_NONE, NULL,
		(float*) parm_ptr,NULL, NULL);

	/* FUZZ */
	fz_shdr_get_parm_ptr(shdr_ptr,PARAM_ID_FUZZ,&parm_ptr);
	fz_fuim_new_slid_edit_pcent_float(fuim_tmpl,parent_group,"Fuzz",FZ_FUIM_N
ONE,
		FZ_FUIM_NONE,0.0,1.0,0.0,100.0,
		FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
		FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL, NULL,
		(float*) parm_ptr,NULL, NULL);


	/*STANDARD AREA SAMPLE */
	fz_shdr_fuim_create_asample_items(fuim_tmpl,shdr_ptr,parent_group,NULL);

	return(FZRT_NOERR);
}
```

The pre render function (recommended)

```
fzrt_error_td fz_shdr_cbak_colr_pre_render(
	void **shdr_data
	);
```

This function is called once before the start of each rendering. It is expected to precompute information that will be used by the pixel function. Using the pre render function can significantly speed up the execution of a shader. Certain information, that is needed during the calculation of the shader pattern does not change during the rendering. For example, a shader may use a floating point value from a shader parameter, but really needs the inverse (1.0 / value) during the pixel calculation. Instead of computing 1.0 / value each time during the execution of the pixel function, the value can be computed once in the pre render function and then be reused in the pixel function. Any of the shader parameters defined in the set parameters function can be retrieved in the pre render function. For the standard parameter function, there are the equivalent function which get the current value of a standard parameter. They are :

```
fz_shdr_get_noise_type
fz_shdr_get_noise_impulses
```

Note, that there is no function to get the scale parameter. **form•Z** automatically applies the scale factor, if it exists, to the texture space or 3D coordinate before it is used by the pixel function. For custom parameters, a single API call retrieves the value of a given parameter:

```
fz_shdr_get_parm(PARAM_ID,&data);
```

The parameter is identified by the first argument to the function, which is the id used when the parameter was defined, or the id generated by **form•Z**, if -1 was passed for the id. The standard reflection parameters for ambient, diffuse, specular, mirror, transmission and glow should not be retrieved in the pre render function but in the pixel function. This is described in more detail later in this section.

The pre render function typically will allocate a data structure, fill it with precomputed information and pass the pointer of the structure back to **form•Z** via the function argument. This pointer will be passed back into the pixel function and also the post_render function, which should de-allocate the structure. The pre_render function for the Sine Wave color shader is shown below.

```
fzrt_error_td sinewave_colr_pre_render(void **shdr_data)
{
	sinewave_data_td*	sinewave;
	fz_type_td		data;
	fzrt_error_td		rv = FZRT_NOERR;
	double			fuzz;

	*shdr_data = NULL;
	if((*shdr_data = fz_mem_zone_alloc(
		NULL,
		sizeof(sinewave_data_td),FALSE)) == NULL)
	{	rv = fzrt_error_set (
				FZ_MALLOC_ERROR,
				FZRT_ERROR_SEVERITY_ERROR,
				FZRT_ERROR_CONTEXT_APP, 0 );
	}
	else
	{	sinewave = (sinewave_data_td*) *shdr_data;
		fz_shdr_get_parm(PARAM_ID_COLOR1,&data);
		fz_type_get_rgb_float(&data, &sinewave->col1);
		fz_shdr_get_parm(PARAM_ID_COLOR2,&data);
		fz_type_get_rgb_float(&data, &sinewave->col2);
		fz_shdr_get_parm(PARAM_ID_HEIGHT,&data);
		fz_type_get_double(&data, &sinewave->ampl);
		fz_shdr_get_parm(PARAM_ID_FUZZ,&data);
		fz_type_get_double(&data, &fuzz);
```

```
            sinewave->ampl *= 0.25;
            if (sinewave->ampl < 0.0) sinewave->ampl = 0.0;

            fuzz *= 0.25;

            if (fuzz < 0.0)  fuzz = 0.0;
            if (fuzz > 0.25) fuzz = 0.25;

            sinewave->min_left  = 0.25 - fuzz;
            sinewave->min_right = 0.25 + fuzz;
            sinewave->max_left  = 0.75 - fuzz;
            sinewave->max_right = 0.75 + fuzz;
      }
      return(rv);
}
```

The pixel function (required)

The pixel function is called during a rendering once of more for each pixel. Depending on which kind of shader is written, the pixel function needs to compute different types of information. The pixel function has a single argument, `shdr_data`. It is the pointer which was returned to **form•Z** by the pre render function. As described above, it is usually a pointer to a data structure which contains precomputed information.

The color pixel function

```
fz_rgb_float_td fz_shdr_cbak_colr_pixel (
      void *shdr_data
      );
```

For a color shader, the pixel function needs to compute and return the rgb color of the surface pixel, based on the 2d or 3d texture coordinate. This coordinate is retrieved via a **form•Z** API call:

```
      fz_shdr_get_tspace_st(&st);
```

for 2d shaders or

```
      fz_shdr_get_tspace_pnt(&pnt);
```

for 3d shaders. Note, that the scale factor, set up in the set parameters function does not need to be applied to the 2d or 3d texture space coordinate, as **form•Z** already has performed this step. Together with the shader parameters, the point's coordinates can be transformed into a color pattern. A number of **form•Z** API function are offered to facilitate the computation of regular and random patterns. This is described in further detail in later in this section. The pixel function of the Sine Wave color shader is shown below:

```
fz_rgb_float_td sinewave_colr_shade_pixel(void *shdr_data)
{
      fz_xy_td                st;
      sinewave_data_td*       sinewave;
      double                  ss,tt;
      fz_rgb_float_td         col;

      sinewave = (sinewave_data_td*) shdr_data;

      shdr_get_tspace_st(&st);

      ss = fz_shdr_saw_tooth(st.x,1.0);
      tt = fz_shdr_saw_tooth(st.y,1.0) +
```

```
                sin(ss * _2PI)*sinewave->ampl;
      tt = fz_shdr_saw_tooth(tt,1.0);

      tt = fz_shdr_smooth_step(
                sinewave->min_left, sinewave->min_right, tt) *
            (1.0 - _smooth_step(
                sinewave->max_left, sinewave->max_right, tt) );

      col.red   = sinewave->col1.red   * tt + (1.0 - tt) * sinewave->col2.red;
      col.green = sinewave->col1.green * tt + (1.0 - tt) * sinewave->col2.green;
      col.blue  = sinewave->col1.blue  * tt + (1.0 - tt) * sinewave->col2.blue;

      return(col);
}
```

The reflection pixel function

```
fz_rgb_float_td   fz_shdr_cbak_refl_pixel(
      void *shdr_data
      );
```

For a reflection shader, the pixel function is expected to take the pixel color, computed by the color shader and apply shading to it, based on the lighting conditions in the scene. The unshaded pixel color can be retrieved with the API call:

```
      fz_shdr_get_col(&color);
```

If the reflection shader uses any of the standard reflection parameter setup function in the set parameters function, the current value of each parameter needs to the retrieved in the pixel function. Since any of the standard reflection parameters may be altered by a reflection map, the value of a reflection parameter may vary on a surface. Therefore, it cannot be retrieved in the pre render function, stored and passed on to the pixel function via the `shdr_data` parameter. For example, consider the set parameters function of a reflection shader to define the standard diffuse reflection shader:

```
fzrt_error_td     fz_shdr_cbak_refl_set_parameters(void **shdr_data)
{
      ...
      fz_shdr_set_diffuse_parm(0.75);
      ...
}
```

The pixel function of the same reflection shader would retrieve the current value of the diffuse parameter:

```
fz_rgb_float_tdfz_shdr_cbak_refl_pixel(void *shdr_data)
{
      ...
      fz_shdr_get_diffuse_param(&df);
      ...
}
```

`df` will then contain the diffuse factor at the current pixel, taking into account the value of the diffuse factor entered by the user and a possible diffuse map, which will alter the user's value based on the diffuse map's pattern. In addition to obtaining the diffuse factor for a pixel, it is also necessary to perform the actual diffuse illumination. **form•Z** offers API function which perform this task, as well as illumination for ambient, specular, mirror and transmission. Of course, it is up to the plugin developer to implement a custom illumination algorithm, if desired. The illumination function offered by **form•Z** are the same used by the RenderZone display mode. To calculate the diffuse illumination of a pixel the **form•Z** API

```
fz_shdr_get_diffuse_term(&dcol);
```

can be called. The color returned is the illumination from all lights, including shadows. Typically, this color is multiplied (filtered) with the unshaded pixel color, created by the color shader of a surface style to create the final diffuse shaded pixel. The classic shading algorithm computes the final pixel shading using ambient, diffuse and specular illumination with the following algorithm:

```
col_out = col_in * (af * acol + df * dcol) + sf * scol;
```

Where col_in is the unshaded pixel color, `af` is the ambient factor, `acol` is the ambient color (the result of `fz_shdr_get_ambient_term`), `df` is the diffuse factor, `dcol` is the diffuse color (the result of `fz_shdr_get_diffuse_term`), `sf` is the specular factor and `scol` is the specular color (the result of `fz_shdr_get_specular_term`). The full pixel function for such a standard reflection shader would look like this:

```
fz_rgb_float_td fz_shdr_cbak_refl_pixel(void *shdr_data)
{
	double			af,df,sf;
	fz_rgb_float_td	col,acol,dcol,scol;
	refl_data_td		*refl_data;

	refl_data = (refl_data_td*) shdr_data;

	fz_shdr_get_ambient_factor(&af);
	fz_shdr_get_diffuse_factor(&df);
	fz_shdr_get_specular_factor(&sf);

	fz_shdr_get_ambient_term(&acol);
	fz_shdr_get_diffuse_term(&dcol);
	fz_shdr_get_specular_term(refl_data->inv_roughness,&scol);

	fz_shdr_get_col(&col);
	col.red   = col.red   * (af*acol.red   + df*dcol.red)   + sf*scol.red;
	col.green = col.green * (af*acol.green + df*dcol.green) + sf*scol.green;
	col.blue  = col.blue  * (af*acol.blue  + df*dcol.blue)  + sf*scol.blue;

	return(col);
}
```

Note, that the original color is filtered (multiplied) by the ambient and diffuse shading component and the specular color is added on top of it.

Adding raytraced effects.

In addition to the simple shading calculations shown above, it is possible to add reflection effects through raytracing. In the standard reflection shaders offered by **form•Z**, these effects create mirrored and transmission reflections. To add mirrored reflections, a **form•Z** API function can be called:

```
fz_shdr_raytrace_reflected(&world_pt,&mirr_vec,mf,&mirr_col);
```

This function takes the following arguments: `world_pt` is the point where the reflected ray starts on the rendered surface. This point can be retrieved with the API call:

```
fz_shdr_get_world_pnt(&world_pt);
```

which is the point on the surface where the current pixel is rendered. `mirr_vec` is the direction of the reflected ray as it bounces off the surface. For a true mirror surface, this direction is the direction of the view vector, reflected about the normal direction of the surface. The following API functions can be used to calculate this mirror direction:

```
fz_shdr_get_world_shading_normal(&norm);
fz_shdr_get_view_dir(&view_vec);
fz_shdr_ray_reflect(&view_vec,&norm,&mirr_vec);
```

The mirror factor argument `mf` tells the `fz_shdr_raytrace_reflected` API function how much of the calculated mirror color will be added to the final shaded color. If the mirror factor is small, the raytracing can stop earlier, because the added mirror color only makes up a small component of the final pixel color, and it would not make any visible difference to let the raytraced ray bounce longer between other mirroring surfaces. However, if the mirror factor is large, such as in a perfect mirror, the reflected ray needs to bounce longer between other mirroring surfaces to compute accurate reflections. Recall, that the termination of raytraced rays is determined through the options set in the Raytrace Options dialog, which is invoked from the RenderZone Options dialog.

To create transmission effects, which simulate glasslike materials, a similar API function can be called:

```
fz_shdr_raytrace_refracted(&world_pt,&mirr_vec,&tf,&mirr_col);
```

The arguments are the same as to `fz_shdr_raytrace_reflected`. The transmission factor argument `tf`, acts in the same manner as the mirror factor argument. It determines how long refracted rays are allowed to bounce between transmissive and reflective surfaces. In order to calculate the vector with which a refracted ray enters a glass like material, the API function `fz_shdr_ray_refract` can be called. It bends an incoming ray, usually the view direction vector, about the surface normal, using the index of refraction of a material. Thus a complete calculation of a transmission effect can be written like this:

```
if ( tf > 0.0 )
{  fz_shdr_get_world_pnt(&world_pt);
   fz_shdr_get_world_shading_normal(&norm);
   fz_shdr_get_view_dir(&view_vec);

   if(fz_shdr_ray_refract(&view_vec,&norm,refl_data->eta,&mirr_vec) == TRUE)
   {  fz_shdr_raytrace_refracted(&world_pt,&mirr_vec,tf,&mirr_col);
      col.red   += mcol.red   * mf;
      col.green += mcol.green * mf;
      col.blue  += mcol.blue  * mf;

      if ( fz_shdr_ray_inside_solid() == TRUE ) mf = 0.0;
   }
}
```

Note, that the API `fz_shdr_ray_refract` returns a boolean value, which is TRUE, if the incoming ray is bent so that it enters the surface. When the incoming ray is angled in such a way, that with the given index of refraction, it would bounce off the surface rather than enter it, the API return FALSE. In this case no transmission needs to be calculated. Raytracing usually causes a recursive call to the shading pipeline. For example, a ray which is spawned through the call `fz_shdr_ray_reflect` as shown above, may hit another surface. The color of that point on the surface needs to be calculated through the same shader calls as the original surface pixel on the screen. As a result, the same pixel function may be invoked again in a nested fashion. Consider two parallel opposing mirrors. A ray bouncing off one mirror in an exact perpendicular direction would bounce between the two mirror infinitely. **form•Z** will pre-empt this process at a given time, when a satisfactory accuracy of the color to be calculated is achived. It is quite possible, that there may be as many as 10 or more rays, before this occurs. In this case, the pixel function of the mirror reflection shader would be called in a stack of 10 nestings. The same may be the case with `fz_shdr_ray_refract`. A typical glass like material is both refractive and reflective. This means that both raytrace APIs are called. If the ray from a refraction calculation is currently bouncing inside a solid material, such as the wall of a glass bottle, it is only necessary to spawn off another refracted ray when the ray exists the material on the other side. Only when the ray enters the material is it necessary to compute refraction and reflection. In the code example above, the API `fz_shdr_ray_inside_solid()` is called to determine, whether the current ray is inside or outside a solid material. If it is inside, the mirror factor for the subsequent reflection calculation is set to 0.0, effectively disabling mirroring for this ray. Putting all shading components together, a complete reflection shader can be written as shown below. This is actually the code which is used to implement the Generic reflection shader offered by **form•Z**.

```
fz_rgb_float_td fz_shdr_cbak_refl_pixel(void *shdr_data)
{
   double          af,df,sf,mf,tf,gf;
   fz_rgb_float_td col,acol,dcol,scol,mcol,gcol;
   refl_data_td    *refl_data;
   fz_xyz_td       world_pt,norm,view_vec,mirr_vec;

   refl_data = (refl_data_td*) shdr_data;

   fz_shdr_get_col(&col);
   gcol = col;  /* SAVE UNSHADED SURFACE COLOR FOR GLOW LATER */


   /* GET REFLECTION FACTORS */
   fz_shdr_get_ambient_factor(&af);
   fz_shdr_get_diffuse_factor(&df);
   fz_shdr_get_specular_factor(&sf);
   fz_shdr_get_mirror_factor(&mf);
   fz_shdr_get_transmission_factor(&tf);
   fz_shdr_get_glow_factor(&gf);

   /* CALCULATE BASIC SHADING */
   fz_shdr_get_ambient_term(&acol);
   fz_shdr_get_diffuse_term(&dcol);
   fz_shdr_get_specular_term(refl_data->inv_roughness,&scol);

   col.red   = col.red   * (af*acol.red   + df*dcol.red)   + sf*scol.red;
   col.green = col.green * (af*acol.green + df*dcol.green) + sf*scol.green;
   col.blue  = col.blue  * (af*acol.blue  + df*dcol.blue)  + sf*scol.blue;

   /* CALCULATE RAYTRACE EFFECTS */
   if ( mf > 0.0 || tf > 0.0 )
   {  fz_shdr_get_world_pnt(&world_pt);
      fz_shdr_get_world_shading_normal(&norm);
      fz_shdr_get_view_dir(&view_vec);

      /* CALCULATE REFRACTED RAYS */
      if(tf > 0.0 &&
         fz_shdr_ray_refract(&view_vec,&norm,refl_data->eta,&mirr_vec) == TRUE)
      {
         fz_shdr_raytrace_refracted(&world_pt,&mirr_vec,tf,& mcol);
         col.red   += mcol.red   * tf;
         col.green += mcol.green * tf;
         col.blue  += mcol.blue  * tf;

         if ( fz_shdr_ray_inside_solid() == TRUE ) mf = 0.0;
      }

      /* CALCULATE REFLECTED RAYS */
      if ( mf > 0.0 )
      {  fz_shdr_ray_reflect(&view_vec,&norm,&mirr_vec);
         fz_shdr_raytrace_reflected(&world_pt,&mirr_vec,mf,& mcol);
         col.red   += mcol.red   * mf;
         col.green += mcol.green * mf;
         col.blue  += mcol.blue  * mf;
      }
```

```
    }

    /* NOW ADD GLOW, IF ANY */
    if ( gf > 0.0 )
    {
       col.red   += gcol.red   * gf;
       col.green += gcol.green * gf;
       col.blue  += gcol.blue  * gf;
    }

    return(col);
}
```

The transparency pixel function

```
double      fz_shdr_cbak_trns_pixel(
      void* shdr_data
      );
```

The pixel function of a transparency shader is expected to return the level of transparency of the current pixel towards the background. If a value of 0.0 is returned, the pixel is considered completely opaque. If 1.0 is returned, the pixel is considered completely transparent. Values less than 0.0 and larger than 1.0 are not accepted and are clamped to the respective limit. As with a color shader, the transparency shader can compute the pixel transparency based on a pattern. All utility function that can be used by a color shader also apply to a transparency shader. In addition, a transparency shader may compute transparency based on surface geometry. The Neon shader offered by **form•Z** is such a shader. It uses the angle between the surface normal and the view direction to compute the transparency. As such, it is not tagged as a 2d or 3d shader and therefore shows up in the correct section in the Transparency menu in the Surface Style Parameters dialog. The sine wave transparency shader pixel function is shown below:

```
double sinewave_trns_shade_pixel(void *shdr_data)
{
      fz_xy_td                      st;
      sinewave_data_td*       sinewave;
      double                  ss,tt;
      double                  trn;

      sinewave = (sinewave_data_td*) shdr_data;

      shdr_get_tspace_st(&st);

      ss = fz_shdr_saw_tooth(st.x,1.0);
      tt = fz_shdr_saw_tooth(st.y,1.0) + sin(ss * _2PI)*sinewave->ampl;
      tt = fz_shdr_saw_tooth(tt,1.0);

      tt = fz_shdr_smooth_step(
                  sinewave->min_left, sinewave->min_right, tt) *
            (1.0 - fz_shdr_smooth_step(
                  sinewave->max_left, sinewave->max_right, tt));

      trn = sinewave->val1 * tt + (1.0 - tt) * sinewave->val2;

      return(trn);
}
```

The bump pixel function

```
double       fz_shdr_cbak_bump_pixel(
      void  *shdr_data
      );
```

The pixel function of a bump shader is expected to return the bump amplitude (height) of the current pixel. Values should be in the range of 0.0 to 1.0, but may be smaller and larger. The pixel function of a bump shader is actually called more than once per pixel. A number of calls to this function determine how the surface bends around the area of the pixel. This information is then used to alter the normal direction used for the shading calculation during the pixel function of the reflection shader or a surface style. Bump shaders are usually either 2d or 3d and should therefore be tagged as such in the set parameters function. Special care should be taken when writing a bump shader that is based on a pattern. The transition of high and low areas in the pattern should be gradual and smooth for best bump results. For example, the sine wave shader shown below creates a "fuzzy" zone between the wave and background part of the pattern. This is achieved via the fuzz parameter using the `fz_shdr_smooth_step` utility API, which is described in further detail later in this section. If the transition between the wave and the background area would be sharp, the bumps would not be as pronounced, even with a large amplitude parameter. The sine wave bump shader pixel function is shown below:

```
double sinewave_bump_shade_pixel(void *shdr_data)
{
      fz_xy_td                      st;
      sinewave_data_td*       sinewave;
      double                  ss,tt;
      double                  ampl;

      sinewave = (sinewave_data_td*) shdr_data;

      shdr_get_tspace_st(&st);

      ss = fz_shdr_saw_tooth(st.x,1.0);
      tt = fz_shdr_saw_tooth(st.y,1.0) + sin(ss * _2PI)*sinewave->ampl;
      tt = fz_shdr_saw_tooth(tt,1.0);

      tt = fz_shdr_smooth_step(
                  sinewave->min_left, sinewave->min_right, tt) *
            (1.0 - fz_shdr_smooth_step(
                  sinewave->max_left, sinewave->max_right, tt));

      ampl = sinewave->val1 * tt + (1.0 - tt) * sinewave->val2;

      return(ampl);
}
```

Note, that the sine wave transparency and bump shader pixel function are actually identical. If a plugin would register both shaders, just one, generic function could be written and assigned to both callback function sets.

The background pixel function

```
fz_rgb_float_td   fz_shdr_cbak_bgnd_pixel(
      void  *shdr_data
      );
```

The pixel function of a background shader is expected to calculate the color of a pixel in the background of the scene. A background pixel is a part of the image, which is not covered by a surface, or which may be visible through a transparent surface. No tagging as 2d or 3d is necessary for this shader in the set parameters function. The coordinate of the current background pixel can be retrieved with the API call:

```
	fz_shdr_get_ispace_xy(&bg_pixel);
```

The coordinate for the upper left corner of the pixel would be x = 0.0, y = 0.0, the lower right corner is x = 1.0, y = 1.0 regardless of the image pixel resolution. The sine wave background shader pixel function is shown below:

```
fz_rgb_float_td sinewave_bgnd_shade_pixel(void *shdr_data)
{
	fz_xy_td			st;
	sinewave_data_td*	sinewave;
	double			ss,tt;
	fz_rgb_float_td		col;

	sinewave = (sinewave_data_td*) shdr_data;

	fz_shdr_get_ispace_xy(&st);

	ss = fz_shdr_saw_tooth(st.x,1.0);
	tt = fz_shdr_saw_tooth(st.y,1.0) + sin(ss * _2PI)*sinewave->ampl;
	tt = fz_shdr_saw_tooth(tt,1.0);

	tt = fz_shdr_smooth_step(
			sinewave->min_left, sinewave->min_right, tt) *
		(1.0 - fz_shdr_smooth_step(
			sinewave->max_left, sinewave->max_right, tt) );

	col.red   = sinewave->col1.red   * tt + (1.0 - tt) * sinewave->col2.red;
	col.green = sinewave->col1.green * tt + (1.0 - tt) * sinewave->col2.green;
	col.blue  = sinewave->col1.blue  * tt + (1.0 - tt) * sinewave->col2.blue;

	return(col);
}
```

Note, that this function is the same as the pixel function of the sine wave color shader, with the exception of the API call to get the pixel coordinate. The color pixel function uses `fz_shdr_get_tspace_xy(&st);` to get the texture space coordinate, whereas the background pixel function uses `fz_shdr_get_ispace_xy(&st);` to get the image space coordinate. Similar to the color shader pixel function, the standard scale factor is already contained in the image space coordinate.

The depth effect (foreground) pixel function

```
fz_rgb_float_td	fz_shdr_cbak_fgnd_pixel(
	void	*shdr_data
	);
```

The pixel function of a depth effect shader is expected to change the color of a pixel based on the depth of the surface pixel in the scene. The depth effect shader is the last shader invoked in the shader pipeline. The API function `fz_shdr_get_dist_eye_world_pnt` can be called to get the distance of the pixel's world coordinate point to the eye point. If the current pixel is a background pixel, the API function will return FALSE. In this case, there is no surface to be rendered at that pixel. An example of a simple depth effect shader, that adds a constant color to a pixel based on its distance between the eye point and the yon view clipping plane is shown below:

```
fz_rgb_float_td sample_fgnd_shade_pixel(void *shdr_data)
{
	sample_data_td*		sample_data;
	fz_rgb_float_td		col;
	double			dist,ratio,inv_ratio;

	sample_data = (sample_data_td *) shdr_data;

	fz_shdr_get_col(&col);
	if( fz_shdr_get_dist_eye_world_pnt(&dist) == TRUE )
	{

		ratio = dist / sample_data->yon;
		if ( ratio > 1.0 ) ratio = 1.0;

		inv_ratio = 1.0 - ratio;

		col.red   = col.red   * inv_ratio + sample_data->col.red   * ratio;
		col.green = col.green * inv_ratio + sample_data->col.green * ratio;
		col.blue  = col.blue  * inv_ratio + sample_data->col.blue  * ratio;
	}

	return(col);
}
```

The post render function (recommended)

```
fzrt_error_td fz_shdr_cbak_colr_post_render(
	void  *shdr_data
	);
```

This function is called once at the end of each rendering. It is expected to perform any tasks necessary when the shader is done rendering the image. Usually this means, that the data allocated in the pre render function is deallocated in the post_render function. The sine wave shader post render function is shown below:

```
fzrt_error_td sinewave_post_render(void *shdr_data)
{
	if (shdr_data)
	{	fz_mem_zone_free(NULL,(fzrt_ptr*)&shdr_data);
	}
	return(FZRT_NOERR);
}
```

Shader utiltity functions

There are a number of additional API function in the function set `fz_shdr_fset`, which are intended to facilitate the implementation of a shader plugin. The most important of these APIs are described in more detail below.

Repeating patterns

If a pattern is regular and repeats in a tile like fashion, such as bricks or checkers, the values of the texture coordinate need to be modulated. This can be done with the API call:

```
s = fz_shdr_saw_tooth(st.x,1.0);
t = fz_shdr_saw_tooth(st.t,1.0);
```

This guarantees, that the incoming values st.x and st.y, for example, oscillate between 0.0 and 1.0. The pattern algorithm then only needs to consider values in that range. In the Sine Wave shader, for example, the y component of the 2d texture coordinate is modified with `fz_shdr_saw_tooth`. This will yield one sine curve for each texture tile, instead of just one sine curve in the whole texture space. The saw tooth function can also be described through this simple algorithem:

```
if ( val_in < 0.0 ) val_out = -fmod(val_in,module);
else                val_out =  fmod(val_in,module);
```

Random Patterns

form•Z offers a number of utility functions, which compute a random pattern based on a single value, a 2d coordinate or a 3d coordinate. They are

```
fz_shdr_turbulance_1d
fz_shdr_turbulance_2d
fz_shdr_turbulance_3d
fz_shdr_noise_1d
fz_shdr_noise_2d
fz_shdr_noise_3d
```

The turbulance and noise functions are very similar. The turbulance functions take an additional integer parameter, which creates more detail if passed in with a higher value. The input to the noise and turbulance functions is usually a value of the texture space coordinate of the pixel to be rendered. The function returns a pseudo random number between 0.0 and 1.0. This number can be used to design a pattern. For example, the code below creates a random pattern of black dots on a white background:

```
fz_shdr_get_tspace_st(&st);

val = fz_shdr_noise_2d(&st,FZ_SHDR_TURB_TYPE_BETTER,0);

if ( val < 0.5 )  col = black_color;
else              col = white_color;
```

It is up to the creativity of the shader developer to use noise and turbulance functions to break up regular patterns and to create unique pattern designs. In **form•Z** these functions are used in a number of shaders. For example, the Textured Brick shader uses noise functions to mix two brick colors and also to break up the straight line of the mortar edges. The Textured Marble color shader uses turbulance functions to mix the marble colors.

Smooth transitions

It is often desirable to create a soft transition between two colors in a pattern. In **form•Z** shaders, this softening of contrast is called fuzz and offered in many shaders. Not only can it be used to create different variations of the shader pattern, but it also help to avoid aliasing artifacts. A API utility function is availabe to compute smooth transitions:

```
val_out = fz_shdr_smooth_step(min,max,val_in);
```

If the val parameter is less than min fz_shdr_smooth_step will return 0.0. If the val parameter is greater than max fz_shdr_smooth_step will return 1.0. If the val parameter is between min and max, fz_shdr_smooth_step will return a value between 0.0 and 1.0. However, the value is not a linear interpolation, When plotted as a function graph, the curve resembles a leaning S, connecting y = 0.0 and y = 1.0 in a smooth fashion. This function can be used to create fuzz along edges of sharp contrast in a pattern.

For example consider a simple pattern of horizontal stripes:

```
fz_shdr_get_tspace_st(&st);
st.y = fz_shdr_saw_tooth(st.y,1.0);
if ( st.y < 0.5 )        col = black;
else                     col = white;
```

This will create a crips border between the black and white color. To create a fuzzy border, fz_shdr_smooth_step can be used:

```
fz_shdr_get_tspace_st(&st);
st.y = fz_shdr_saw_tooth(st.y,1.0);
val = fz_shdr_smooth_step(0.4,0.6,st.y);
col = val * white + (1.0 - val) * black;
```

If st.y is less than 0.4 fz_shdr_smooth_step returns 0.0 and the color computation yields :

```
col = 0.0 * white + (1.0 - 0.0) * black;
```

which is all black. If st.y is greater than 0.6 fz_shdr_smooth_step returns 1.0 and the color computation yields :

```
col = 1.0 * white + (1.0 - 1.0) * black;
```

which is all white. In the zone where st.y is between 0.4 and 0.6 black and white are mixed. More black is used as st.y approaces 0.4 and more white is used as it approaches 0.6.This creates a smooth color transition from black to white.

Natrually, the smooth step function is not limited to the context of blending colors. It is just as useful to create smooth transitions between opaque and transparent areas in a transparency shader and between high and low areas in a bump shader.

Another method to create smooth transitions is the API

```
fz_shdr_spline_color(val,ncolors,colors,&color_out);
```

It computes a smoothly blended color from a list of individual colors. The first argument is a parametric value that must be in the range of 0.0 to 1.0. For example, if there are four colors, and the val argument is below 0.25, the first color is returned. If val is around 0.25, a mixture between the first and second color is returned. If it is between 0.25 and 0.5 the second color is returned, etc. This function can be combined with a turbulence function to create a pattern of random colored spots.

```
fz_shdr_get_tspace_st(&st);
val = fz_shdr_turbulance_2d(&st,3,FZ_SHDR_TURB_TYPE_BETTER,0);
fz_shdr_spline_color(val,5,colors_in,&color_out);
```

2.8.8 Tool Plugins

Tool plugins are extensions that complement the form•Z tool set and behave consistent with the form•Z tools. They appear in the form•Z interface in the icon tool palettes just like a form•Z tool. Tools can either be ***operators*** or ***modifiers***. An operator creates or edits the form•Z project data (objects, lights, etc.) though graphic manipulation in the form•Z Project window. A modifier is a tool that controls a setting that affects a group of operators. For example, the self/copy modifier tools affect how the transformation operator tools function. Modifiers are never implemented as a single tool but rather a set of tools that have a number of modifiers representing different options and a set of operators that are sensitive to the selected modifier.

The user selects a tool from a tool icon menu or via a key shortcut to make it the active tool. A click (or multiple clicks) in the project window or input in the prompt palette is used to execute the tool. Tools are dependent on a project window and are expected to function on the provided project window. Tools are unavailable when there is no open project window.

Tools my have user controlled options associated with them. These options appear in the tool options palette when the tool is active. The options can also be accessed in a dialog that is invoked by double clicking on the tool's icon or by ***right***-clicking on the tool's icon. The dialog can also be invoked by pressing ***option*** (Macintosh) or ***ctrl+shift*** (Windows) while clicking on the tool's icon.

Tools are very flexible and can do a variety of things. Object ***creation***, ***editing*** and ***derivation*** operations are common uses of tools. In an object creation tool, input from the user in the form of clicks and/or prompt entry is used to construct an object. To create an interactive tool, a base object should be constructed as early in the tool as possible and then refined as additional input is acquired.

An editing operation modifies existing objects. A derivative operation uses existing objects as a starting point to create new objects. Both of these operations need to execute pick operations to select the objects (or other topological levels) to operate on. The tools should support the prepick and postpick model that is standard in **form•Z**.

The graphic image of the icon is supplied by the plugin via one of the standard form•Z bitmap image formats (TIFF, Targa, PNG, BMP, JPEG). If one is not provided, a default plugin icon is supplied by form•Z. The plugin can also specify where in the tool palette the icon for the tool is positioned. If a position is not provided, then the tool is placed at the bottom of the tool palette. The icons for tool plugins appear at the bottom of the **Tool Set** in the **Icons Customization** dialog. It can be customized as with any form•Z tool. All tools appear in the **Key Shortcuts Manager** dialog so that they may have key shortcuts assigned for them.

The Samples directory in the form•Z SDK folder contains a folder named Tools that contains a number of examples of tool plugins. These can be very valuable as both starting points for development as well as examples of how the functions work. The samples include the following plugins:

- **Triangle**: Creates a tool in the tool palette to interactively draw a triangle.
- **Star**: Creates a control object definition for a star object and a tool in the tool palette to interactively draw a star.
- **Frame**: Creates a control object definition for a derivative frame object (tube derived from an existing object) and a tool in the tool palette to execute the operation.

Tool plugin type and registration.

Tool plugins are identified with the plugin type of `FZ_TOOL_EXTS_TYPE` and must implement the `fz_tool_cbak_fset` call back function set. The following shows the registration of a tool and a call back implementation. This is done from the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3. Tool plugins may provide the `fz_notf_cbak_fset` function set to be notified when changes occur within **form•Z**.

```
fzrt_error_td my_tool_register_plugins()
{
	fzrt_error_td		err = FZRT_NOERR;
	char			my_name[256];

	/* Get the title string "My Tool" from the plugin's resource file */
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_name)) ==
FZRT_NOERR)
	{
		/* register the plugin "My Tool"  */
		err = fzpl_glue->fzpl_plugin_register(
			MY_PLUGIN_UUID,
			my_name,
			MY_PLUGIN_VERSION,
			MY_PLUGIN_VENDOR,
			MY_PLUGIN_URL,
			FZ_TOOL_EXTS_TYPE,
			FZ_TOOL_EXTS_VERSION,
			my_plugin_error_string_func,
			0,
			NULL,
			&my_plugin_runtime_ID);

		if ( err == FZRT_NOERR )
		{
			/* add the function set for the tool */
			err = fzpl_glue->fzpl_plugin_add_fset(
				my_plugin_runtime_id,
				FZ_TOOL_CBAK_FSET_TYPE,
				FZ_TOOL_CBAK_FSET_VERSION,
				FZ_TOOL_CBAK_FSET_NAME,
				FZPL_TYPE_STRING(fz_tool_cbak_fset),
				sizeof ( fz_tool_cbak_fset ),
				my_tool_cbak_fill_fset,
				FALSE);
		}
	}
	return (err);
}
```

Tool call back function set.

Tool plugins are implemented by the call back function set `fz_tool_cbak_fset`. There are twenty-four functions in this function set. The following example shows the assignment of the plugins defined functions into the function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above. Note that some of these functions are optional and some are mutually exclusive hence a plugin would never implement all of these functions. Each of these functions is described in the following sections.

```
fzrt_error_td my_tool_cbak_fill_fset (
		const fzpl_fset_def_ptr fset_def,
		fzpl_fset_td * const fset )
{
	fzrt_error_td		err = FZRT_NOERR;
```

```
    fz_tool_cbak_fset          *tool_fset

    /* check that the provided function set is of the expected version */
    err = fsf->fzpl_fset_def_check ( fset_def,
            FZ_TOOL_CBAK_FSET_VERSION,
            FZPL_TYPE_STRING(fz_tool_cbak_fset),
            sizeof ( fz_tool_cbak_fset ),
            FZPL_VERSION_OP_NEWER );

    if ( err == FZRT_NOERR )
    {
            /* fill function set structure with local plugins functions */
            tool_fset = (fz_tool_cbak_fset *)fset;

            tool_fset->fz_tool_cbak_init                    = my_tool_init;
            tool_fset->fz_tool_cbak_finit                   = my_tool_finit;
            tool_fset->fz_tool_cbak_info                    = my_tool_info;
            tool_fset->fz_tool_cbak_name                    = my_tool_name;
            tool_fset->fz_tool_cbak_uuid                    = my_tool_uuid;
            tool_fset->fz_tool_cbak_help                    = my_tool_help;

            tool_fset->fz_tool_cbak_avail                   = my_tool_avail;
            tool_fset->fz_tool_cbak_active                  = my_tool_active;
            tool_fset->fz_tool_cbak_select                  = my_tool_select;

            tool_fset->fz_tool_cbak_click                   = my_tool_click;
            tool_fset->fz_tool_cbak_track                   = my_tool_track;
            tool_fset->fz_tool_cbak_prompt                  = my_tool_prompt;
            tool_fset->fz_tool_cbak_undo                    = my_tool_undo;
            tool_fset->fz_tool_cbak_redo;                   = my_tool_redo;
            tool_fset->fz_tool_cbak_cancel                  = my_tool_cancel;
            tool_fset->fz_tool_cbak_icon_menu               = my_tool_icon_menu;
            tool_fset->fz_tool_cbak_icon_menu_adjacent      = my_tool_icon_menu_adjacent;
            tool_fset->fz_tool_cbak_icon_rsrc               = my_tool_icon_rsrc;
            tool_fset->fz_tool_cbak_icon_file               = my_tool_icon_file;

            tool_fset->fz_tool_cbak_pref_io                 = my_tool_pref_io;

            tool_fset->fz_tool_cbak_opts_name               = my_tool_opts_name;
            tool_fset->fz_tool_cbak_opts_uuid               = my_tool_opts_uuid;
            tool_fset->fz_tool_cbak_opts_help               = my_tool_opts_help;
            tool_fset->fz_tool_cbak_opts_iface_tmpl         = my_tool_opts_iface_tmpl;
    }

    return err;
}
```

The tool initialization function (optional)

```
fzrt_error_td fz_tool_cbak_init(
    void
    );
```

This function is called by **form•Z** once when the plugin is successfully loaded and registered. The initialization function is where the plugin should initialize any data that may be needed by the other functions in the function set. If the tool is an editing operation which creates new objects from selected objects, the status of objects options for the tool needs to be initialized by calling `fz_sys_cmnd_set_status_of_objt` in the tool's initialization funtion.

```
fzrt_error_td my_tool_init(
       void
       );
{
       fzrt_error_td       err = FZRT_NOERR;

       /** Do initialization here **/

       return(err);
}
```

The tool finalization function (optional)

```
fzrt_error_td fz_tool_cbak_finit(
       void
       );
```

This function is called by **form•Z** once when the plugin is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to free any memory that was allocated in the initialization function or during the life of the tool.

```
fzrt_error_td my_tool_finit(
       void
       );
{
       fzrt_error_td       err = FZRT_NOERR;

       /** Free any initialized data here **/

       return(err);
}
```

The tool info function (required)

```
fzrt_error_td fz_tool_cbak_info(
       fz_tool_kind_enum   *kind
       fz_proj_level_enum  *level
       );
```

This function is called by **form•Z** once when the plugin is successfully loaded to determine the kind and level of the tool that is implemented by the function set. The `kind` parameter indicates if the tool is an operator (`FZ_TOOL_KIND_OPERATOR`) or a modifier (`FZ_TOOL_KIND_MODIFIER`). form•Z uses the value in this parameter to determine how the icons are handled when they are selected by the user.

The `level` parameter indicates the context of the tool. form•Z uses the value in this parameter to determine which tool palette to add the icon for the tool plugin. The following are the available values:

`FZ_PROJ_LEVEL_MODEL`: Indicates that the tool operates on the projects modeling content (objects for example).
`FZ_PROJ_LEVEL_MODEL_WIND`: Indicates that the tool operates on modeling window specific content (views for example) of modeling windows.
`FZ_PROJ_LEVEL_DRAFT`: Indicates that the tool operates on the projects drafting content (elements for example).
`FZ_PROJ_LEVEL_DRAFT_WIND`: Indicates that the tool operates on drafting window specific content (views for example) of drafting windows.

.

```
fzrt_error_td my_tool_cbak_info(
       fz_tool_kind_enum   *kind
       fz_proj_level_enum  *level
       );
{
       fzrt_error_td       err = FZRT_NOERR;

       /* set kind and level for the tool */
       *kind  = FZ_TOOL_KIND_OPERATOR;
       *level = FZ_PROJ_LEVL_MODEL;

       return(err);
}
```

The tool name function (recommended)

```
fzrt_error_td fz_tool_cbak_name(
       char          *name,
       long          max_len
       );
```

This function is called by **form•Z** to get the name of the tool. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the tool name string is stored in a .fzr file so that it is localizable. This function is recommended for all tool plugins. If this function is not provided , the name of the plugin provided in the `fzpl_plugin_register` function is used.

```
fzrt_error_td my_tool_name(
       char          *name,
       long          max_len
       )
{
       fzrt_error_td       err = FZRT_NOERR;
       char                my_str[256];

       /* Get the title string "My Tool" from the plugin's resource file */
       if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_str)) ==
FZRT_NOERR)
       {
              /* copy the string to the name parameter */
              strncpy(name,  my_str, max_len);
       }
       return(err);
}
```

The tool uuid function (recommended)

```
fzrt_error_td fz_tool_cbak_uuid
       fzrt_UUID_td  uuid
       );
```

This function is called by **form•Z** to get the UUID of the tool. This unique ID is used by **form•Z** to distinguish the tool from other tools. This function is recommended for all tool plugins. If a UUID is not provided, one will be generated internally by **form•Z**. In this situation the UUID will not be the same each time **form•Z** is run and hence persistent information will not be retained. This includes any preference information provided by a supplied `fz_tool_cbak_pref_io` function or any user customization like key shortcuts and tool icon layout.

```
#define MY_TOOL_ID
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td  my_tool_uuid(
       fzrt_UUID_td uuid
       )
{
       fzrt_error_td       err = FZRT_NOERR;

       /* copy constant UUID to into the uuid parameter */
       fzrt_UUID_copy(MY_TOOL_ID, uuid);

       return(err);
}
```

The tool help function (optional)

```
fzrt_error_td fz_tool_cbak_help(
       char          *help,
       long          max_len
       );
```

This function is called by **form•Z** to display a help string that describes the detail of what the tool does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` bytes of data. It is recommended that the tool name is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 512 bytes (characters).

```
fzrt_error_td my_tool_help(
       char          *help,
       long          max_len
       )

{
       fzrt_error_td       err = FZRT_NOERR;
       char                my_str[512];

       /* Get the help string from the plugin's resource file */
       if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) ==
FZRT_NOERR)
       {
              /* copy the string to the help parameter */
              strncpy(help,  my_str, max_len);
       }
       return(err);
}
```

The tool available function (optional)

```
fzrt_error_td fz_tool_cbak_avail(
       long                windex,
       long                *rv
       );
```

This function is called by **form•Z** at various times to see if the tool is available. This is useful if the tool is dependent on certain conditions and it is desirable to restrict its use when the conditions are not currently satisfied. If the tool is not available, then it is shown as inactive (dimmed) in the **form•Z** tool palette. Key shortcuts are also disabled for the tool when it is not available. If this function is not provided then the tool is always available.

Availability is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the tool is available, a value of 0 indicates that the tool is unavailable.

```
fzrt_error_td my_tool_avail(
	long				windex,
	long				*rv
	);
{
	fzrt_error_td		err = FZRT_NOERR;

	/* return 1 for available, 0 for not available */
	*rv = 1;

	return(err);
}
```

The tool active function (required for modifiers, not used for operators)

```
fzrt_error_td fz_tool_cbak_active(
	long				windex,
	long				*rv
	);
```

This function is called by **form•Z** at various times to see if the modifier tool is active. This is used by form•Z to draw the icon in the selected state. The value that is returned by the `rv` parameter determines if the tool is active or not. A value of 1 indicates that the tool is active, a value of 0 indicates that the tool is inactive.

```
fzrt_error_td my_tool_active(
	long				windex,
	long				*rv
	);
{
	fzrt_error_td		err = FZRT_NOERR;

	/* return 1 for active, 0 for not active */
	if(my_modifier_state ==2)*rv = 1;
	else *rv = 0;

	return(err);
}
```

The tool select function (optional)

```
fzrt_error_td fz_tool_cbak_select(
	long				windex
	);
```

This function is called by **form•Z** when the tool is selected from the tool icon palette or when a key shortcut for the tool is invoked.

For operator tools, the select function is where any tool specific preparation occurs for the execution of the tool (which is triggered by a click in the project window). The select function should set the prompt string (in the prompts palette) for the tool. The select function is also called after the execution of the tool to prepare it for the next execution.

The following example shows the select function for an operator tool that draws a line. It starts by asking for the origin point for an object in the prompts palette. Note the prompt string is shown here for readability. It should be stored in a .fzr resource file and loaded with `fzrt_get_string` to support localization.

```
fzrt_error_td my_tool_select(
	long		windex
	)
{
char				prompt_str[256];
short				pre_pick;
long				i,npick;
fz_model_pick_enum		pkind;'
fzrt_error_td			err = FZRT_NOERR;

	/* Get the prompt string "First point:" from the plugin's resource file
*/
	if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 3, prompt_str)) ==
FZRT_NOERR)
	{
		err = fz_fuim_prompt_line(
			prompt_str,			/* prompt string */
			FZ_FUIM_PROMPT_LINE_NEXT,	/* place it on the next line */
			FZ_FUIM_PROMPT_EDIT_XYZ);	/* set the edit mode of prompt */
	}
	return(err);
}
```

The following example shows the select function for a tool that starts by asking the user to select an object. Note that the prompt handles prepick and postpick by checking the state of the pick buffer.

```
fzrt_error_td my_tool_select(
	long		windex
	)
{
char				prompt_str[256];
short				pre_pick;
long				i,npick;
fz_model_pick_enum		pkind;
fzrt_error_td			err = FZRT_NOERR;

	/* Get the number of picked entities */
	fz_model_pick_get_count(windex,&npick);

	/* loop through picked entities */
	for(i = 0; i < npick; i++)
	{
		/* get one picked entity */
		fz_model_pick_get_data(windex,i,&pkind,NULL,NULL,NULL);

		/* check if it was picked at the object level */
		if ( pkind == FZ_MODEL_PICK_OBJT )
		{	pre_pick = TRUE;
			break;
		}
	}

	/* check if it was picked at the object level */
	if(pre_pick)
	{	/* Get the string "Click to frame selected objects" */
		err = fzrt_fzr_get_string(my_rfzr_refid, 1, 4, prompt_str);
	}
	else
	{	/* Get the string "Select object to frame" */
```

```
        err = fzrt_fzr_get_string(my_rfzr_refid, 1, 5, prompt_str);
    }

    err = fz_fuim_prompt_line(
        prompt_str,                 /* prompt string */
        FZ_FUIM_PROMPT_LINE_NEXT,   /* place it on the next line */
        FZ_FUIM_PROMPT_EDIT_NONE);        /* set the edit mode of prompt to
    none */


    return(err);
}
```

For modifier tools, the select function should change the state of the modifier to the desired value for the selected icon. The modifier is usually a global variable in the plugin that can be accessed by the tools that use it.

```
fzrt_error_td my_tool_select(
    long                windex
    )
{
fzrt_error_td       err = FZRT_NOERR;

    /* Set modifier state for the tool */
    my_modifier_state = 2;
}
```

The tool click function (required for operators, not used for modifiers)

```
fzrt_error_td fz_tool_cbak_click
    long                            windex,
    fzrt_point                      *where,
    fz_xyz_td                       *where_3d,
    fz_map_plane_td                 *map_plane,
    fz_fuim_click_enum              clicks,
    long                            click_count,
    fzrt_boolean                    *click_handled,
    fz_fuim_click_wait_enum         *click_wait,
    fzrt_boolean                    *done
    );
```

This function is called by **form•Z** for operators when the tool is the active tool and a click occurs in the active project window. This function is called by **form•Z** for each click in the project window until TRUE is returned in the `done` parameter (or from the `fz_tool_cbak_prompt` function) or the user cancels the operation.

The `windex` parameter is the active window. The `where` parameter indicates in 2 dimensional screen space where the mouse was clicked. The `where_3d` parameter indicates the 3 dimensional location in world space where the mouse was clicked. This is a point on the active reference plane provided in the `map_plane` parameter. The `clicks` parameter indicates if the click is a single, double or triple click. The `click_count` parameter is the number of clicks since the start of the tool. This value starts at 1 for the first click and increases with each click of the mouse.

The `click_handled` parameter should be set to TRUE if the click function handled the click and it should be set to FALSE if the function did not handle the click. The default value is TRUE. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the click function again. The default is FZ_FUIM_CLICK_WAIT_NOT. The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

The following example shows the click function for a tool that draws a line. The first click creates a new object with a single segment (edge) with identical start and end points at the click point. The second click fixes the end point at the click point. This is done in this manner to accommodate the track function (see following section). If a track function is not provided then the object does not need to be created until the final click. In this situation, the click points could be accumulated into a buffer and then used to create the object. Note that this is not an ideal interface for the user as they will get no interactive feedback during the operation. If performance is a concern because of the complexity of the operation, then a proxy should be used so that the user gets some feedback during the tools execution.

```
typedef struct
{	fz_objt_ptr		obj;
	fz_xyz_td		points[3];
} line_data_td;

line_data_td  line_data;

fzrt_error_td my_tool_click(
	long				windex,
	fzrt_point			*where,
	fz_xyz_td			*where_3d,
	fz_map_plane_td			*map_plane,
	fz_fuim_click_enum		clicks,
	long				click_count,
	fzrt_boolean			*click_handled,
	fz_fuim_click_wait_enum		*click_wait,
	fzrt_boolean			*done
	)
{
	fzrt_error_td		err;
	char			prompt_str[256];
	long			pindx[2];

	if(click_count == 1)				/* handle first click */
	{
		/* make new object */
		if((err = fz_objt_cnstr_objt_new(windex,&line_data.obj)) ==
FZRT_NOERR )
		{
			/* construct line object */
			line_data.points[0] = *where_3d;
			line_data.points[1] = *where_3d;
			fz_objt_fact_add_pnts(windex,obj,line_data.points,2);

			pindx[0] = 0;
			pindx[1] = 1;
			fz_objt_fact_create_wire_face(
				windex, line_data.obj,pindx,2,NULL);

			/* add object to the project */
			err = fz_objt_add_to_project(windex,line_data.obj);

			if ( err != FZRT_NOERR )
			{	fz_objt_edit_delete_objt(windex,line_data.obj);
			}
			else
			{
				/* Get the string "Second point:" */
```

```
                        err = fzrt_fzr_get_string(my_rfzr_refid, 1, 6,
prompt_str);

                        /* set prompt for next point */
                        fz_fuim_prompt_line(prompt_str,
                              FZ_FUIM_PROMPT_LINE_NEXT,
                              FZ_FUIM_PROMPT_EDIT_XYZ);
                  }
            }
      }
      else if(click_count == 2)              /* handle second click */
      {
            /* reset object and construct with new second point */
            fz_objt_fact_reset(windex, line_data.obj);
            line_data.points[1] = *where_3d;
            fz_objt_fact_add_pnts(windex, line_data.obj,line_data.points,2);

            pindx[0] = 0;
            pindx[1] = 1;
            fz_objt_fact_create_wire_face(windex, line_data.obj,pindx,2,NULL);

            *done = 1;                       /* tool complete */
      }
}
```

If the operation requires the picking (selection) of objects (or other topological levels), then this should be handled following the **form•Z** prepick and postpick standard. That is for each click the pick buffer is inspected to see if the requirements have been satisfied for the operation (prepick). If it is not satisfied, the function fz_model_pick is called to handle the click as a postpick and then the pick buffer is re-inspected. If the pick requirements have been satisfied with the prepick or postpick then the operation completes. The prompts palette should also be updated in the click function to reflect the desired user actions using the `fz_fuim_prompt_line` function.

```
fzrt_error_td my_tool_click(
      long                          windex,
      fzrt_point                    *where,
      fz_xyz_td                     *where_3d,
      fz_map_plane_td               *map_plane,
      fz_fuim_click_enum            clicks,
      long                          click_count,
      fzrt_boolean                  *click_handled,
      fz_fuim_click_wait_enum       *click_wait,
      fzrt_boolean                  *done
      );
{
      fzrt_error_td       err = FZRT_NOERR;

      *done = FALSE;

      /* Get the number of picked entities */
      fz_model_pick_get_count(windex,&npick);
      if(npick < 2)
      {
            /* use the click to pick an object */
            fz_model_pick(windex,where,FZ_MODEL_PICK_OBJT);
            fz_model_pick_get_count(windex,&npick);
      }

      /* check if enough picked to execute operation */
      if(npick >= 2)
      {
            /* get first two objects from pick buffer */
            fz_model_pick_get_data(windex,0,&pkind1,NULL,&pick_obj1,NULL);
```

```
            fz_model_pick_get_data(windex,1,&pkind2,NULL,&pick_obj2,NULL);
            if(pkind1 == FZ_MODEL_PICK_OBJT && pkind2 == FZ_MODEL_PICK_OBJT)
            {
                  /** operate on objects here **/
            }

            *done = TRUE;
      }

      return(err);
}
```

If the tool is an editing operation which creates new objects from selected objects, the status of objects functionality should be implemented. This can be done easily with two api calls: `fz_objt_edit_handle_status_of_opnd` and `fz_objt_edit_handle_new_objt_volms`. These two functions correspond directly to the options in the Status Of Objects palette. Note that the tool also needs to initialize its status of objects option in the `fz_tool_cbak_init` callback function by calling `fz_syst_cmnd_set_status_of_objt` with the appropriate arguments.

The tool prompt function (required for operators, not used for modifiers)

```
fzrt_error_td tool_cbak_prompt
      long                              windex,
      fz_xyz_td                         *prompt_value,
      char                              *prompt_string,
      fz_map_plane_td                   *map_plane,
      long                              click_count,
      fzrt_boolean                      *prompt_handled,
      fz_fuim_click_wait_enum           *click_wait,
      fzrt_boolean                      *done
      );
```

This function is called by **form•Z** when the tool is the active tool and the user makes input in an editable prompt string in the prompts palette. This function is very similar to the click function and each input of data in the prompts palette is treated by **form•Z** the same as a click. This function is called by **form•Z** each time the user enters data in the prompts palette and then presses the enter or return keys. Like the click function, this function is called until TRUE is returned in the `done` parameter (or TRUE is returned in the `done` parameter from the click function) or the user cancels the operation.

The windex parameter is the active window. The `prompt_value` and `prompt_string` parameters are the users input from the prompts palette. An editable prompt is created by calls to the `fz_fuim_prompt_line` function in the select function, click function, undo function, redo function or previous click handling in the prompt function. Editable input is specified by the last parameter to the `fz_fuim_prompt_line` function. This parameter instructs the prompts palette as to what type of input is desired (if any). The following table shows the available options.

Name	Description
FZ_FUIM_PROMPT_EDIT_NONE	No editable text in prompt string
FZ_FUIM_PROMPT_EDIT_XY	Standard 2D dimensional world Cartesian coordinate
FZ_FUIM_PROMPT_EDIT_XYZ	Standard 3D dimensional world Cartesian coordinate
FZ_FUIM_PROMPT_EDIT_ANGLE	Angular dimension
FZ_FUIM_PROMPT_EDIT_LINEAR_X	Liner dimension
FZ_FUIM_PROMPT_EDIT_LINEAR_XY	Liner 2D dimension
FZ_FUIM_PROMPT_EDIT_LINEAR_XYZ	Liner 3D dimension
FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_X	Liner dimension, displayed in decimal format.

FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XY	Liner 2D dimension, displayed in decimal format.
FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XYZ	Liner 3D dimension, displayed in decimal format.
FZ_FUIM_PROMPT_EDIT_STRING	string

Note that the FZ_FUIM_PROMPT_EDIT_STRING does not return a value for the prompt_value parameter. Instead the raw string is returned in the prompt_string parameter. The prompt_value parameter is interpreted based on the type of the prompt edit shown in the above table. If the prompt edit is FZ_FUIM_PROMPT_EDIT_ANGLE, FZ_FUIM_PROMPT_EDIT_LINEAR_X, or FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_X, then the value is found in the first field (x). If the prompt edit is FZ_FUIM_PROMPT_EDIT_XY, FZ_FUIM_PROMPT_EDIT_LINEAR_XY, or FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XY, then the values are found in the first two fields (x and y). If the prompt edit is FZ_FUIM_PROMPT_EDIT_XYZ, FZ_FUIM_PROMPT_EDIT_LINEAR_XYZ, or FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XYZ, then the values are found all three fields (x, y and z).

The map_plane parameter is the active reference plane. The click_count parameter is the number of clicks (or prompts) since the start of the tool. This value starts at 1 for the first click (or prompt) and increases with each click (or prompt).

The prompt_handled parameter should be set to TRUE if the prompt function handled the prompt and it should be set to FALSE if the function did not handle the prompt. The default value is TRUE. The click_wait parameter tells **form•Z** to wait until a specific type of click happens before calling the next click function. The default is FZ_FUIM_CLICK_WAIT_NOT. The done parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

The following example shows the prompt function for a tool that draws a line. The prompt function is very similar to the click function in the previous line tool example. In the prompt function the coordinate location comes from the prompt_value parameter rather than the click point.

```
typedef struct
{       fz_objt_ptr                 obj;
        fz_xyz_td                   points[3];
} line_data_td;

line_data_td  line_data;

fzrt_error_td my_tool_prompt        (
        long                                windex,
        fz_xyz_td                           *prompt_value,
        char                                *prompt_string,
        fz_map_plane_td                     *map_plane,
        long                                click_count,
        fzrt_boolean                        *prompt_handled,
        fz_fuim_click_wait_enum             *click_wait,
        fzrt_boolean                        *done
        )
{
        fzrt_error_td         err = FZRT_NOERR;
        char                  prompt_str[256];
        long                  pindx[2];
        fz_xyz_td             xyz;

        /* Get the prompt data */
```

```
	xyz = *prompt_value;

	if(click_count == 1)				/* handle first click */
	{
		/* make new object */
		if((err = fz_objt_cnstr_objt_new(windex,&line_data.obj)) ==
FZRT_NOERR )
		{
			/* construct line object */
			line_data.points[0] = xyz;
			line_data.points[1] = xyz;
			fz_objt_fact_add_pnts(windex,
line_data.obj,line_data.points,2);

			pindx[0] = 0;
			pindx[1] = 1;
			fz_objt_fact_create_wire_face(
				windex, line_data.obj,pindx,2,NULL);

			/* add object to the project */
			err = fz_objt_add_to_project(windex,line_data.obj);

			if ( err != FZRT_NOERR )
			{	fz_objt_edit_delete_objt(windex,line_data.obj);
			}
			else
			{
				/* Get the string "Second point:" */
				err = fzrt_fzr_get_string(my_rfzr_refid, 1, 6,
prompt_str);

				/* set prompt for next point */
				fz_fuim_prompt_line(prompt_str,
					FZ_FUIM_PROMPT_LINE_NEXT,
					FZ_FUIM_PROMPT_EDIT_XYZ);
			}
		}
	}
	else if(click_count == 2)		/* handle second click */
	{
		/* reset object and construct with new second point */
		fz_objt_fact_reset(windex, line_data.obj);
		line_data.points[1] = xyz;
		fz_objt_fact_add_pnts(windex, line_data.obj,line_data.points,2);

		pindx[0] = 0;
		pindx[1] = 1;
		fz_objt_fact_create_wire_face(windex,obj,pindx,2,NULL);

		*done = 1;				/* tool complete */
	}
	return(err);
}
```

The tool track function (optional, not used for modifiers)

```
fzrt_error_td fz_tool_cbak_track(
	long			windex,
	fzrt_point		*where,
	fz_xyz_td		*where_3d,
	fz_map_plane_td		*map_plane,
	long			click_count
	);
```

This function is called by **form•Z** when the tool is the active tool and the mouse is moved in the active project window after the first click. This function is used to update any interactive input as the mouse moves in the window. In general this function performs the same action as the next click would allowing the input to appear interactive

The `windex` parameter is the active window. The `where` parameter indicates in 2 dimensional screen space where the cursor is located. The `where_3d` parameter indicates the 3 dimensional location in world space where the cursor is located. This is a point on the active reference plane provided in the `map_plane` parameter. The `click_count` parameter is the number of clicks since the start of the tool (first click).

The following example shows the track function for a tool that draws a line. This complements the previous line tool example for the click and prompt functions. In this function the location of the second point is updated to the current cursor location.

```
typedef struct
{       fz_objt_ptr                 obj;
        fz_xyz_td                   points[3];
} line_data_td;

line_data_td  line_data;

fzrt_error_td my_tool_track(
        long                        windex,
        fzrt_point                  *where,
        fz_xyz_td                   *where_3d,
        fz_map_plane_td             *map_plane,
        long                        click_count
        );
{
        fzrt_error_td       err = FZRT_NOERR;
        long                pindx[2];

        if(click_count == 1)
        {
                /* reset object and construct with new second point */
                fz_objt_fact_reset(windex, line_data.obj);
                line_data.points[1] = *where_3d;
                fz_objt_fact_add_pnts(windex, line_data.obj,line_data.points,2);

                pindx[0] = 0;
                pindx[1] = 1;
                fz_objt_fact_create_wire_face(windex, line_data.obj,pindx,2,NULL);
        }
        return(err);
}
```

The tool cancel function (optional)

```
fzrt_error_td fz_tool_cbak_cancel (
        long                windex,
        long                click_count
        );
```

This function is called by **form•Z** when a tool is interrupted. A tool can be canceled by the user using the key cancel key shortcut or by **form•Z** if a **form•Z** operation ID executed that cancels the current operation (selecting another tool for example). This function is used to cleanup any data that was generated during the execution of the tool.

The `windex` parameter is the active window. The `click_count` parameter is the number of clicks since the start of the tool (first click).

The following example complements the previous line tool example for the click, prompt and track functions. In this function, the object that was created in the prior functions is deleted.

```
typedef struct
{       fz_objt_ptr                 obj;
        fz_xyz_td                   points[3];
} line_data_td;

fzrt_error_td my_tool_cancel(
        long                windex,
        long                click_count
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        /* delete object crated at first click */
        if(click_count  >= 1)fz_objt_edit_delete_objt(windex,line_data.obj);

        return(err);
}
```

The tool undo function (optional)

```
fzrt_error_td fz_tool_cbak_undo (
        long                                windex,
        long                                click_count,
        fz_fuim_click_wait_enum             *click_wait,
        fzrt_boolean                        *done
        );
```

This function is called by **form•Z** when the user selects the undo menu item from the Edit menu during the execution of the tool. This function is used to back the input up to the state of the previous click. If this function is not provided, the tool does not perform undos during the tool.

The `windex` parameter is the active window. The `click_count` parameter is the number of clicks which will be one less than the last call to the click or prompt functions. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the click function again.

The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

```
fzrt_error_td my_tool_undo(
        long                                windex,
        long                                click_count,
        fz_fuim_click_wait_enum             *click_wait
        fzrt_boolean                        *done
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        /** return to previous click state here ***/

        return(err);
}
```

The tool redo function (optional)

```
fzrt_error_td fz_tool_cbak_redo (
	long				windex,
	long				click_count,
	fz_fuim_click_wait_enum		*click_wait
	fzrt_boolean			*done
	);
```

This function is called by **form•Z** when the user selects the redo menu item from the Edit menu during the execution of the tool. This function is used to move the input up to the state of the previously undone click. If this function is not provided, the tool does not perform redos during the tool. This function is only called immediately after a call to the undo function. Once a click or prompt entry occur, the redo is reset.

The `windex` parameter is the active window. The `click_count` parameter is the number of clicks that will be one more that the last call to the undo function. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the click function again.

The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

```
fzrt_error_td my_tool_redo (
	long				windex,
	long				click_count,
	fz_fuim_click_wait_enum		*click_wait
	fzrt_boolean			*done
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/** return to previously undone click state here ***/

	return(err);
}
```

The tool icon menu function (Optional, mutually exclusive with icon menu adjacent function)

```
fzrt_error_td fz_tool_cbak_icon_menu (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			group_uuid,
	fz_fuim_icon_group_enum		*group_pos,
	long				*group_row,
	long				*group_col
	);
```

This function is called by **form•Z** to add the tool to the Tool icon menu. The presence of this function places the tool in the Tool set of tools. If no other parameters are set then the tool will get added to a group of icons at the bottom (end) of the icon menu. Note that this only adds the position to the tool menu. The function `fz_tool_cbak_icon_rsrc` or `fz_tool_cbak_icon_file` must be provided to add custom graphics for the icon. If one of these is not provided, **form•Z** uses a generic plugin icon graphic.

The `group_uuid` parameter is assigned to all tools that should be grouped together. That is, all `fz_tool_cbak_icon_menu` implemented functions that return the same `group_uuid` parameter are placed together in the system icon menu in the same group (pop-out tool menu). This group is added to the bottom (end) of the menu. The placement of the item in the group is controlled by the `group_pos` parameter. A value of `FZ_FUIM_ICON_GROUP_START` places the item at the start of the group and a value of `FZ_FUIM_ICON_GROUP_END` places it at the end of the group. Note that these may not always yield constant results because plugin load order can vary hence multiple uses of `FZ_FUIM_ICON_GROUP_END` my note build the menu in the expected order. When `FZ_FUIM_ICON_GROUP_CUSTOM` is selected, then the `group_row` and `group_col` parameters specify the position of the item in the tool menu group.

```
#define MY_GRUP_ID
"\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

fzrt_error_td my_tool_icon_menu (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			group_uuid,
	fz_fuim_icon_group_enum		*group_pos,
	long				*group_row,
	long				*group_col
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
	*group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
	*group_row = 1;
	*group_col = 1;

	return(err);
}
```

The function `fz_fuim_exts_icon_group` can be called to better control the group containing the set of tools. This adds the ability to name the group and insert the pop-out menu group in the existing menu groups. The icon pop-out menu can be created in each `fz_tool_cbak_icon_menu` so that if the grouped items are actually in separate plugins, and the user has disabled one of the plugins, the icon menu will still be formed properly. **form•Z** ignores attempts to create a menu when the uuid already exists. That situation would occur if all the plugins are enabled. The following is an example of a pop-out menu.

```
fzrt_error_td my_tool_icon_menu (
	const fzrt_UUID_td		icon_menu_uuid,
	fzrt_UUID_td			group_uuid,
	fz_fuim_icon_group_enum		*group_pos,
	long				*group_row,
	long				*group_col
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	err = fz_fuim_exts_icon_group (
		MY_GRUP_ID, "My  Group", icon_menu_uuid,
		FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE,
		FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE);

	if(err == FZRT_NOERR)
	{	fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
		*group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
		*group_row = 1;
		*group_col = 1;
	}
	return(err);
}
```

The tool icon menu adjacent function (Optional, mutually exclusive with icon menu function)

```
fzrt_error_td fz_tool_cbak_icon_menu_adjacent(
	const fzrt_UUID_td			icon_menu_uuid,
	fzrt_UUID_td				adjacent_uuid,
	fz_fuim_icon_adjacent_enum		*where
	);
```

This function is called by **form•Z** to add the tool to the system icon menu. It serves the same purpose as the `fz_cmds_cbak_proj_icon_menu` function, however it specifies the location of the icon item quite differently. The location is identified by referencing another tool in the icon menu. The `adjacent_uuid` parameter is the UUID of the tool to which the icon should be added adjacent. The `where` parameter specifies to which side of the adjacent icon the icon should be added. The available options are `FZ_FUIM_ICON_ADJACENT_TOP`, `FZ_FUIM_ICON_ADJACENT_BOTTOM`, `FZ_FUIM_ICON_ADJACENT_LEFT`, `FZ_FUIM_ICON_ADJACENT_RIGHT`. The default action is specified by `FZ_FUIM_ICON_ADJACENT_DEFAULT` which currently is the same as `FZ_FUIM_ICON_ADJACENT_RIGHT`. New pop-out groups can not be created with this function. The following example ads the icon to the right of the **form•Z** primitive spheroid tool.

```
fzrt_error_td my_tool_icon_menu_adjacent (
	const fzrt_UUID_td			icon_menu_uuid,
	fzrt_UUID_td				adjacent_uuid,
	fz_fuim_icon_adjacent_enum		*where
	);
{
	fzrt_error_td		err = FZRT_NOERR;

	/* copy UUID of adjacent tool */
	fzrt_UUID_copy(FZ_CMND_MODEL_PRIM_SPHERE, adjacent_uuid);
	*where = FZ_FUIM_ICON_ADJACENT_RIGHT;

	return(err);
}
```

The tool icon file function (Optional, mutually exclusive with icon resource function)

```
fzrt_error_td fz_tool_cbak_icon_file (
	fz_fuim_icon_enum			which,
	fzrt_floc_ptr				floc,
	long					*hpos,
	long					*vpos,
	fzrt_floc_ptr				floc_mask,
	long					*hpos_mask,
	long					*vpos_mask
	);
```

This function is called by **form•Z** to get an icon for the tool from an image file. The icon image can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. **form•Z** will request an icon when the tool is displayed in a tool menu using `fz_tool_cbak_icon_menu` or `fz_tool_cbak_icon_menu_adjacent`.

form•Z supports 3 styles of icon display. Recall that these are selectable by the user from the Icon Style menu in the Icons Customization dialog. The first two options (White and Gray) are generated from a black and white source graphic with different treatments at drawing time. The third option is generated from a color source graphic. The first two options are older icon styles that are provided for backward compatibility. The color icons became the default with v 4.0. Note that if an icon of one type or the other (or both) is not provided, then **form•Z** uses a generic plugin icon graphic.

The `which` parameter indicates the type of source graphic icon that is needed by **form•Z**. For each type of icon source (black and white and color), there are two possible sizes. The full size icon is the size that is used in the main tool palettes and tear off tool palettes. The black and white source full size is 30 x 30 pixels and indicated by `FZ_FUIM_ICON_MONOC`. The color source is 32 x 32 pixels and indicated by `FZ_FUIM_ICON_COLOR`. The alternate size is the smaller size used for window icons that are drawn in the lower margin of the window. The alternate size for both black and white and color sources is 20 x 16 pixels and indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively.

The `floc` parameter should be filled with the file name and location of the file that contains the icon graphic. The `hpos` and `vpos` parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the plugin file. This makes it simple to find the file. The location of the plugin file can be retained during the `FZPL_PLUGIN_INITIALIZE` stage using the `fsf->fzpl_plugin_file_get_floc` function.

The `floc_mask` parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the `floc` parameter). The icon mask defines the transparent areas of the icon. The `hpos_mask` and `vpos_mask` parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided than the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of tools by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

```
fzrt_error_td my_tool_icon_file (
	fz_fuim_icon_enum		which,
	fzrt_floc_ptr			floc,
	long				*hpos,
	long				*vpos,
	fzrt_floc_ptr			floc_mask,
	long				*hpos_mask,
	long				*vpos_mask
	)
{
	fzrt_error_td	err = FZRT_NOERR;

	switch(which)
	{
		case FZ_FUIM_ICON_MONOC:
			err = fzrt_file_floc_copy(my_plugin_ floc,floc);
			if(err == FZRT_NOERR)
			{	err = fzrt_file_floc_set_name(floc,"my_icon_bw.tif");
				*hpos = 0;
				*vpos = 0;
			}
		break;
		case FZ_FUIM_ICON_COLOR:
			err = fzrt_file_floc_copy(my_plugin_ floc,floc);
			if(err == FZRT_NOERR)
			{	err = fzrt_file_floc_set_name(floc,"
		my_icon_col.tif");
				*hpos = 0;
				*vpos = 0;
			}
```

```
                break;
        }
        return(err);
}
```

The tool icon resource function (optional, mutually exclusive with icon file function)

```
fzrt_error_td fz_tool_cbak_icon_rsrc (
        fz_fuim_icon_enum           which,
        fzrt_icon_ptr               *icon
        );
```

This function is called by **form•Z** to load an icon for the tool from a platform's native (Macintosh or Windows) resource file format. This function works the same as the above icon file function except that the icon data is read from the resource file instead of the image file. These two functions are mutually exclusive (only one should be provided). Although this function and the method for loading resources is cross platform, the resource formats are not hence the data must be generated differently for each platform. This function is provided for situations where resources in these formats are already available. It is recommended that all new artwork use the icon file method described above as it is cross platform and simpler to create the content.

This function can be used to load the icon from the plugin file's resource data by using the function `fzpl_plugin_get_rlib_idx` to obtain the index for the plugins files resource data. The function `fzrt_rlib_load_icon` must be called to load the resource from the file. Use `FZRT_LOAD_ICON_BW` to indicate black and white icons and indicate color icons using `FZRT_LOAD_ICON_COLOR`. On the Macintosh platform, the black and white icons are read from 'ICON' resources and color icons from 'cicn'. On Windows black and white icons must be stored as a 1 bit depth bitmap resource with the type "FZICON" in the resource file and color icons can be stored as either a native Windows ICON or as an 8 bit deep bitmap resource. Note that on Windows, black and white icons and color icons stored as a bitmap resource will not have an icon mask. **form•Z** releases the memory for the resource when the plugin is unloaded.

All icons are stored in 32 x 32 pixel resources, however, depending on the type of the icon, only part of the resource will be used. Only the top left 30 x 30 pixels of the 32 x 32 are used for the black and white full icon size indicated by `FZ_FUIM_ICON_MONOC`. The bottom and right two pixels are NOT used (and will be cropped). The entire 32 x 32 is used for the color full icon size indicated by `FZ_FUIM_ICON_COLOR`. For the alternate size icons indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively, **form•Z** uses the bottom left 20 x 16 pixels. The top 16 and right 12 pixels are NOT used (and will be cropped).

```
fzrt_error_td my_tool_icon_rsrc (
        fz_fuim_icon_enum           which,
        fzrt_icon_ptr               *icon
        )
{
        long                        err = FZRT_NOERR;
        short                       rlib_index;

        err = fzpl_plugin_get_rlib_idx(my_plugin_runtime_ID, &rlib_index );

        if(err == FZRT_NOERR)
        {
                switch(which)
                {
                        case FZ_FUIM_ICON_MONOC:
                                err = fzrt_rlib_load_icon(
                                        rlib_index,FZRT_LOAD_ICON_BW,128,icon);
```

```
                    break;
                    case FZ_FUIM_ICON_COLOR:
                            err = fzrt_rlib_load_icon(
                                    rlib_index,FZRT_LOAD_ICON_COLOR,128,icon);
                    break;
            }
            break;
    }
    return(err);
}
```

The tool preferences IO function (optional)

```
fzrt_error_td fz_tool_cbak_pref_io (
    fz_iost_ptr                        iost,
    fz_iost_dir_td_enum                dir,
    fzpl_vers_td * const               version,
    unsigned long                      size
    );
```

form•Z calls this function to read and write any tool specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a tools data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the tool preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
fzrt_error_td my_tool_iost(
    fz_iost_ptr                        iost,
    fz_iost_dir_td_enum                dir,
    fzpl_vers_td * const               version,
    unsigned long                      size
    );
{
    fzrt_error_td          err = FZRT_NOERR;

    if ( dir == FZ_IOST_WRITE ) *version = 1;
```

```
        err = fz_iost_one_long(iost,&my_tool->value1)
        if(err == FZRT_NOERR)
        {       err = fz_iost_one_long(iost,&my_tool->value2);
                if(err == FZRT_NOERR)
                {       err = fz_iost_one_long(iost,&my_tool->value3);
                        if(err == FZRT_NOERR)
                        {       err = fz_iost_one_long(iost,&my_tool->value4);

                                if(*version >= 1)
                                {       err = fz_iost_one_long(iost,&my_tool->value5);
                                }
                        }
                }
        }

        return(err);
}
```

The tool options name function (Optional)

```
fzrt_error_td fz_tool_cbak_opts_name(
        char            *name,
        long            max_len
        );
```

This function is called by **form•Z** to get the name of the tools options. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the tool name is stored in a .fzr file so that it is localizable

```
fzrt_error_td my_tool _opts_name(
        char            *name,
        long            max_len
        )
{
        fzrt_error_td           err = FZRT_NOERR;

        /* Get the title string “My Tool Options” from the plugin’s resource file
*/
        if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 11, my_str)) ==
FZRT_NOERR)
        {
                /* copy the string to the name parameter */
                strncpy(name,  my_str, max_len);
        }
        return(err);
}
```

The tool options uuid function (optional)

```
fzrt_error_td fz_tool_cbak_opts_uuid
        fzrt_UUID_td  uuid
        );
```

This function is called by **form•Z** to get the uuid of the tools options. This unique ID is used by **form•Z** to distinguish the tool from other tools. This function is recommended for all tool plugins. If a UUID is not provided, one will be generated internally by **form•Z**. in this situation the UUID will not be the same each time **form•Z** is run and hence persistent information will not be retained. This any user customization like key shortcuts.

```
#define MY_TOOL_OPTS_ID  \
       "\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td  my_tool_opts_uuid(
       fzrt_UUID_td  uuid
       )
{
       fzrt_error_td       err = FZRT_NOERR;

       /* copy constant UUID to into the uuid parameter */
       fzrt_UUID_copy(MY_TOOL_OPTS_ID, uuid);

       return(err);
}
```

The tool options help function (optional)

```
fzrt_error_td fz_tool_cbak_opts_help(
       char          *help,
       long          max_len
       );
```

This function is called by **form•Z** to display a help string that describes the detail of what the tool does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` bytes of data. It is recommended that the tool name is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 512 bytes (characters).

```
fzrt_error_td my_tool_opts_help(
       char          *help,
       long          max_len
       )

{
       fzrt_error_td       err = FZRT_NOERR;
       char                my_str[512];

       /* Get the help string from the plugin's resource file */
       if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) ==
FZRT_NOERR)
       {
              /* copy the string to the help parameter */
              strncpy(help,  my_str, max_len);
       }
       return(err);
}
```

The tool options interface template function (optional)

```
fzrt_error_td fz_tool_cbak_opts_iface_tmpl (
       fz_fuim_tmpl_ptr           tmpl_ptr,
       fzrt_ptr                   tmpl_data
       )
```

This function is called by **form•Z** when the interface for the tool options is needed. This template is displayed inside the tool options palette when the tool is active and in a dialog when the user invokes the dialog from the icon. The **form•Z** interface template functions should be called to construct the interface of the palette in this function. Please see section 2.6 for more details on the fuim template functions. The full fuim template documentation can be found in the API reference.

The following sample is a template for 3 buttons grouped inside a boarder with a title.

```
#define MY_STRINGS         1

enum
{      MY_STRING_NAME = 1,
       MY_STRING_TYPE,
       MY_STRING_1,
       MY_STRING_2,
       MY_STRING_3
};

enum
{      MY_BUTTON1=1,
       MY_BUTTON2,
       MY_BUTTON3
};

fzrt_error_td my_tool_opts_iface_tmpl (
       fz_fuim_tmpl_ptr          tmpl_ptr,
       fzrt_ptr                  tmpl_data
       )
{
       fzrt_error_td       err;
       short               gindx;
       char                str[256];

       /* get the options title from plugin's resource file */
       fzrt_fzr_get_string(fz_rsrc_ref_func, MY_STRINGS, MY_STRING_NAME, str);
       if((err = fz_fuim_tmpl_init(tmpl_ptr, str,
              FZ_FUIM_NONE, MY_TOOL_OPTS_ID, 0)) == FZRT_NOERR)
       {
              /* create a static text item */
              fzrt_fzr_get_string(fz_rsrc_ref_func, MY_STRINGS, MY_STRING_TYPE,
str);
              gindx = fz_fuim_new_text_static(tmpl_ptr, -1, FZ_FUIM_NONE,
                            FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_EQSZ, str, NULL,
                     NULL);

                     /* create a button */
                     fzrt_fzr_get_string(fz_rsrc_ref_func,
                            MY_STRINGS, MY_STRING_1, str);
                     fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON1,
                            FZ_FUIM_FLAG_NONE, str, my__item_func, NULL);

                     /* create a button */
                     fzrt_fzr_get_string(fz_rsrc_ref_func,
                            MY_STRINGS, MY_STRING_2, str);
                     fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON2,
                            FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);

                     /* create a button */
                     fzrt_fzr_get_string(fz_rsrc_ref_func,
                            MY_STRINGS, MY_STRING_3, str);
                     fz_fuim_new_button(tmpl_ptr, gindx, MY_BUTTON3,
                            FZ_FUIM_FLAG_NONE, str, my_item_func, NULL);
       }

       return (err);
}
```

2.8.9 Utility Plugins

Utility plugins are designed to execute a task which is either less frequently used or an item in the **form•Z** interface is not desired. Utility plugins are best used on tasks that are linear in nature (like batch processing). Utility plugins are not loaded by **form•Z** at startup. This allows **form•Z** startup faster and use less memory. Utility plugins are not listed in the Extensions Manager dialog and they do not need to be located in the Extensions Manager's search paths.

The user invokes a utility plugin by selecting the Run Utility… item from the Extensions menu. The user is then prompted with a standard file open dialog to select the plugin file (.fzp) file to run. If the plugin file contains a single utility plugin, then that utility is immediately executed. If the plugin file contains multiple plugins, a dialog is presented which lists the names of the plugins in the plugin file. The user than selects the plugin to run.

When the utility plugin is invoked, **form•Z** loads the utility plugin, calls the utility main execution function to execute plugin and then the plugin is unloaded. The plugin can call **form•Z** API functions (including interface) in the main execution function to perform its task. While a utility is executing no other tasks can take place in **form•Z** (except network rendering communication). Control remains within the utility until the plugin has completed its task. To provide the best user experience is recommended that you provide the ability to cancel the operation and provide a progress bar for time-consuming tasks.

There are two variants to the utility plugins, system and project. System utilities are not dependent on a project window. Project utilities are dependent on a project window and are expected to function on the provided project window. A plugin that renders all of the **form•Z** projects in a directory is an example of a system utility.

2.8.9.1 System Utility

System utilities are defined using the `FZ_UTIL_SYST_EXTS_TYPE` and the `fz_util_cbak_syst_fset` function set as described in the following sections. The user invokes a system utility plugin by selecting the Run Utility… item from the Extensions menu. A system utility can also be invoked from another plugin or script by calling the API functions `fz_syst_plugin_exec_util` or `fz_syst_script_exec_util`. The desired utility is specified by its location and plugin file name. If the plugin file contains more than one utility, the UUID of the desired plugin must also be specified.

System utility plugin type and registration.

System utility plugins are identified with the plugin type of `FZ_UTIL_SYST_EXTS_TYPE` and must implement the `fz_util_cbak_syst_fset` call back function set. The following shows the registration of a System utility and a call back implementation. This is done from the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3.

```
fzrt_error_td my_util_syst_register_plugins()
{
	fzrt_error_td		err = FZRT_NOERR;

	err = fzpl_glue->fzpl_plugin_register(MY_PLUGIN_UUID,
		MY_PLUGIN_NAME,
		MY_PLUGIN_VERSION,
		MY_PLUGIN_VENDOR,
		MY_PLUGIN_URL,
		FZ_UTIL_SYST_EXTS_TYPE,
```

```
		FZ_UTIL_SYST_EXTS_VERSION,
		NULL /*error string function*/,
		0,
		NULL,
		&my_plugin_runtime_id);

	if ( err == FZRT_NOERR )
	{
		err = fzpl_glue->fzpl_plugin_add_fset(
			my_plugin_runtime_id,
			FZ_UTIL_CBAK_SYST_FSET_TYPE,
			FZ_UTIL_CBAK_SYST_FSET_VERSION,
			FZ_UTIL_CBAK_SYST_FSET_NAME,
			FZPL_TYPE_STRING(fz_util_cbak_syst_fset),
			sizeof ( fz_util_cbak_syst_fset ),
			my_fill_util_cbak_syst_fset, FALSE);
	}

	return (err);
}
```

System utility call back function set.

System utility plugins are implemented by the call back function set `fz_util_cbak_syst_fset`. There are three functions in this function set. Only the main execution function is required unless the plugin has multiple `fz_util_cbak_syst_fset` function sets. The following shows the fill in of a `fz_util_cbak_syst_fset` function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above.

```
fzrt_error_td my_fill_util_cbak_syst_fset (
		const fzpl_fset_def_ptr fset_def,
		fzpl_fset_td * const fset )
{
	fzrt_error_td			err = FZRT_NOERR;
	fz_util_cbak_syst_fset		*util_syst;

	err = fzpl_glue->fzpl_fset_def_check ( fset_def,
		FZ_UTIL_CBAK_SYST_FSET_VERSION,
		FZPL_TYPE_STRING(fz_util_cbak_syst_fset),
		sizeof ( fz_util_cbak_syst_fset ),
		FZPL_VERSION_OP_NEWER );

	if ( err == FZRT_NOERR )
	{
		util_syst = (fz_util_cbak_syst_fset *)fset;

		util_syst->fz_util_cbak_syst_main = my_util_syst_main;
		util_syst->fz_util_cbak_syst_name = my_util_syst_name;
		util_syst->fz_util_cbak_syst_uuid = my_util_syst_uuid;
	}

	return err;
}
```

The main execution function (required)

```
fzrt_error_td  fz_util_cbak_syst_main(
	void
```

```
        );
```

This is the main function for a System utility. When the plugin is invoked, this function is called to perform the work of the plugin. All execution for the plugin is done inside this function (or local plugin functions called from this function). When execution flow exits this function, the plugin is unloaded.

```
fzrt_error_td my_util_syst_main(
        void
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        /* Do utility work (without windex), call API functions etc. */

        return(err);
}
```

The name function (optional, required for plugins with multiple function sets)

```
fzrt_error_td fz_util_cbak_syst_name(
        char          *name,
        long          max_len
        );
```

This function is called by **form•Z** to get the name of the utility. It is recommended that the utility name is stored in .fzr file so that it is localizable. This function is recommended for all utility plugins, however, it is required if the plugin file contains multiple utility plugins. The name is presented to the user when the plugin file is selected so that the user can select which plugin should be executed.

```
fzrt_error_td my_util_syst_name(
        char          *name,
        long          max_len
        )
{
        fzrt_error_td       err = FZRT_NOERR;

        strncpy(name, "My System Utility", max_len);

        return(err);
}
```

The uuid function (optional, required for files with multiple function sets)

```
fzrt_error_td fz_util_cbak_syst_uuid
        fzrt_UUID_td  uuid
        );
```

This function is called by **form•Z** to get the UUID of the utility. This unique id is used by **form•Z** to distinguish the utility from other utilities. This function is recommended for all utility plugins, however, it is required if the plugin file contains multiple utility plugins. The UUID is used to determine which plugin to execute when the plugin file is invoked.

```
#define MY_UTIL_SYST_UUID
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td  my_util_syst_uuid(
```

```
	fzrt_UUID_td	uuid
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	fzrt_UUID_copy(MY_UTIL_SYST_UUID, uuid);

	return(err);
}
```

2.8.9.2 Project Utility

Project utilities are defined using the `FZ_UTIL_PROJ_EXTS_TYPE` and the `fz_util_cbak_proj_fset` function set as described in the following sections. The user invokes a system utility plugin by selecting the Run Utility… item from the Extensions menu. Since project utilities require a project window, they will not execute when there is no open project windows. A project utility can also be invoked from another plugin or script by calling the API functions `fz_proj_plugin_exec_util` or `fz_proj_script_exec_util`. The desired utility is specified by its location and plugin file name. If the plugin file contains more than one utility, the UUID of the desired plugin must also be specified.

Project utility plugin type and registration

Project utility plugins are identified with the plugin type of `FZ_UTIL_PROJ_EXTS_TYPE` and must implement the `fz_util_cbak_proj_fset` call back function set. The following shows the registration of a Project utility and a call back implementation. This is done from the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3.

```
fzrt_error_td my_util_proj_register_plugins()
{
	fzrt_error_td			err = FZRT_NOERR;

	err = fzpl_glue->fzpl_plugin_register(
		MY_PLUGIN_UUID,
		MY_PLUGIN_NAME,
		MY_PLUGIN_VENDOR,
		MY_PLUGIN_URL,
		MY_PLUGIN_VERSION,
		FZ_UTIL_PROJ_EXTS_TYPE,
		FZ_UTIL_PROJ_EXTS_VERSION,
		NULL /*error string function*/,
		0,
		NULL,
		&my_plugin_runtime_id);

	if ( err == FZRT_NOERR )
	{
		err = fzpl_glue->fzpl_plugin_add_fset(
			my_plugin_runtime_id,
			FZ_UTIL_CBAK_PROJ_FSET_TYPE,
			FZ_UTIL_CBAK_PROJ_FSET_VERSION,
			FZ_UTIL_CBAK_PROJ_FSET_NAME,
			FZPL_TYPE_STRING(fz_util_cbak_proj_fset),
			sizeof ( fz_util_cbak_proj_fset ),
			my_fill_util_cbak_proj_fset,
			FALSE);
	}
```

```
	return (err);
}
```

Project utility call back function set

Project utility plugins are implemented by the call back function set `fz_util_cbak_proj_fset`. There are three functions in this function set. Only the main execution function is required unless the plugin has multiple `fz_util_cbak_proj_fset` function sets. The following shows the fill in of a `fz_util_cbak_proj_fset` function set. This function is provided to the `fzpl_plugin_add_fset` function call shown above.

```
fzrt_error_td my_fill_util_cbak_proj_fset (
		const fzpl_fset_def_ptr fset_def,
		fzpl_fset_td * const fset )
{
	fzrt_error_td			err = FZRT_NOERR;
	fz_util_cbak_proj_fset		*util_proj;

	err = fzpl_glue->fzpl_fset_def_check ( fset_def,
		FZ_UTIL_CBAK_PROJ_FSET_VERSION,
		FZPL_TYPE_STRING(fz_util_cbak_proj_fset),
		sizeof ( fz_util_cbak_proj_fset ),
		FZPL_VERSION_OP_NEWER );

	if ( err == FZRT_NOERR )
	{
		util_proj = (fz_util_cbak_proj_fset *)fset;

		util_proj->fz_util_cbak_proj_main= my_util_proj_main;
		util_proj->fz_util_cbak_proj_name= my_util_proj_name;
		util_proj->fz_util_cbak_proj_uuid= my_util_proj_uuid;
	}

	return err;
}
```

The main execution function (required)

```
fzrt_error_td  fz_util_cbak_proj_main(
	long   windex
	);
```

This is the main function for a project utility. When the plugin is invoked, this function is called to perform the work of the plugin. All execution for the plugin is done inside this function (or local plugin functions called from this function). When execution flow exits this function, the plugin is unloaded.

```
fzrt_error_td  my_util_proj_main(
	long   windex
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	/* Do utilty work with windex, call API functions etc. */
```

```
	return(err);
}
```

The name function (optional, required for plugins with multiple function sets)

```
fzrt_error_td fz_util_cbak_proj_name(
	char		*name,
	long		max_len
	);
```

This function is called by **form•Z** to get the name of the utility. It is recommended that the utility name is stored in .fzr file so that it is localizable. This function is recommended for all utility plugins, however, it is required if the plugin file contains multiple utility plugins. The name is presented to the user when the plugin file is selected so that the user can select which plugin should be executed.

```
fzrt_error_td my_util_proj_name(
	char		*name,
	long		max_len
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	strncpy(name, “My Project Utility”, max_len);

	return(err);
}
```

The uuid function (optional, required for files with multiple function sets)

```
fzrt_error_td fz_util_cbak_proj_uuid
	fzrt_UUID_td	uuid
	);
```

This function is called by **form•Z** to get the UUID of the utility. This unique id is used by **form•Z** to distinguish the utility from other utilities. This function is recommended for all utility plugins, however, it is required if the plugin file contains multiple utility plugins. The UUID is used to determine which plugin to execute when the plugin file is invoked.

```
#define MY_UTIL_PROJ_UUID
"\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

fzrt_error_td my_util_proj_uuid(
	fzrt_UUID_td	uuid
	)
{
	fzrt_error_td		err = FZRT_NOERR;

	fzrt_UUID_copy(MY_UTIL_PROJ_UUID, uuid);

	return(err);
}
```

2.8.10 Surface Style

The surface style plugin class is intended to be used in conjunction with a renderer plugin (see section 2.8.7). It allows the renderer to define its own surface style extension. A renderer may use the existing surface style parameters used by RenderZone and the other display modes. However, if the renderer has a need for specialized parameters not present in the existing definition of a surface style, it needs to create them. This is achieved by the surface style plugin. Most notably, such a plugin will create a new tab in the Surface Style Parameters dialog, where these parameters are displayed to the user. For example, a renderer may choose to use the color pattern of a surface style, as defined by the Color shader, but may need its own reflection parameters. In such a case, the renderer would use the existing api functions to extract the color shader and its parameters from the surface style, but define the reflection parameters by implementing a surface style plugin. **form•Z** will handle all maintenance issues of the plugin, such as initialization, copying, deletion and io. As with other plugins, the surface style extension plugin needs to provide a set of callback functions, that are automatically invoked when an action of a certain kind occurs.

Throughout this section, the surface style plugin implemented by the x pixel sample renderer is used as an example. The original source code can be found in the **form•Z** SDK Samples directory. The x pixel sample renderer demonstrates how a renderer uses both, existing surface style parameters, and its own surface style extension. It can be used as the base for developing a complete rendering plugin including a surface style extension.

Surface Style plugin type and registration

An surface style plugin is identified with the plugin type `FZ_SREP_EXTS_TYPE` and version of `FZ_SREP_EXTS_VERSION`, and must implement the `fz_srep_cbak_fset` call back function set. The following code example shows the registration of a surface style plugin and a surface style callback function set. This is done from the plugin file's entry function while handling the `FZPL_PLUGIN_INITIALIZE` message as described in section 2.3.

```
fzrt_error_td my_srep_register_plugin ()
{
	fzrt_error_td err = FZRT_NOERR;
	/* REGISTER THE SURFACE STYLE PLUGIN */
	err = fzpl_glue->fzpl_plugin_register(
		MY_SREP_PLUGIN_UUID,
		MY_SREP_PLUGIN_NAME,
		MY_SREP_PLUGIN_VERSION,
		MY_SREP_PLUGIN_VENDOR,
		MY_SREP_PLUGIN_URL,
		FZ_SREP_EXTS_TYPE,
		FZ_SREP_EXTS_VERSION,
		NULL /*error string function*/, 0, NULL,
		&my_plugin_runtime_id);
	if ( err == FZRT_NOERR )
	{
		/* REGISTER THE SURFACE STYLE FUNCTION SET */
		err = fzpl_glue->fzpl_plugin_add_fset(
				my_plugin_runtime_id,
				FZ_SREP_CBAK_FSET_TYPE,
				FZ_SREP_CBAK_FSET_VERSION,
				FZ_SREP_CBAK_FSET_NAME,
				FZPL_TYPE_STRING(fz_srep_cbak_fset),
				sizeof (fz_srep_cbak_fset),
```

```
                              my_fill_srep_cbak_fset,
                              FALSE);
       }
       return(err);
}
```

Since a surface style plugin is usually not done as a separate plugin, but instead in combination with a renderer plugin, it is not necessary to establish two distinct instances of a plugin, one for the renderer and one for the surface style. It is better to register one plugin, that contains the function sets for the renderer and surface style. This is demonstrated in the x pixel render plugin and is expected to be the preferred implementation of a renderer and a surface style plugin. The following code example shows the registration of a renderer plugin that contains a renderer and a surface style callback function set.

```
/* create the plugin as a renderer plugin */
err = fset_glue->fzpl_plugin_register( X_RNDR_PLUGIN_UUID,
               "X Pixel",
               X_RNDR_PLUGIN_VERSION,
               X_RNDR_PLUGIN_VENDOR,
               X_RNDR_PLUGIN_URL,
               FZ_RNDR_EXTS_TYPE,
               FZ_RNDR_EXTS_VERSION,
               x_rndr_error_str,
               0,
               NULL,
               &_x_rndr_plugin_runtime_id);

if (err == FZRT_NOERR)
{
       /* register a renderer fset for the plugin with matching runtime id. */
       err = fset_glue->fzpl_plugin_add_fset( _x_rndr_plugin_runtime_id,
               FZ_RNDR_CBAK_FSET_TYPE,
               FZ_RNDR_CBAK_FSET_VERSION,
               FZ_RNDR_CBAK_FSET_NAME,
               FZPL_TYPE_STRING(fz_rndr_cbak_fset),
               sizeof(fz_rndr_cbak_fset),
               x_pixel_fill_rndr_fset,
               FALSE);
}

if (err == FZRT_NOERR)
{
       /* register a surface style fset for the plugin with matching runtime id.
*/
       err = fset_glue->fzpl_plugin_add_fset( _x_rndr_plugin_runtime_id,
               FZ_SREP_CBAK_FSET_TYPE,
               FZ_SREP_CBAK_FSET_VERSION,
               FZ_SREP_CBAK_FSET_NAME,
               FZPL_TYPE_STRING(fz_srep_cbak_fset),
               sizeof(fz_srep_cbak_fset),
               x_pixel_fill_srep_fset,
               FALSE);
}
```

Surface Style call back function set

Surface style plugins are implemented by the call back function set `fz_srep_cbak_fset`. The plugin developer must pass a fill function to `fzpl_plugin_add_fset` which assigns the pointers of the functions which define the plugin's functionality to an instance of the `fz_srep_cbak_fset` callback function set. An example of the fill function for a sample surface style is shown below.

```
fzrt_error_td x_pixel_fill_srep_fset (
       const fzpl_fset_def_ptr    fset_def,
       fzpl_fset_td* const        fset
       )
{
       fzrt_error_td        err = FZRT_NOERR;
       fz_srep_cbak_fset   *srep_fset;

       err = _fset_glue->fzpl_fset_def_check( fset_def,
                      FZ_SREP_CBAK_FSET_VERSION,
                      FZPL_TYPE_STRING(fz_srep_cbak_fset),
                      sizeof(fz_srep_cbak_fset),
                      FZPL_VERSION_OP_NEWER );

       if (err == FZRT_NOERR)
       {
              srep_fset = (fz_srep_cbak_fset*)fset;


              srep_fset->fz_srep_cbak_name                    = x_srep_name;
              srep_fset->fz_srep_cbak_uuid                    = x_srep_uuid;
              srep_fset->fz_srep_cbak_info                    = x_srep_info;
              srep_fset->fz_srep_cbak_init                    = x_srep_init;

              srep_fset->fz_srep_cbak_data_io                 = x_srep_data_io;
              srep_fset->fz_srep_cbak_data_init               = x_srep_data_init;
              srep_fset->fz_srep_cbak_data_finit              = NULL;
              srep_fset->fz_srep_cbak_data_copy               = NULL;
              srep_fset->fz_srep_cbak_data_are_equal          = NULL;
              srep_fset->fz_srep_cbak_data_iface_tmpl         = x_srep_data_iface_tmpl;
              srep_fset->fz_srep_cbak_data_iface_pview        = x_srep_data_iface_pview;
       }

       return err;
}
```

Of all the functions in the set, several are required. They are:

```
fz_srep_cbak_name
fz_srep_cbak_uuid
fz_srep_cbak_info
fz_srep_cbak_data_io
fz_srep_cbak_data_init
fz_srep_cbak_data_iface_tmpl
```

All others are optional. Note, that there is no callback function to explicitly create an surface style. **form•Z** will automatically allocate the space necessary to store the parameters of the plugin when a new surface style is created. Likewise when a surface style is deleted, **form•Z** will automatically free the previously allocated space. The callback functions fall into two categories. Four are called only once, at startup and define the basic behavior of the surface style. The remaining function operate on an instance of a surface style and are called when necessary through a runtime session of formZ. The all contain the expression `_data_` in the function name.

The name function (required)

```
fzrt_error_td fz_srep_cbak_name (
       char *name,
       long max_len
```

```
        );
```

This function is called by form•Z to get the name of the surface style extension. This name shows up in the
Surface Style Parameters dialog, where a new tab will be created to display the parameters defined by the plugin. The length of the string assigned cannot exceed `max_len` characters. It is recommended that the surface style extension name be stored in a `.fzr` resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example below, this step is omitted for the purpose of simplicity.

```
fzrt_error_td x_srep_name (
        char            *name,
        long            max_name
        )
{
        strncpy(name,"X Pixel",max_name);
        return(FZRT_NOERR);
}
```

The uuid function (required)

```
fzrt_error_td fz_srep_cbak_uuid (
        fzrt_UUID_td uuid
        );
```

This function is called by form•Z to get the uuid of the surface style extension. This unique id is used by form•Z to distinguish this surface style extension from other surface style extensions. For example, when a form•Z project file is written to disk, any surface style parameters of this plugin are saved as well and identified with this uuid. When the project file is later opened again, form•Z will connect the loaded surface style data with the installed surface style plugin. If the plugin that created the parameter is not installed, the parameters are automatically deleted. The uuid function needs to assign this unique identifier string to the function's uuid argument. An example is shown below.

```
#define X_SREP_UUID
        "\x52\xa1\x05\xa6\xb1\xc1\x4b\x01\x82\x36\xe5\xb9\x92\x70\xde\xb3"
fzrt_error_td  x_srep_uuid (
        fzrt_UUID_td                uuid
        )
{
        fzrt_UUID_copy(X_SREP_UUID,uuid);
        return(FZRT_NOERR);
}
```

The info function (required)

```
fzrt_error_td fz_srep_cbak_info (
        long *size,
        long *flags
        );
```

The info function is called by **form·Z** to retrieve basic information about the surface style extension. Two
separate pieces of information must be supplied: `size` and `flags`. **form·Z** manages the storage of each instance of a surface style extension. In order to do so, **form·Z** needs to know, what the data size (in # of bytes) of the surface style extension's content is. The size argument must be set

to the number of bytes that the surface style data storage requires. In most cases, a plugin developer will create a structure with fields which describe the surface style extension's content. The size returned to **form•Z** via this callback can be acquired with a `sizeof(structure_type)` call.

The `flags` argument tells **form•Z** basic information about the surface style extension. This argument is currently not used and is reserved for the future. It should be assigned the value of 0. The info function for the x pixel plugin is shown below.

```
typedef struct
{
	float	ambient;
	float	diffuse;
	float	specular;
	float	specular_expo;

} x_srep_td;

fzrt_error_td  x_srep_info (
	long		*size,
	long		*flags
	)
{
	*size = sizeof(x_srep_td);
	*flags = 0;
	return(FZRT_NOERR);
}
```

The init function (optional)

```
fzrt_error_td fz_srep_cbak_init (
	void
	);
```

The init function is called by **form•Z** once, at startup. It gives the plugin the opportunity to perform one time initialization procedures. One step that may be taken at that time is to acquire the run time index of the surface style extension. This run time index can be used in the api call `fz_rmtl_get_srep_data` for faster access of the parameters of a surface style. The x pixel init function is shown below.

```
long	_x_srep_indx;

fzrt_error_td  x_srep_init (
		void
	)
{
	fz_rmtl_get_srep_index(X_SREP_UUID,&_x_srep_indx);
	return(FZRT_NOERR);
}
```

The data init function (required)

```
fzrt_error_td fz_srep_cbak_data_init (
	long				windex,
	void				*data
	);
```

When a new surface style is created by formZ, the default parameters of the surface style extension need to be set. This task is performed by the data init function. A pointer to the already allocated data block is passed to the data init function (`void *data`). It can be cast to the structure that defines the surface style parameters of the plugin. If the surface style extension contains any dynamic structures, such as arrays, they need to be allocated by this function.

```
fzrt_error_td  x_srep_data_init (
       long                              windex,
       void                              *data
       )
{
       x_srep_td     *x_srep;

       x_srep = (x_srep_td*) data;

       x_srep->ambient            = 1.0;
       x_srep->diffuse            = 0.75;
       x_srep->specular           = 0.25;
       x_srep->specular_expo      = 0.5;

       return(FZRT_NOERR);
}
```

The data finit function (optional)

```
fzrt_error_td fz_srep_cbak_data_finit (
       long                              windex,
       void                              *data
       );
```

When a surface style is deleted, **form•Z** will call the data finit function. This function is optional, as the basic storage for the surface style extension is handled automatically by formZ. However, if the plugin contains any dynamical structures, that were allocated in the data init function, they need to be disposed by the plugin. This step should be performed in the data finit function. Since the x pixel sample plugin does not define any dynamic arrays, a hypothetical data finit function is shown below.

```
static fzrt_error_td        my_srep_data_finit (
       long                              windex,
       void                              *data
       )
{
       my_srep_td    *my_srep;

       my_srep = (my_srep_td*) data;

       free(my_srep->dynamic_array);

       return(FZRT_NOERR);
}
```

The data copy function (optional)

```
fzrt_error_td fz_srep_cbak_data_copy (
       long                              src_windex,
       void                              *src_data,
       long                              dst_windex,
       void                              *dst_data
       );
```

The data copy function is called anytime a surface style is copied in formZ. If the plugin's surface style extension does not contain any dynamic structures, the copy function can be omitted. In this case **form•Z** will make a byte by byte copy of the surface style parameters. However, if there are dynamic structures, the copy function must be implemented and it is responsible to copy the dynamic memory appropriately. Since the x pixel sample plugin does not define any dynamic arrays, a hypothetical data copy function is shown below. Note, that the byte by byte copy has already been performed and only the dynamic fields need to be handled.

```
fzrt_error_td my_srep_cbak_data_copy (
       long                               src_windex,
       void                               *src_data,
       long                               dst_windex,
       void                               *dst_data
       )
{
       my_srep_td    *src_srep,*dst_srep;

       src_srep = (my_srep_td*) src_data;
       dst_srep = (my_srep_td*) dst_data;

       dst_srep->dynamic_array = malloc(src_srep->n_array);
       memcpy(dst_srep->dynamic,src_srep->dynamic,src_srep->n_array);

       return(FZRT_NOERR);
}
```

The data io stream function (required)

```
fzrt_error_td        fz_srep_cbak_data_io (
       long                               windex,
       fz_iost_ptr                        iost,
       fz_iost_dir_td_enum                dir,
       fzpl_vers_td * const               version,
       unsigned long                      size,
       void                               *data
       )
```

form•Z calls this function to write a surface style extension to and read it from file. It is expected from the plugin to keep track of version changes of the surface style extension . For example, in its first release, an surface style extension may consist of four float values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, the plugin developer adds a fifth float value to increase the size to 20 bytes. When writing this new surface style extension, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the surface style extension , **form•Z** will pass in the version number of the surface style extension when it was written, in this case 0. This indicates to the plugin, that only four floats, 16 bytes, need to be read and the fifth float should be set to a default value. Likewise, it is possible, that an older version of the plugin will be asked to read a newer version of the surface style extension. This may be the case when backsaving a **form•Z** project file to an older version and then reading that file with an older version of **form•Z** that contains the older version of the surface style plugin. In this case, the plugin may choose to read the data, i.e. the first 16 bytes of version 0. For safety, it may also choose to skip any attribute data that is written with a version that is newer than the one it is currently set to. An example of the surface style io steam function is shown below. Note, that **form•Z** will allocate the basic storage for the surface style extension when reading. That is, the data pointer passed in is allocated to the size defined by the surface style extension through the `fz_srep_cbak_info` callback function.

```
fzrt_error_td        x_srep_data_io (
        long                            windex,
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum             dir,
        fzpl_vers_td* const             version,
        unsigned long                   size,
        void                            *data
        )
{
        x_srep_td           *x_srep;
        fzrt_error_td rv = FZRT_NOERR;

        x_srep = (x_srep_td*) data;

        if ( dir == FZ_IOST_WRITE ) *version = 0;

        if((rv = fz_iost_float(iost,&x_srep->ambient,1)) == FZRT_NOERR &&
           (rv = fz_iost_float(iost,&x_srep->diffuse,1)) == FZRT_NOERR &&
           (rv = fz_iost_float(iost,&x_srep->specular,1)) == FZRT_NOERR )
        {   rv = fz_iost_float(iost,&x_srep->specular_expo,1);
        }

        return(rv);
}
```

The compare function (optional)

```
fzrt_error_td fz_srep_cbak_data_are_equal(
        void *data1,
        void *data2,
        fzrt_boolean *are_equal
        );
```

For certain operations in form•Z, it is necessary to determine, whether two surface style extensions are equal in their content. The compare callback function is expected to perform this task. If this function is not implemented by the plugin, form•Z automatically determines whether the two surface styles extensions are equal, by comparing each byte in the data. The number of bytes compared is the same as the # of bytes returned by the `fz_srep_cbak_info` function. The compare function should be implemented when a straight byte comparison will not yield the proper result. This is the case, for example, when the surface styles extension contains dynamically allocated arrays. The compare function of a sample surface style extension with a dynamic array is shown below.

```
fzrt_error_td my_srep_are_equal (
        void          *data1,
        void          *data2,
        fzrt_boolean  *are_equal
)
{
        my_srep_td          *my_srep1,*my_srep2;
        fzrt_error_td       err = FZRT_NOERR;
        long                i;

        *are_equal = TRUE;
        my_srep1 = (my_srep_td*) data1;
        my_srep2 = (my_srep_td*) data2;

        /* COMPARE ARRAY SIZE */
```

```
	if (my_srep1->n_array == my_srep2->n_array )
	{

		/* COMPARE ARRAY CONTENT */
		for(i = 0; i < my_srep1->n_array; i++)
		{
			if (my_srep1->array[i] != my_srep2->array[i] )
			{
				*are_equal = FALSE;
				break;
			}
		}

		if (*are_equal == TRUE)
		{
			/* COMPARE REMAINING FIELDS */
			if (my_srep1->value1 != my_srep2-> value1 ||
			    my_srep1->value2 != my_srep2-> value2 )
			{
				*are_equal = FALSE;
			}
		}
	}
	else
	{
		*are_equal = FALSE;
	}

	return(err);
}
```

The dialog function (required)

```
fzrt_error_td		fz_srep_cbak_data_iface_tmpl (
	long				windex,
	fz_fuim_tmpl_ptr		fuim_tmpl,
	long				tab_id,
	fzrt_ptr			fuim_data
	);
```

The dialog template function is expected to create the dialog items, with which the parameters of the surface style extension are displayed. In the **Surface Style Parameters** dialog, each surface style plugin receives its own tab in which the dialog items are organized. The id of the tab group is passed to the `fz_srep_cbak_data_iface_tmpl` callback function. It is important that each new dialog item created by the `fz_srep_cbak_data_iface_tmpl` callback function is derived from this group. Recall, that all template item creation functions receive a parent id argument (see section 2.6.3 for more details). The `tab_id` argument or id's of children of the tab group must be used as this parent id argument in the creation functions. The dialog function which creates four sliders for the parameters of the x pixels renderer is shown below.

```
fzrt_error_td x_srep_data_iface_tmpl (
	long			windex,
	fz_fuim_tmpl_ptr	fuim_tmpl,
	long			tab_id,
	fzrt_ptr		fuim_data
	)
{
	short		align[4];
	x_srep_td	*x_srep;
```

```
x_srep = (x_srep_td*) fuim_data;

fz_fuim_new_slid_edit_pcent_float(
	fuim_tmpl,
	tab_id,
	"Ambient Factor",
	FZ_FUIM_NONE,
	FZ_FUIM_NONE,
	0.0,
	1.0,
	0.0,
	100.0,
	FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
	FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL,
	NULL,
	&x_srep->ambient,
	&align[0],
	NULL
);

fz_fuim_new_slid_edit_pcent_float(
	fuim_tmpl,
	tab_id,
	"Diffuse Factor",
	FZ_FUIM_NONE,
	FZ_FUIM_NONE,
	0.0,
	1.0,
	0.0,
	100.0,
	FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
	FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL,
	NULL,
	&x_srep->diffuse,
	&align[1],
	NULL
);

fz_fuim_new_slid_edit_pcent_float(
	fuim_tmpl,
	tab_id,
	"Specular Factor",
	FZ_FUIM_NONE,
	FZ_FUIM_NONE,
	0.0,
	1.0,
	0.0,
	100.0,
	FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
	FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL,
	NULL,
	&x_srep->specular,
	&align[2],
	NULL
);
fz_fuim_new_slid_edit_pcent_float(
	fuim_tmpl,
	tab_id,
	"Specular Shininess",
	FZ_FUIM_NONE,
	FZ_FUIM_NONE,
	0.0,
	1.0,
	0.0,
	100.0,
```

```
                FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
                FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL,
                NULL,
                &x_srep->specular_expo,
                &align[3],
                NULL
        );
        fz_fuim_item_align(fuim_tmpl, FZ_FUIM_ALIGN_VLFT | FZ_FUIM_ALIGN_MAX, 4, align);

        return(FZRT_NOERR);
}
```

The dialog preview function (optional)

```
fzrt_error_td           fz_srep_cbak_data_iface_pview (
        long                    windex,
        long                    pview_windex,
        long                    action,
        fzrt_boolean            dirty,
        fz_fuim_tmpl_ptr        fuim_tmpl,
        fzrt_ptr                data,
        unsigned char           *image_buffer,
        fzrt_boolean            *complete
        );
```

This function is called whenever a Surface Style is edited. It is expected to create the preview rendering that show the surface style rendered on a sample object. The preview scene is defined as a separate **form•Z** project, whose index is passed in as the `pview_windex` parameter. All project settings are defined in such a way, that the plugin can render the scene using the view, surface styles, images size ... of that project.

The `action` argument tells the preview function, when it is called. A value of 0 indicates, that it is called when the dialog is first opened by the user. It gives the plugin the opportunity to initialize any data necessary for the preview. It is not expected to generate a rendering at that time. A value of 1 indicates, that a new preview rendering is needed. It is called as often with that value as the user makes changes. A value of 2 indicates that the user closed the dialog. The plugin may now finalize any data. No rendering is expected at that time.

The preview function should call the api `fz_rmtl_iface_pview_update` as frequently as possible to allow the user to see the progress of the preview rendering in the dialog. A good interval would be, for example, once every scanline. The `fz_rmtl_iface_pview_update` api expects the preview rendering to be defined as an rgb pixel buffer of `FZ_SREP_PVIEW_SIZE * FZ_SREP_PVIEW_SIZE * 3` bytes. This buffer is allocated by **form•Z** and passed to the preview function via the `image_buffer` argument.

The preview function should also call the api `fz_rmtl_iface_pview_interrupt` frequently. If this api returns `TRUE`, the preview rendering needs to be interrupted and the preview function must return `FALSE` for the `complete` argument. For example, the user may have selected a dialog item while the rendering is executing. In order to do this, the rendering needs to stop, controll needs to be returned to the dialog driver, which will handle the user's click. If the user changed a setting, the preview function will be called again with `TRUE` for the `dirty` argument, meaning that a new image needs to be started. If the user did not make any changes, the preview function will be called with `FALSE` for the `dirty` argument. The preview function may then continue the rendering, where it was previously interrupted. If a preview rendering is completed, the `complete` argument must be returned as `TRUE`. Again, it is important, that interrupting is handled properly and in a responsive manner to allow the user to interact with the dialog while the rendering is proceeding.

If this function is not implemented, **form•Z** will display the default preview rendering, that is also shown in the Simple or RenderZone tab. The preview function for the x pixel renderer is shown below.;

```
static long   _last_scanline = 0;

fzrt_error_td        x_srep_data_iface_pview (
       long                       windex,
       long                       pview_windex,
       long                       action,
       fzrt_boolean               dirty,
       fz_fuim_tmpl_ptr           fuim_tmpl,
       fzrt_ptr                   fuim_data,
       unsigned char              *image_buffer,
       fzrt_boolean               *complete
       )
{
       long          first_scanline,scan_line;
       fzrt_boolean  cancelled;

       switch(action)
       {
              case 0 :            break;


              case 1 :
                     cancelled = FALSE;
                     /* previously cancelled rendering */
                     /* starts back at the last scanline rendered */
                     if ( dirty == 0 )   first_scanline = _last_scanline+1;
                     /* new rendering starts at scanline 0 */
                     else                first_scanline = 0;

                     /* render the scene using the preview windex */
                     x_pixel_activate(pview_windex);
                     x_pixel_image_prep(pview_windex);

       x_pixel_image_render_core(pview_windex,FZRT_NOERR,NULL,fuim_tmpl,
                            image_buffer,first_scanline,&cancelled,&scan_line);
                     x_pixel_image_finit(pview_windex);
                     x_pixel_deactivate(pview_windex);

                     /* return cancel status */
                     *complete = cancelled ? FALSE : TRUE;

                     /* store last rendered scanline */
                     if ( cancelled )    _last_scanline = scan_line;
                     else                _last_scanline = 0;
              break;


              case 2 :            break;


       }

       return(FZRT_NOERR);
}
```

3.0 Writing form•Z Scripts

3.1 Introduction

A **form•Z** script is an extension to **form•Z** in the form of a file that contains a set of instructions which execute **form•Z** functionality. These instructions are stored in binary form, and are generated from a text based script file with the .fsl file extension. The script is written in the **form•Z** Script Language (FSL) and compiled into a binary representation. This binary file is referred to as the script executable file and must have a .fsb extension to identify it as a **form•Z** script.

form•Z automatically recognizes scripts by finding them in designated directories at startup. The default directory is the "Scripts" folder in the **form•Z** application folder (and can be customizable through the Extensions dialog's search paths). When **form•Z** finds a file with the .fsb extension it validates it as a script file. The validation process prevents a non-script file with an .fsb extension from producing undesirable results.

form•Z validates each script file to be sure that it is in fact a **form•Z** script and not another file that has been given the .fsb extension. This prevents **form•Z** from crashing when attempting to load an invalid file. The validation is done automatically at startup. Script executable files generated by **form•Z** always carry the proper markings, which allow **form•Z** to identify them as scripts.

The communication between **form•Z** and a script is done through functions. There are two types of functions: **API** and **call back** functions. API functions are provided by **form•Z** for the script to use. They typically execute processes or operations that are available in the main **form•Z** program, for example a section operation or an object face building process. Call back functions are implemented by the script. These functions are called by **form•Z** as needed to perform the script tasks. Call back functions must have a specific name, based on the type of script, and a specific number of arguments.

form•Z uses UUIDs (Universal Unique Identifier) throughout for uniquely identifying entities and avoiding naming collisions. A UUID is a 16-byte string that is generated using an algorithm that guarantees a unique sequence of bytes (string) among all such generated strings. Scripts must use UUIDs in various places to guarantee that they do not collide with other scripts or **form•Z**. For example, a script that defines a RenderZone shader must provide a UUID. This distinguishes it from other scripts and also allows **form•Z** to retain information about the script (for example, its user-controlled enable state in the **Extensions** dialog). **form•Z** comes with a utility plugin to automatically generate UUIDs which is of particular use for extension developers. It is not recommended to create a UUID by "making one up" without a computer.

3.2 FSL Language Reference

The **form•Z** Script language (FSL) is a simple C-like programming language, which allows a **form•Z** user to extend the functionality of **form•Z** by writing scripts. A script is a simpler version of a plugin. It is not compiled by a separate development environment, such as Microsoft Visual Studio or MetroWerks CodeWarrior. A script is compiled by **form•Z** and stored in a more compact binary file, which is ready for efficient execution. **form•Z** has a simple text editor, where the source code of a script can be edited and compiled. A typical cycle of working with a script would look like this:

1. The user opens a new script text edit window or opens an existing script in the **form•Z** script editor application.This is an application dedicated entirely to writing and compiling scripts.
2. New source code is developed or changes are made to the source code of an existing script.
3. The script is compiled and stored in a binary version on disk.
4. The next time the regular **form•Z** application starts up, it searches for scripts. Depending on the script type, the script appears in different **form•Z** areas, such as RenderZone shaders or modeling tools.
5. Executing the script within **form•Z** will run the compiled binary version of the script (.fsb file).
6. Depending on the outcome of the operation, the user may make further changes to the script source code and repeat steps 3 to 5.

A more detailed discussion, of which areas of **form•Z** can be extended by scripts is provided in section 3.7. The following sections focus on the script language itself, rather than on particular uses of it. The sample code provided is purposely kept simple and generic so that the reader may follow the text here, without having to execute the script code in **form•Z** to see its effects.

The basic syntax and structure of the FSL follows that of the C programming language, but is simplified to enable novice programmers and users with little programming background to use it easily. At the end of this chapter the specific differences from the C programming language are outlined.

3.2.1 Basic language and script structure

A script consists of two parts: a **header** and a **body**. The header tells **form•Z** what type the script is. The body contains one or more functions. Certain functions are call back functions and are required by **form•Z** to be able to connect to and execute the script. In addition to the required functions, a script developer may also add as many functions as desired to accomplish the tasks at hand.

A function consists of a **header** and a **body**. The function header contains the **return type**, the **function name**, and the **function argument list**. The return type specifies what type of a value the function returns, if any. If it returns no value, “void” is used as a return type. The function name is an FSL **name**. The argument list, which may or may not exist, is a list of names with a type specification before each one.

An FSL **name** is a string of at most 128 characters, which may be lower or upper case letters, the character _ (underscore), or numbers in any combination, except that the first character can not be a number.

The function body contains a set of different types of statements. The statement types available in FSL are:

Declarations, **assignment** statements, **if** statements, **switch** statements, **for** loops, **while** loops, **do while** loops, **break** statements, **continue** statements, **goto** statements, **function calls**, and **return** statements.

All types of statements are delineated by a semi-colon (;). More than one statement can be written on the same line. If a function body contains declarations, they should be at the top of the body, and if a function returns a value its execution should always end with a return statement that specifies what value is returned. The different types of FSL statements are discussed in full detail later in this document.

Within a script file one may also provide documentation, or **comments**, anywhere to further describe what the code is doing. Comments are useful for explaining difficult-to-follow code, and provide for easier maintenance during the life of a script.

There are also different types of scripts and complete details about these types and their respective script structures are again discussed later in this document. The simplest type is the **utility script**, which, once compiled, can be executed through the **Run Utility...** item in the **Extensions** menu. This item invokes a standard Open File dialog from where the script to be executed can be selected. A simple but fairly complete example of a utility script is presented in the next section.

3.2.2 Introductory example

The script code in the following example will generate a simple **form•Z** object, by calling functions provided by **form•Z**. **form•Z** actually offers a large number of functions, called API functions, that perform a wide range of tasks. For example, they allow a script to create objects, modify existing objects, perform complex geometry calculations, create and edit lights, surface styles etc. As a matter of fact, most of the operations, that can be executed in **form•Z** through the user interface can also be executed through one or more API function calls.

```
void create_cube(long windex)
{
      fz_xyz_td         wdh,origin;
      fz_objt_ptr       obj;

      wdh = {10.0, 10.0, 30.0};
      origin.x = 100.0;
      origin.y = 0.0;
      origin.z = 0.0;

      fz_objt_cnstr_cube(windex,wdh,origin,NULL,obj);

      fz_objt_add_objt_to_project(windex,obj);
}
```

The function `create_cube` creates a simple **form•Z** cuboid of a given size and location. While it creates an object, it returns no value and the function is thus declared as `void`. It takes one function argument, `windex`, which is a long integer. This argument is usually supplied by **form•Z** and identifies a project, such as the one belonging to the currently active modeling window. Most **form•Z** API functions also take `windex` as an argument, to identify in which project a particular operation is executed. The function declares three variables:

```
      fz_xyz_td         wdh,origin;
```

```
fz_objt_ptr        obj;
```

They are two `fz_xyz_td` and a pointer to a modeling object `fz_objt_ptr`. `wdh` and `origin` will be used to define how large the cube is and where it is located. The next four statements assign values to the two `fz_xyz_td` variables:

```
wdh = {10.0, 10.0, 30.0};
origin.x = 100.0;
origin.y = 0.0;
origin.z = 0.0;
```

In the case of `wdh,` the assignment is done in one line. A `fz_xyz_td` type of variable has three components, named `x`, `y`, and `z`. Individually they are written `wdh.x, wdh.y`, and `wdh.z`. They can all get their values with one assignment, using the notation `{10.0, 10.0, 30.0}`, as in the example. Or they can be assigned values individually, as is done for `origin.` Both methods are equally valid, and make no difference to the compiler.

After the assignments, the cube is created by calling the **form•Z** API function:

```
fz_objt_cnstr_cube(windex,wdh,origin,NULL,obj);
```

The `windex` argument received by the `create_cube` function is passed on to the API function. The `wdh` and `origin` variables are also passed in. An optional rotation parameter is not supplied. Instead the `NULL` argument is passed, meaning that the default rotation of 0° is to be used. The result of the API call is a new object, `obj`, which is returned by the API function through the last argument. Another **form•Z** API call takes the cube and makes it a permanent part of the current project:

```
fz_objt_add_objt_to_project(windex,obj);
```

Without this last API call, the cube would remain tagged as a temporary object and would not show up in the Object palette or on the screen.

The above code is a function generating a cuboid, but it is not a complete script yet. To become a script we need to add a script header and supply a required call back function required by **form•Z**, and call the `create_cube` function within the call back function. This would be as follows:

```
script_type FZ_UTIL_PROJ_EXTS_TYPE

void create_cube(long windex)
{
	fz_xyz_td		wdh,origin;
	fz_objt_ptr		obj;

	wdh = {10.0, 10.0, 30.0};
	origin.x = 100.0;
	origin.y = 0.0;
	origin.z = 0.0;

	fz_objt_cnstr_cube(windex,wdh,origin,NULL,obj);

	fz_objt_add_objt_to_project(windex,obj);
}

long fz_util_cbak_proj_main(long windex)
```

```
{
       create_cube(windex);

       return(FZRT_NOERR);
}
```

The above code can now be saved as a script file, using **Save As...**. Once saved and compiled it can be executed, using **Run Utility...**.

Concluding the example we should note that the cuboid generated by the above script will always be of a fixed size, since the assignment of its size is hard coded in the script. Typically the advantage of creating objects in this manner would be to automatically generate thousands of objects, perform routine complex operations, and any other procedural task. A more general cube creating function would probably accept the dimensions, origin, and rotation of the object as arguments to the function. Through the normal user interface of **form•Z**, the dimensions of the cuboid are input by the user through either numeric or graphic input. This too can be accomplished with a script. In addition, an object generation operation would normally be executed through a tool with an icon. Needless to say that all these options are possible to be implemented in a script, but will be discussed later.

3.2.3 Types of variables and constants

In the **form•Z** script language, values are stored in **variables**. Since there are different types of values that can be stored, each variable needs to be declared as a specific data type before it is used to store a value. A variable, once declared with a type and an FSL name, can then be accessed by its name to store and retrieve the values in it. Values may also be explicitly defined as **constants**, which are also of different types. The types provided in FSL, for both variables and constants, are as follows:

```
fzrt_boolean
long
double
fz_string_td
fzrt_UUID_td

fz_xy_td
fz_xyz_td
fz_xy_mm_td
fz_xyz_mm_td
fz_plane_equ_td
fz_rgb_float_td
fzrt_point
fzrt_rect

fz_mat3x3_td
fz_mat4x4_td
fz_map_plane_td
```

fz_tag_td
various pointers and
various enums

fzrt_boolean, long, double, fz_string_td, and fzrt_UUID_td are referred to as **simple** types. fz_xy_td, fz_xyz_td, fz_xy_mm_td, fz_xyz_mm_td, fz_plane_equ_td, fz_rgb_float_td, fzrt_point, and fzrt_rect are referred to as **structured** types, and fz_mat3x3_td, fz_mat4x4_td, fz_map_plane_td, and fz_tag_td are referred to as **complex** types.

A simple type has exactly one numeric (long, double), logical (fzrt_boolean), or text (string) value. A structure type has two or more **fields**, each of which has its own numeric value. For example the fz_xy_td type has an x and a y field, which can be accessed by adding the field name to the end of the variable name. Assuming a variable named my_xy is declared to be of type fz_xy_td, the x and y fields of my_xy can be accessed as my_xy.x and my_xy.y. A complex type is also composed of more than one numeric value but its fields are not known to the script and cannot be accessed like those of a structure type. Variables of complex types are usually initialized and set to specific values by a call to a **form•Z** API function. For example, the **form•Z** API function fz_math_mat4x4_set_identity sets a 4 by 4 matrix variable to the identity matrix. Pointer and enum types are special simple types and are discussed in more detail in subsequent sections.

A type in a script can manifest itself as a constant value or as a variable. A constant value explicitly states the value of the type. For example, an integer constant would be written as a whole number, such as 1 or -33. A variable of a certain type is declared at the beginning of a function. Different values may be assigned to the variable as the function statements are executed. Examples of how variables are declared and of how constants are written are included in the following paragraphs. Note, that only simple and structured types can be used as both constants and variables. Complex types can only be variables.

The fz_string_td and fzrt_UUID_td types are effectively equivalent and only the fz_string_td type is described in the remainder of this document. The fzrt_UUID_td type is used to store characters for a unique identifier string, which is used throughout the script types to uniquely identify various script components. This is described in more detail in section 1.4.1.

Booleans

Syntax: fzrt_boolean

Variables of this type can only contain the values TRUE or FALSE. TRUE and FALSE are also the constant values when used in the script code. For example:

```
fzrt_boolean      can_do;

can_do = TRUE;
```

Integer numbers

Syntax: long

This is an integer (whole) number in the range -2^{31} to 2^{31}. The keyword "long" is taken from the C language where it stands for a long integer (as opposed to a short integer, which can only take on

values of the range -2^{16} to 2^{16}). Constant integer values are written by using whole numbers, with an optional minus sign before the number.
For example:

```
long  my_int;

my_int = -1000;
my_int = 0;
my_int = 32767 – 1000000 + 1 – 9999999;
```

Floating point numbers

Syntax: `double`

This is a double precision floating point (real) number. The maximum range depends on the hardware and operating system. Constant floating point numbers are written with a whole and a fractional part, separated by a dot. If a minus sign is written before the number, it becomes negative. For example:

```
double            my_float;

my_float  = -1;
my_float  = 1.001;
my_float  = -9999.1 + .5;
my_float  = 0.;
my_float  = 0.0;
my_float  = 1000.;
```

Text strings

Syntax: `fz_string_td`

The `fz_string_td` type provides storage for text strings. A constant string contains one or more characters which must be enclosed by quotes. For example:

```
fz_string_td      my_string;

my_string = "This is a string";
```

There is a limit of 255 characters for each fz_string_td.

Universal Unique Identifier (UUID)

Syntax: `fzrt_UUID_td`

The `fzrt_UUID_td` provides storage for a 16 byte character string, which serves as a unique identifier. It is used throughout **form•Z**, to distinguish one entity from another. For example, a script may be tagged with a UUID so that **form•Z** can keep it apart from another script. The assignment of a UUID is done like a text string, by enclosing 16 character bytes in double quotes. Typically, a UUID byte is written in hexadecimal notation, and is usually generated by a computer to guarantee uniqueness. It is not advisable to simply make one up. An assignment of a UUID constant to a variable would look like this:

```
fzrt_UUID_td      my_uuid
```

```
my_uuid = "\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9";
```

2D Coordinate

Syntax: `fz_xy_td`

The `fz_xy_td` type is a composite type, usually identifying a 2D coordinate value. It consists of two floating point members, named `x` and `y`. When a variable is declared as being of type `fz_xy_td`, the content of the variable is accessed by using the variable name and adding `.x` or `.y` to it. For example:

```
fz_xy_td     my_xy;

my_xy.x = 100.0;
my_xy.y = 0.0;
```

A constant `fz_xy_td` value is written by enclosing two floating point numbers in braces, and separating them by a comma. The content of the variable `my_xy` can be set to the same value as above using a `fz_xy_td` constant value:

```
my_xy = { 100.0, 0.0 };
```

3D Coordinate

Syntax: `fz_xyz_td`

The `fz_xyz_td` type is a composite type, usually identifying a 3D coordinate value. It consists of three floating point members, named `x, y` and `z`. When a variable is declared as being of type `fz_xyz_td`, the content of the variable is accessed by using the variable name and adding `.x, .y` or `.z` to it. For example:

```
fz_xyz_td    my_xyz;

my_xyz.x = 100.0;
my_xyz.y = 200.0;
my_xyz.z = 0.0;
```

A constant `fz_xyz_td` value is written by enclosing three floating point numbers in braces and separating them by commas. The content of the variable `my_xyz` can be set to the same value as above using a `fz_xyz_td` constant value:

```
my_xyz = { 100.0, 200.0, 0.0 };
```

2D bounding box

Syntax: `fz_xy_mm_td`

The `fz_xy_mm_td` type is a composite type, usually identifying a 2D bounding box, that has a lower and an upper limit in the x and y direction. It consists of four floating point members, named `xmin, ymin, xmax` and `ymax`. When a variable is declared as being of type `fz_xy_mm_td`, the content of the variable is accessed by using the variable name and adding `.xmin, .ymin, .xmax` or `.ymax` to it. For example:

```
fz_xy_mm_td my_xy_mm;

my_xy_mm.xmin = -100.0;
my_xy_mm.ymin = 0.0;
my_xy_mm.xmax = 100.0;
my_xy_mm.ymax = 200.0;
```

A constant `fz_xy_mm_td` value is written by enclosing four floating point numbers in braces, and separating them by a comma. The content of the variable `my_xy_mm` can be set to the same value as above using a `fz_xy_mm_td` constant value:

```
my_xy_mm = { -100.0, 0.0, 100.0, 200.0 };
```

3D bounding box

Syntax: `fz_xyz_mm_td`

The `fz_xyz_mm_td` type is a composite type, usually identifying a 3D bounding box, that has a lower and an upper limit in the x, y and z direction. It consists of siz floating point members, named `xmin, ymin, zmin, xmax, ymax` and `zmax`. When a variable is declared as being of type `fz_xyz_mm_td`, the content of the variable is accessed by using the variable name and adding `.xmin, .ymin, .zmin, .xmax, .ymax` or `.zmax` to it. For example:

```
fz_xyz_mm_td       my_xyz_mm;

my_xyz_mm.xmin = -100.0;
my_xyz_mm.ymin = 0.0;
my_xyz_mm.zmin = 0.0;
my_xyz_mm.xmax = 100.0;
my_xyz_mm.ymax = 200.0;
my_xyz_mm.zmax = 50.0;
```

A constant `fz_xyz_mm_td` value is written by enclosing six floating point numbers in braces, and separating them by a comma. The content of the variable `my_xyz_mm` can be set to the same value as above using a `fz_xyz_mm_td` constant value:

```
my_xyz_mm = { -100.0, 0.0, 0.0, 100.0, 200.0, 50.0 };
```

Plane equation

Syntax: `fz_plane_equ_td`

The `fz_plane_equ_td` type is a composite type, usually identifying a 3D plane equation of the form $ax + by + cz + d = 0$. It consists of four floating point members, named `a, b, c` and `d`. When a variable is declared as being of type `fz_plane_equ_td`, the content of the variable is accessed by using the variable name and adding `.a, .b, .c` or `.d` to it. For example:

```
fz_plane_equ_td          my_plane;

my_plane.a = 1.0;
my_plane.b = 0.0;
my_plane.c = 0.0;
my_plane.d = 100.0;
```

A constant fz_plane_equ_td value is written by enclosing four floating point numbers in braces, and separating them by commas. The content of the variable my_plane can be set to the same value as above using a fz_plane_equ_td constant value:

```
my_plane = {1.0, 0.0, 0.0, 100.0};
```

RGB Color

Syntax: fz_rgb_float_td

The fz_rgb_float_td type is a composite type, usually identifying an rgb color (red, green, blue). It consists of three floating point members, named r, g and b. When a variable is declared as being of type fz_rgb_float_td, the content of the variable is accessed by using the variable name and adding .r, .g, or .b to it. For example:

```
fz_rgb_float_td          my_color;

my_color.r = 1.0;
my_color.g = 1.0;
my_color.b = 0.0;
```

A constant fz_rgb_float_td value is written by enclosing three floating point numbers in braces, and separating them by commas. The content of the variable my_color can be set to the same value as above using a fz_rgb_float_td constant value:

```
my_color = {1.0, 1.0, 0.0};
```

Usually colors are represented by values between 0.0 and 1.0. For example, all white would be {1.0, 1.0, 1.0}, all black would be {0.0, 0.0, 0.0}, and pure red would be {1.0, 0.0, 0.0}.

Screen point

Syntax: fzrt_point

The fzrt_point type is a composite type, usually identifying a point on the screen with the upper left corner being (0, 0). It consists of two integer members, named h and v, (for horizontal and vertical position). When a variable is declared as being of type fzrt_point, the content of the variable is accessed by using the variable name and adding .h or .v to it. For example:

```
fzrt_point        my_point;

my_point.h = 100;
my_point.v = 200;
```

A constant fzrt_point value is written by enclosing two integer numbers in braces, and separating them by commas. The content of the variable my_point can be set to the same value as above using a fzrt_point constant value:

```
my_point = {100, 200};
```

Screen rectangle

Syntax: fzrt_rect

The `fzrt_rect` type is a composite type, usually identifying a rectangle on the screen. It consists of four integer members, named `left, top, right` and `bottom`. When a variable is declared as being of type `fzrt_rect` , the content of the variable is accessed by using the variable name and adding `.left, .top, .right` or `.bottom` to it. For example:

```
fzrt_rect          my_rect;

my_rect.left       = 0;
my_rect.top        = 0;
my_rect.right      = 200;
my_rect.bottom     = 100;
```

A constant `fzrt_rect` value is written by enclosing two integer numbers in braces, and separating them by commas. The content of the variable `my_rect` can be set to the same value as above using a `fzrt_rect` constant value:

```
my_rect = {0, 0, 200, 100};
```

3 by 3 matrix

Syntax: `fz_mat3x3_td`

The `fz_mat3x3_td` type is a complex type. It identifies a 3 by 3 matrix. It does not have any fields that can be accessed directly, like the `fz_xyz_td` type. There are a number of math functions, which set and use 3 by 3 matrices. A simple example is shown below:

```
fz_mat3x3_td            mat;

fz_math_3x3_set_identity(mat);
```

There is no constant value for a 3 by 3 matrix.

4 by 4 matrix

Syntax: `fz_mat4x4_td`

The `fz_mat4x4_td` type is a complex type. It identifies a 4 by 4 matrix. It does not have any fields that can be accessed directly, like the `fz_xyz_td` type. There are a number of math functions, which set and use 4 by 4 matrices. A simple example is shown below:

```
fz_mat4x4_td            mat;

fz_math_4x4_set_identity(mat);
```

There is no constant value for a 4 by 4 matrix.

Mapping plane

Syntax: `fz_map_plane_td`

The fz_map_plane_td type is a complex type. It defines a plane, which has an origin and rotation, in 3d space. It does not have any fields that can be accessed directly, like the `fz_xyz_td` type.

There are a number of math functions, which set and use mapping planes. A simple example is shown below:

```
fz_map_plane_td   my_plane;
fz_xyz_td   p1,p2,p3;

p1 = {100.0, 0.0, 0.0};
p2 = {0.0, 0.0, 0.0};
p3 = {0.0, 100.0, 0.0};
fz_math_3d_map_plane_from_pts(p1,p2,p3,my_plane);
```

Pointers

There are many pointer types supported by FSL with extensions "`_ptr`", however, the generic pointer `fzrt_ptr`, will be explained first.

Syntax: `fzrt_ptr`

The `fzrt_ptr` type identifies a location in memory to which it points. This location in memory usually contains data that will be operated on. A variable of type `fzrt_ptr` cannot be set to an explicit constant value other than `NULL`, which is the only pointer constant. It means that the pointer is not pointing to a location in memory, but is unassigned. Usually the `fzrt_ptr` variable is set by calling a function, or by assigning another pointer type to it.

There are also quite a few specific pointer types, for example, the pointer type `fz_objt_ptr`. In this case, a variable of type `fz_objt_ptr` points to a **form•Z** object in memory. **form•Z** API functions that operate on specific entities, expect pointers of a given type to be passed in. For example, an API function that deletes an object requires the argument to be of type `fz_objt_ptr`. Whereas the API function which deletes a light requires the argument to be of type `fz_lite_ptr`.

```
fz_objt_ptr objt;

objt = NULL; /* set objt variable to "no memory location" */

fz_objt_cnstr_cube(windex,wdh,origin,NULL,objt); /* create objt,
                                                    now it points to
                                                    memory */
fz_objt_edit_delete_objt(windex, objt);   /* delete objt */
```

Tags

Syntax: `fz_tag_td`

The `fz_tag_td` type stores a unique tag for identifying different kinds of data or entities within **form•Z**. This tag id is usually provided by **form•Z** via a function call to ascertain information about some entity. Later one uses the tag in subsequent function calls to change or retrieve the information of the entity identified by the tag. For example, one may want to set an existing surface style to a newly created object. One must get the surface style tag from the surface style pointer, and pass that tag to the API function which sets the surface style of the newly created object.

```
Fz_objt_ptr new_obj;
Fz_rmtl_ptr surf_style;
```

```
Fz_tag_td surf_tag;

... /* create new_obj, get a surf_style */

fz_rmtl_ptr_to_tag(windex, surf_style, surf_tag);
fz_objt_attr_set_objt_rmtl(windex, new_obj, surf_tag);
```

Enums

There are many enum types, all of which end with the letters "`_enum`". An enum is similar to an integer, except that it can take on only certain values. The values an enum can take on are predefined as constants. For example, a variable of type `fz_objt_model_type_enum` can only take one of the following three values: `FZ_OBJT_MODEL_TYPE_UNSPEC`, `FZ_OBJT_MODEL_TYPE_FACT`, or `FZ_OBJT_MODEL_TYPE_SMOD`. Note that these three names, written in upper case characters correspond to preset numeric values and are equivalent to constants. These constant values are defined by **form•Z** and can be found in the API documentation and header files.

```
fz_objt_model_type_enum my_model_type;

my_model_type = FZ_OBJT_MODEL_TYPE_FACT;
```

Many **form•Z** API functions take enums as a function argument. By only allowing certain values for the enum, it is ensured that only correct values are passed to the function. For example, a compilation error would occur when trying to assign the constant value `FZ_LITE_TYPE_POINT` to a variable of type `fz_objt_model_type_enum`. The `FZ_LITE_TYPE_POINT` constant is reserved for enums of type `fz_lite_type_enum`.

3.2.4 Functions

A function provides a convenient way to group several statements together, which perform a specific task. A function has a specific structure. It consists of a **header** and a **body**. The function header contains the return type, the function name and the function arguments. The function body starts with zero or more variable declarations and is followed by the statements. A formal syntactic definition of a function is as follows:

Syntax:

return_type function_name(*arguments*)
{ *declarations*
statements
}

arguments is: mod_{opt} *type* arg_name or mod_{opt} *type* arg_name, *arguments*

return_type is: *type* or void

declarations is: zero or more *declaration*

statements is: zero or more *statement*

A *type* is one of the types previously explained. A *declaration* is a type followed by one or more variable names (separated by commas if more than one), and is explained in more detail later. A *statement* will be described later as well.

The function return type

When the script is run and the statements in a function are executed, the function may return a value after the execution of the last statement. Since all values have a type, the function itself must have a return type. This may be any of the data types seen so far. For example, a function may return `TRUE` if it succeeded to perform its task, or `FALSE`, if it didn't. In this case, the return type of the function would be declared as `fzrt_boolean`. Here is a simple example of a function that determines whether an integer value is even or odd.

```
fzrt_boolean is_even(long value)
{
      fzrt_boolean      rv;

      if  ((value / 2) * 2 == value )      rv = TRUE;
      else                                 rv = FALSE;

      return(rv);
}
```

To understand this function, as a side note one must understand “integer division”. When two integers are used in division, the script treats this division differently than floating point division. Integer division drops any remainder. Hence 5 / 2 is 2, whereas if one or more arguments is a floating point number like 5.0 / 2, the answer would be 2.5. When an odd integer value is divided by an integer, any fractional remainder is dropped. As a result, dividing an odd integer by 2 and then multiplying it again by 2 does not yield the original number. However, an even integer does work. In the example above, the result of integer division by 2, and then multiplication by 2 should yield either the original number or not, and hence one can determine if a number is even or odd.

The function is declared as being of type `fzrt_boolean` (its return type). In the function body a `fzrt_boolean` variable is set to `TRUE` or `FALSE`, based on the outcome of the integer calculation and comparison with the original value. The `return` statement, which is the last statement in the function, returns the content of the `rv` variable when the `is_even` function is called. For example:

```
      long  lval;
      fzrt_boolean bval;

      lval = 15;

      bval = is_even(lval);
```

Any of the types described in section 3.2.4 can be used as the return type for a function. In addition, a special type, called `void` can be used. It indicates, that the function does not return anything. In this case, using the statement `return (rv);` would result in a compilation error. If a function is declared to be of type `void`, the `return` statement must be used without any argments:

```
void  my_void_function(long value)
{
```

```
	…
	return;
}
```

Note, that the `return` statement is optional in a `void` function, but should always be the last statement for a function of any other type. Multiple `return` statements within the same function are supported, but are discouraged, as they may lead to undesirable results (described later).

The function name

Each function in a script must be given a unique FSL name. When a script is executed, **form•Z** looks for functions in that script that have a specific name. This is the hook between a script and **form•Z**. Therefore, it is necessary that scripts of a certain type have functions with the required names, the required return type, and the required function arguments. A complete list of the different script types and the required functions in each script type is given in section 3.6.

The function arguments

After the function name, the function header contains the function arguments enclosed in `()`. There can be zero or more arguments, separated by commas. A function argument can be used to pass data to the function and to return data from a function or both. An argument may only pass data in or may also return data. In these two cases the variables are declared differently. Arguments that only pass data in are declared in the following way:

type argument_name

type can be any of the types described in section 3.2.3. The argument name is an FSL name.

If an argument also returns data from a function, it must be declared like this:

`mod` *type* argument_name

The `mod` identifier, means, that the content of the argument can be modified, whereas an argument declared without `mod`, cannot be modified. This is illustrated in the example below:

```
fzrt_boolean		get_midpoint(
	fz_xyz_td		pt1,
	fz_xyz_td		pt2,
	mod fz_xyz_td	mid_pt
	)
{
	fzrt_boolean		rv;

	if ( pt1 == pt2 ) rv = FALSE;
	else
	{
		mid_pt = (pt1 + pt2) / 2.0;
		rv = TRUE;
	}

	return(rv);
}
```

The function `get_midpoint` calculates a 3d coordinate which is exactly halfway between two given 3d points. As described in section 3.2.5, a function can return a value through its return type. In this example, the function returns `FALSE`, if the two given points are identical. It returns `TRUE` if the two points are different and a midpoint is calculated. Of course, the function also needs to return the midpoint itself. Since the return type is already taken to indicate whether the two input points are identical or not, the midpoint is returned through the modifiable argument. Since the two input points are not modified by the function, `fz_xyz_td pt1` and `fz_xyz_td pt2` are declared without the `mod` identifier. However, since `mid_pt` is modified it is declared with `mod`. A call to this function would look like this:

```
fz_xyz_td         pt1,pt2,mid_pt;
fzrt_boolean      rv;

      pt1 = {100.0, 0.0, 0.0};
      pt2 = {200.0, 0.0, 0.0};

      rv = get_midpoint(pt1,pt2,mid_pt);
```

In this case, of course, `rv` will always be `TRUE`, since we set the two points explicitly to different values. However in other cases, the points may come from user input and their values may not be known. It is important to note that arguments, which are not declared with the `mod` identifier, can still be changed through the function statements. However, when the function is called, the variables passed as the function arguments will not change at the point of the function call. This is illustrated in the example below.

```
long  function1(long my_value)
{
      my_value = my_value * my_value;

      return(my_value);
}

long  function2(mod long my_value)
{
      my_value = my_value * my_value;

      return(my_value);
}
```

Both functions calculate the square of the function argument `my_value`. In the header of `function1` the argument is not modifiable. Even though the value of `my_value` is changed inside `function1`, a call to `function1` will leave the variable passed as the argument unchanged where `function1` is called:

```
long  value1,value2;

      value1 = 10;
      value2 = function1(value1);
```

After these statements are executed, `value1` will still be 10. This is different for `function2`. It declares the argument `my_value` with `mod`. Therefore the variable passed for `my_value` will be changed in a call to the function:

```
long  value1,value2;
```

```
value1 = 10;
value2 = function2(value1);
```

After these statements are executed, `value1` will now be 100.

The function body

The function body is enclosed by braces `{}`. After the opening brace `{`, there are zero or more variable declarations. In order to complete calls to **form•Z** APIs or to perform a task through the function statements, variables may be needed to hold values and data. Each variable used in a function must be declared to be of a certain type. This is described in further detail in section 3.2.6. After the variable declarations follow the function statements. Any number of statements can be executed by a function. It is good programming practice to keep the size of a function small enough to not lose sight of the task it is intended to perform. If a function becomes too large, it is easier to make logical mistakes and most likely the task it should perform could be broken up into a number of smaller tasks. It would then be better to break the single function up into a set of functions, corresponding to the smaller tasks. The types of statements that can be executed in a function are described in more detail in section 3.2.8.

3.2.5 Declarations of variables

At the beginning of a function body variables used by the function statements are declared. These may be single variables or array variables.

Declaring single variables

A variable declaration takes the form:

type variable_name;

type can be any of the types described in section 3.2.3. The variable name is an FSL name. Some examples of a simple variable declaration are shown below:

```
long            my_int;
long            a;
fz_xyz_td       pnt1;
fz_mat4x4_td    trl_mat;
```

It is also possible to declare multiple variables of one type in the same declaration, which is done as follows:

type $variable\_name_1$, $variable\_name_2$,…, $variable\_name_n$;

For example:

```
long        my_int,val,a,b;
double      fval1,fval2,fval3;
```

Note that variable names must be unique within the body of a function. (Two different functions may have variables with the same names). Their names cannot collide with each other and the names of the function arguments. When a variable is declared, its initial value is undefined. For example, the declaration `long  my_int;` leaves the value of `my_int` in an unknown state. Later on in the function body `my_int` may receive a value through an assignment statement, such as `my_int = 10;`. It is also possible to assign a value to a variable at the time it is declared, which is called **initialization**. This is done as follows:

type   variable_name = *expression*;

Expressions are described in further detail in the next section. A few examples of variable initialization in the declaration statement are shown below:

```
long              my_int1 = 0, my_int2 = 1, my_int3 = -999;
double            fval = FZ_PI, fval2 = FZ_PI * 0.5;
fz_xyz_td         pnt = {0.0,0.0,0.0};
fzrt_boolean      rv = TRUE;
```

Declaring arrays of variables

A variable declared as discussed above has only space for one value to be stored. In contrast, arrays are sets of variables that can store sets of values of the same type. These are declared as follows:

type   variable_name[*integer literal constant*];
or
type   variable_name[];

The square brackets after the variable name indicates that the variable is an array. Inside the brackets, an optional integer number may be included. This integer must be larger than 0. It indicates the initial size of the array. For example, if it is known that an integer array will be used in the function to store 5 values, it can be declared and used in the following way:

```
long  my_array[5];

my_array[0] = 0;
my_array[1] = 0;
my_array[2] = 1;
my_array[3] = 1;
my_array[4] = 2;
```

In the assignment statements after the declaration, each member of the array is accessed by indexing one of the five positions. This is done by using an integer number in the square brackets to address one specific array location. It is very important to note that the indexing of an array is zero based. That is, the first position in the array is accessed through the index 0. Therefore, the maximum position in an array of size *n* that can be addressed is index *n – 1*. It is not necessary to include the size of the array in the variable declaration. The square brackets may remain empty at declaration time. When array members are accessed later on through function statements, **form•Z** will increase the size of the array automatically to the largest index used by the statements. For example:

```
long  my_array[];

my_array[0] = 0;
my_array[10] = 2;
my_array[5] = 1;
```

In this case, `my_array` was declared without including a specific size. After the three assignment statements, `my_array` will be of size 11, since the largest index used is 10. Likewise, if an array is declared with a specific size, it is OK to later use an index larger than the declaration size. **form•Z** will again increase the array size automatically. The index of an array member does not have to be a literal integer, as in the array declaration, but can be a variable of even an expression. For example, all members of an array can easily be initialized in a simple `for` loop, as follows:

```
long  i,my_array[5];

for(i = 0; i < 5; i++) my_array[i] = 0;
```

For those that have experience with C or other programming languages, FSL only has one-dimensional arrays. If multi-dimensional arrays are required for the implementation of a certain task, then it will have to be done as a plugin rather than a script.

Global Variables

Variables declared outside of all functions are called global variables. They follow the same syntax as explained above, and can even be initialized to a value. Note, however, assignment statements by themselves cannot take place outside of functions. Also, global variables can be accessed in any function to assign or retrieve their values.

A script writer must take caution in using global variables. One can inadvertently make mistakes by declaring function-level variables with the same name as a global variable. In the following example, if one calls the `change_my_value` function, it does not update the global variable `my_value`, but updates the local variable `my_value` instead, which value is “lost” after the function finishes, leaving the global variable with its same value.

```
long my_value;

...

long change_my_value()
{
	long my_value;
	...

	my_value = my_value + 1;
}
```

In the context of shader scripts one can run in to potential multi-threaded or multi-processor system-dependent problems. Global variables should not be used inside the pixel shading function if one is changing the value in the global variable in that call back function.

3.2.6 Expressions

An expression consists of one or more operands. If there is more than one operand, they are separated by operators. An expression always evaluates to a single value. Expressions are not used by themselves, but become part of a statement. As discussed in section 3.2.7, expressions can be used to compute the index of an array member or to initialize a variable.

Single operand expressions

A single operand expression can be a variable, a constant value, or a function call. For example:

```
my_int
15.0
is_even(15)
{0.0, 1.0}
```

These are all single operand expressions. The `my_int` variable evaluates to whatever value the variable has at the time of its use. `15.0` is already a constant value and therefore evaluates to its floating point value "15.0". `is_even(15)` (the function shown in section 3.2.5) evaluates to the boolean value of `TRUE`. `{0.0, 1.0}` is a constant `fz_xy_td` value with the `x` member set to 0.0 and the `y` member set to 1.0.

A single operand expression can be modified by one of two preceding operators, `-` and `!`.

The `-` operator negates the value of a numeric operand and can only be used with integer and floating point operands. For example:

```
-15.0
-my_int
-my_xy.x
```

The `!` operator negates a boolean operand. For example:

```
!TRUE
!my_bool
!15
```

If the `!` operator is used with an integer or floating point operand, the operand is first cast to a `fzrt_boolean`. In the example above, `!15` becomes `!TRUE` which evaluates to `FALSE`.

Expressions formed by two or more operands are separated by operators. These can be grouped in arithmetic, conditional, and assignment operators.

Arithmetic Expressions

An arithmetic expression consists of at least two operands separated by arithmetic operators. For example:

```
5 + 3
```

is an arithmetic expression. The two operands are the integer constants 3 and 5. The operator is +. The expression evaluates to 8. Each of the two operands can be a single operand expression. For example:

```
my_int – function1(15)
```

Recall the function `function1`in example in section 3.2.5, which computes and returns the square of an integer number. In the expression `my_int – function1(15)`, the left operand is a variable and the right operand is a function call. The expression evaluates to whatever value the variable `my_int` has at the time minus 225.

The arithmetic operators supported by FSL are:
`*`   multiplies the two operands

/ divides the first operand by the second
% produces the remainder when the first operand is divided by the second
+ adds the two operands
- subtracts the second operand from the first
& applies the bitwise *and* operation between the two operands
| applies the bitwise *or* operation between the two operands

The bitwise operators & and | require that the two operands be integers. If they are not, they will first be cast to integers. For more about casting see the section below. The *and* operation looks at the binary representation of the two integers. The evaluated number has a 1 in the binary location where both operands have a one and a 0 on all other locations. For example `15 & 8` become the following binary numbers: `0000 0000 0000 1111 & 0000 0000 0000 1000.`
This evaluates to `0000 0000 0000 1000` because only in the fourth location both numbers have a 1.

```
0000 0000 0000 1111
0000 0000 0000 1000
___________________
0000 0000 0000 1000
```

The result of an *or* operation has a 1 in the location where either the first or the second operand have a 1. For example `2 | 3` evaluates to 3 because:

```
0000 0000 0000 0010
0000 0000 0000 0011
___________________
0000 0000 0000 0011
```

Conditional expressions

A conditional expression consists of at least two operands separated by conditional operators. A conditional expression always evaluates to a boolean value (`TRUE` or `FALSE`). For example:

```
my_int == 1
```

The two operands are the variable `my_int` and the integer constant `1`. The operator is ==, which stands for "equal". If `my_int` has a value of `1`, the expression evaluates to `TRUE`, otherwise it evaluates to `FALSE`. As in arithmetic expressions, each of the two operands can be a single operand expression. For example:

```
is_even(15) == is_even(12)
```

In this expression, the two operands are function calls. Using the `is_even` sample function from section 3.2.5, the entire expression evaluates to `FALSE`. The first function call `is_even(15)` will return `FALSE`, and `is_even(12)` will return `TRUE`. Then the expression becomes `TRUE == FALSE`, which evaluates to `FALSE`.

The conditional operators supported by FSL are:
== evaluates the expression to `TRUE`, if operand 1 and operand 2 are equal, and to `FALSE` otherwise
!= evaluates the expression to `TRUE`, if operand 1 and operand 2 are not equal, and to `FALSE` otherwise
> evaluates the expression to `TRUE`, if operand 1 is greater than operand 2, and to `FALSE` otherwise

>= evaluates the expression to `TRUE`, if operand 1 is greater than or equal to operand 2, and to `FALSE` otherwise
< evaluates the expression to `TRUE`, if operand 1 is less than operand 2, and to `FALSE` otherwise
<= evaluates the expression to `TRUE`, if operand 1 is less than or equal to operand 2, and to `FALSE` otherwise
|| evaluates the expression to `TRUE`, if operand 1 is `TRUE` or operand 2 is `TRUE`, and to `FALSE` otherwise. Both operands are first cast to a boolean value if they are not a boolean already.
&& evaluates the expression to `TRUE`, if operand 1 is `TRUE` and operand 2 is `TRUE`, and to `FALSE` otherwise. Both operands are first cast to a boolean value if they are not a boolean already.

Assignment expressions

An assignment expression takes the form:

variable = operand

variable can be a variable of a simple or structure type or a member of an array of a simple or structure type:

array_variable[*integer expression*] = *operand*

The expression inside the square brackets must evaluate to an integer value greater than or equal to zero. The right hand side can be any expression operand, as long as its value is of the same type as the variable on the left hand side or the value can be cast to that type. After the expression on the right hand side has been evaluated, its value is assigned to the variable on the left hand side. The whole expression evaluates to the value of the variable. The variable on the left hand side cannot be a complex type, such as a `fz_mat4x4_td` or `fz_mat3x3_td`. Some examples are:

```
my_int = 1
bval = is_even(15)
my_array[0] = 1
my_array[my_int * 2] = my_array[0]
```

Another assignment expression can take the form: *variable += operand*

This is equivalent to the expression *variable = variable + operand*

For example `my_int += 2`

is the same as `my_int = my_int + 2`

This type of assignment expression also exists for the operators -. *,/, | and &.

A third type of assignment expression is called auto increment. It takes the form:

variable++ or *++variable*

This is equivalent to the expression: *variable = variable* + 1

When using *variable*++, the expression is first evaluated to the value of the variable and then 1 is added to it. Using *++variable* adds 1 to the variable's value first and then evaluates it. This kind of expression is used frequently in the `for` loop statement, described in further detail in section 3.2.8:

```
for(i = 0; i < 10; i++) my_array[i] = 0;
```

Likewise a variable can be decremented by 1 using --*variable* or *variable*--. The variables used in the auto increment or decrement expression can only be integer or floating point types.

Special care must be taken when using the assignment expression and the conditional expression with the == operator. Both look very similar, but have very different effects. Consider the following example:

```
if ( my_int == 0 )
{
        ...
}
```

in contrast to:

```
if ( my_int = 0 )
{
        ...
}
```

Both examples are valid and will not cause a compilation error. The first example compares the value of `my_int` to `0`, and if `TRUE`, executes the statements inside the if body. This is a very common piece of code. In the second example, the expression `my_int = 0` assigns the value `0` to the variable `my_int`. The expression is evaluated to the value of the variable, in this case `0`. Therefore, the if clause will never be `TRUE` and the code in the if body will never be executed. This is a very common mistake.

Expressions with more than two operands

In the previous examples, the expressions shown were composed on two operands separated by one operator. Expressions may be constructed with more than two operands, as long as they are separated by operators. For example:

```
15 - 2 + 5 - 8
```

In this case the expression is evaluated from left to right, yielding a value of 10 . When a part of an expression is enclosed in parenthesis (), the expression inside is evaluated first:

```
15 - (2 + 5) - 8
```

becomes

```
15 - 7 - 8
```

which yields 0. This grouping of expressions can be nested any number of layers deep. The inner most expression is evaluated first.

```
15 - (2 + (5 - (8 - 3)))
```

becomes

```
15 - (2 + (5 - 5))
```

becomes

```
15 - (2 + 0)
```

becomes

```
15 — 2
```

It is of course possible to mix assignment and conditional operators in the same expression. In addition, operators have a different level of priority if they are not enclosed in parenthesis. Below is a table, which lists the priority of all the expression operators, with the highest priority operators listed at the top, performing their operations before the operations listed below them.

( )					
!	++	--			
*	/	%			
+	-				
<	<=	>	>=		
==	!=				
&					
&&					
=	+=	-=	*=	/=	%=

In the examples below, the expression on the left is shown again on the right with parenthesis or evaluated operands to illustrate, which part of the expression takes priority.

```
5 + 2 * 3                       5 + (2 * 3)
!5 * ++my_int                   0 * (my_int = my_int+1)
a && b || c                     (a && b) || c
a & b < c                       a & (b < c)
i += 2 || ++j & 1 && a          (i = i + 2) || (((j = j + 1) & 1) && a)
```

Expressions can become complex and difficult to understand quite easily, as it is the case in the last of the five examples above. It is better to separate parts of an expression into individual statements, if possible. The complicated example from above can be untangled in such a way:

```
i = i + 2;
j = j + 1;
b = (j & 1) && a;
i || b
```

3.2.7 Assignment statements

Statements make up the second part of the body of a function. They perform the actual tasks. An expression, such as `my_int = 0` or `is_even(15)` becomes a statement, when it is followed by a semicolon. For example:

```
my_int = 0;
rv = is_even(15);
```

Statements of various kinds have already been used throughout this document in simple examples. A complete list of all statement types supported by FSL was included in section 3.2.1. All types are discussed in detail in the following sections.

An assignment statement is formed by using an assignment expression and adding a semicolon at the end:

```
long  i,j;

      i = 0;
      j = i + 10;
```

Assignment statements may also be formed by assigning the right hand side value to multiple variables. This takes the form:

$variable_1$ = $variable_2$ = ... $variable_n$ = *operand*;

All the variables on the left hand side of the operand must be of the same type. For example, this would assign all the variables to the same value:

```
long  a,b,c,d;

      a = b = c = d = 10;
```

3.2.8 Function calls

A function call has the syntax:

function_name($expression_1$,$expression_2$,...$expression_n$);

Calling a function that is defined in a script looks exactly like a call to an API function defined by **form•Z**. The arguments in the function definition determine what kind of expressions can be used in the function call. If a function argument of a script function is designated with a `mod` identifier (see section 3.2.5), the expression used at the same place in the function call must be a variable of the same type as the argument in the function definition. If the function is a **form•Z** API function and an argument is a pointer to a given type, the same rule applies. If the argument in a script function definition does not have a `mod` identifier or the argument in a **form•Z** API is not a pointer, any expression may be used in the function call, as long as the expression can be cast to the argument's type. Lets look at a script function definition and a **form•Z** API function definition. The function to construct a cube is called `fz_objt_cnstr_cube`. Its prototype is in the API html documentation and in the header file `fz_objt_prim_api.h`. A function prototype defines the function return type, the function name and its arguments without showing the function body. All available API functions have their prototypes defined in the API html documentation. From a function prototype, script writers can tell how the function must be called. Here is the prototype for the API function to create a cube:

```
long  fz_objt_cnstr_cube (
      long              windex,
      mod fz_xyz_td     wdh,
      mod fz_xyz_td     origin,
      mod fz_xyz_td     rotation,
      mod fz_objt_ptr   obj
      )
```

A script writer may write a custom script function to use this API function. For example, to create a cube with equal width, depth, and height, one could do the following:

```
void  create_square_cube(
            long                windex,
            fz_xyz_td           location,
            double              size,
            mod fz_objt_ptr     new_obj )
{

      fz_xyz_td   scale;

      scale.x = scale.y = scale.z = size;

      fz_objt_cnstr_cube(windex,scale,location,NULL,new_obj);

}
```

When inspecting prototypes via the .h header files one may notice that many function have a return type of `fzrt_error_td`. In a script, this type is equivalent to a `long`. All **form•Z** functions that return a `fzrt_error_td` are assumed to succeed, if the return value is `FZRT_NOERR`, which maps to the long integer value `0`. Any other return value means, that some kind of error occurred.

The `fz_objt_cnstr_cube` prototype also shows, that there are five function arguments. The first, `long windex`, is a simple integer. All the others are modifiable arguments, (denoted by an asterisk * in the header files). The API html document also contains basic information about the function, which is not apparent from the prototype. For example, the arguments `origin` and `rotation` are tagged as optional. That means, that the `NULL` pointer constant may be passed in a call to this function. **form•Z** will substitute a meaningful default value for these arguments in that case. When calling the script function `create_square_cube` from within the script, a call could look like this:

```
fz_objt_ptr new_objt;
      ...
      create_square_cube(windex,{100.0, 0.0, 0.0},50.0,new_obj);
```

Note, that the expressions used for the location and size arguments are constants. This is allowed, because these two arguments are not tagged with the `mod` identifier. Inside `create_square_cube` the **form•Z** API function is called. Since the `wdh`, `location` and `rotation` arguments are `mod` arguments, the expressions passed for the argument must be variables of the same type as the argument. The only exception is the `NULL` pointer, which can be substituted for an optional argument.

There is one exception to the matching argument type rule with **form•Z** API functions. if the prototype of an API function contains an argument of the `fz_type_td` type usually denoted by a type of `mod void`, the script may call this API function with a variable whose type may vary. Api functions which use the `fz_type_td` type are functions which set or get parameters of entities. For example, the API function `fz_objt_edit_cube_parm_set` can be called to change the parameters of an existing cube object. The same function can be called to change the height, which is a floating point parameter, as well as the origin, which is an `fz_xyz_td` parameter. The function definition in `fz_objt_prim_api.h` looks like this:

```
typedef fzrt_error_td  (FZRT_SPEC *fz_objt_edit_cube_parm_set_func) (
      long                    windex,
      fz_objt_ptr             obj,
      fz_objt_cube_parm_enum  which,
```

```
      fz_type_td *            data
      );
```

And the script prototype in the API html documentation looks like this:

```
long fz_objt_edit_cube_parm_set(
      long                    windex,
      fz_objt_ptr             obj,
      fz_objt_cube_parm_enum  which,
      mod void                data)
```

The `fz_objt_cube_parm_enum which` argument is designed to identify which cube parameter is to be changed (of varying data types). Depending on which value is used for the `which` argument, the appropriate type must be used for the `data` argument. The definition of the `fz_objt_cube_parm_enum` type in the documentation tells which type that is:

```
      FZ_OBJT_CUBE_PARM_WIDTH
            Editing - Cube width.
            Type: double
            Range: 0.0
      FZ_OBJT_CUBE_PARM_DEPTH
            Editing - Cube depth.
            Type: double
            Range: 0.0
      FZ_OBJT_CUBE_PARM_HEIGHT
            Editing - Cube height.
            Type: double
            Range: 0.0
      FZ_OBJT_CUBE_PARM_ORIGIN
            Editing - Cube origin.
            Type: fz_xyz_td
      FZ_OBJT_CUBE_PARM_ROTATION
            Editing - Cube rotation.
            Rotation angles are applied in z y x order to transform
            the cube from alignment with the world axes to it's
            3d orientation
            Type: fz_xyz_td
```

Calls to this function in a script can look like this:

```
double            height;
fz_xyz_td         origin;

      ...

      height = 10.0;
      fz_objt_edit_cube_parm_set(windex,
                                 obj,
                                 FZ_OBJT_CUBE_PARM_HEIGHT,
                                 height);

      origin = {0.0, 200.0, 5.0};
      fz_objt_edit_cube_parm_set(windex,
                                 obj,
                                 FZ_OBJT_CUBE_PARM_ORIGIN,
```

```
origin);
```

3.2.9 The `if` statement

Syntax:

```
if ( expression )
      statement
```

Or

```
if ( expression )
      statement1
else
      statement2
```

The if statement allows the script code to execute a statement based on the value of an expression. The expression inside the parenthesis after if, is evaluated and cast to a boolean, if not already a boolean. If the boolean value is TRUE, the statement following the if is executed. If the boolean value is FALSE, the statement is skipped. Alternatively, the if statement may be paired with an else clause. In this case, if the boolean value is FALSE, the statement after the else keyword is executed. Recall, that a statement may be represented by a group of statements by enclosing them in braces. An example of an if statement is shown below:

```
if ( is_even(my_int)
{
      i = i + 1;
      j = j + 1;
}
else
{
      i = i + 2;
      j = j + 2;
}
```

The `if – else` statement may be extended to a series of `if – else if – else if – else` statements, which takes the form:

```
      if ( expression1 )
            statement1
      else if ( expression2 )
            statement2
      ...
      else if ( expressionn )
            statementn
      else
            statementn+1
```

This allows for a multiple choice decision. The expressions are evaluated top to bottom. The first expression which evaluates to `TRUE`, causes the following statement to be executed, and the rest skipped. If none of the expressions evaluate to `TRUE`, the statement after the final `else` is executed. The final `else` is of course optional, in which case none of the statements would be executed if no expression evaluates to `TRUE`.

3.2.10 The `switch` statement

Syntax:

```
switch ( expression )
{
      case integer constant expression1: statement1
      case integer constant expression2: statement2
      ...
      case integer constant expressionn: statementn
      default: statementn+1
}
```

The switch statement is similar to the multiple group `if – else if – else` statement. It allows for a multiple choice decision. The expression after switch is evaluated and cast to an integer value, if not already an integer. After the switch expression follows a statement group, enclosed in braces. Although any type of statement may be placed in this group, the case and default statements matter. Zero or more case statements and the optional default statement may be placed in the switch statement group. The case keyword is followed by an integer constant. The integer constants of all case statements in a switch must be different. When the switch statement is executed, the value of the switch expression causes the execution to jump to the case statement, whose integer constant is the same as the switch expression value. The statements after that case will be executed next. Typically a break statement is inserted after the last statement of a case and before the next case. This will cause an immediate exit of the execution from the switch. An example:

```
long        my_int;
double      fval;
      ...

      switch (my_int)
      {
            case 0:
            case 1:
            case 2:
                  fval = 1.0;
            break;

            case 3:
                  fval = 20.0;
            break;

            case 4:
                  fval = 30.0;
            break;

            default:
                  fval = 0.0;
            break;
      }
```

The sample code above set the value of `fval` based on the value of `my_int`. For example, if `my_int` is `4`, the execution jumps to the statement `case 4:` and next executes `fval = 30.0;`.

The next statement is a `break` statement, which causes the execution to jump past the rest of the `switch` statements. If `my_int` is not `0,1,2,3` or `4` the statements after the `default` statement are executed. If the expression does not match any of the `case` statements and the `default` statement is not present, no statements in the `switch` are executed. Note, that in the example above, the `case 0:` and `case 1:` statements do not have a `break` statement. This allows the switch to jump to the same place, if `my_int` evaluates to `0, 1` or `2`. Omitting the `break` statement must be handled with care. It allows, as in the example above, multiple values to jump to the same location. However, if the `break` statement would be omitted by error, it causes statements to be executed that might not be intended. For example, if the `break` statement after `fval = 20.0;` is omitted by accident, the next statement that is executed would be `fval = 30.0;` which would overwrite the previous assignment of `fval`.

3.2.11 Loop statements

Loops are a language construct that allow the same statement to be executed many times, until a terminating condition is met. For example, one can create 10 cubes either by adding the code to create the cube 10 times, or the code to create a single cube can be added in a loop structure, that is executed 10 times. Three different kind of loop statements are provided by FSL: `for`, `while`, and `do while`. They are illustrated in more detail in the next three sections.

The `for` loop statement

Syntax:

```
for (expression1opt; expression2opt; expression3opt )
       statement
```

The for loop statement uses three expressions, separated by semicolons, and the actual statement that is executed by each iteration of the loop. The first expression is the starting expression. It is evaluated before the first iteration of the loop. The second expression is the terminating condition. It is evaluated before each iteration of the loop and cast to a boolean value. If the expression evaluates to TRUE, the next loop iteration will be executed. If it evaluates to FALSE, the loop stops. The third expression is evaluated before each loop iteration and before evaluating the second (terminating) expression. In a practical application, the three expression are used to initialize a loop counter, check the value of the loop counter against an upper limit and increment the loop counter. For example:

```
long                i;
fz_xyz_td           origin,scale;
fz_objt_ptr         new_obj;

      scale = {20, 20, 20};
      for(i = 0; i < 10; i++)
      {
            origin.x = i * 100.0;
            origin.y = 0.0;
            origin.z = 0.0;
            fz_objt_cnstr_cube(windex,
                               scale,
                               origin,
                               NULL,
                               new_obj);
            fz_objt_add_objt_to_project(windex,new_obj);
```

```
}
```

In this case the for loop is executed 10 times, with `i` taking on values from 0 to 9. As soon as `i` becomes 10, the terminating condition is met and the loop stops. The loop counter `i` is also used in the statements to compute a different location of the 10 cubes created. The first cube is placed at `{0.0, 0.0, 0.0}` ,the second at `{100.0, 0.0, 0.0}` etc. It is also common practice to initialize and increment more than one loop counter. For example, with the additional variable `j`:

```
scale = {20, 20, 20};
for(i = 0,j = 1; i < 10; i++, j += 2)
{
      origin.x = j * 100.0;
      origin.y = 0.0;
      origin.z = 0.0;
      fz_objt_cnstr_cube(windex,
                          scale,
                          origin,
                          NULL,
                          new_obj);
      fz_objt_add_objt_to_project(windex,new_obj);
}
```

Note that `for` loops support chained expressions, separated by commas, for the initialization and increment expressions. It is also possible, although less common to omit any of the three expressions. The example above can be written without any of the expression inside the for statement:

```
i = 0;
j = 1;
scale = {20, 20, 20};
for( ; ; )
{
      origin.x = j * 100.0;
      origin.y = 0.0;
      origin.z = 0.0;
      fz_objt_cnstr_cube(windex,
                          scale,
                          origin,
                          NULL,
                          new_obj);
      i++;
      j += 2;
      if ( i == 10 ) break;
}
```

Omitting the terminating expression assumes the condition to be permanently `TRUE`. Therefore the loop must be stopped by other means, in this case a `break` statement. Clearly, this is not the most elegant use of the `for` loop statement. Special care should be taken that loops don't turn into an infinite loop. If the break statement were to be omitted in the example above, the `for` loop would iterate forever. This may lead to severe problems. In this case, **form•Z** would eventually run out of memory, because too many cubes were created. The user may have to abnormally terminate the program to get out of this situation, permanently losing all the work unsaved up to this point.

The `while` loop statement

Syntax:

```
while (expression)
       statement
```

The while loop is very similar to the for loop statement. It executes its statements as long as its expression evaluates to TRUE. A for loop lends itself to a loop structure, that has an explicit counter, that is incremented and compared against an upper limit. The while loop tends to be more appropriate when testing against a terminating condition, such as a function call. For example:

```
      fz_lite_ptr       lite;

      fz_lite_get_next_light(windex,NULL,lite);
      while (lite != NULL)
      {
             ...
             fz_lite_get_next_light(windex,lite,lite);
      }
```

The example above loops through all the lights defined in a project. The first time **form•Z** API function `fz_lite_get_next_light` is called, the second argument, `lite`, is set to `NULL`. This retrieves the first light in the project in the third function argument. Subsequent calls pass in the previously retrieved light, which gets the next light in the list. For the last light in the list, `fz_lite_get_next_light` returns `NULL`, in which case the `while` loop stops.

The `do while` loop statement

Syntax:

```
do statement while (expression) ;
```

The do while loop is different from the for and while loops, in, that it always executes at least once. The terminating condition is checked after the statements are executed. The example below shows, how to loop through all segments of a curve of an object:

```
long        cindx,sindx,shead,snext;
fz_objt_ptr obj;

      ...
      fz_objt_curv_get_sindx(windex,obj,cindx,
                             FZ_OBJT_MODEL_TYPE_FACT,shead);
      sindx = shead;
      do
      {
             ...

             fz_objt_segt_get_next(windex,obj,sindx,
                                   FZ_OBJT_MODEL_TYPE_FACT,snext);
             sindx = snext;

      } while (sindx != shead && sindx != -1);
```

The API function call `fz_objt_curv_get_sindx` retrieves the first segment of a curve, stored in the variable `shead`. Inside the loop, the API function `fz_objt_segt_get_next` gets the next segment index `snext` from the current segment index `sindx`. Then `snext` is assigned to `sindx`. The terminating expression checks whether `sindx` and `shead` are the same. If they are, the loop has gone once around all the segments of a curve. If `sindx` becomes `–1`, the curve was an open curve, and the loop terminates as well.

A more elaborate example of a combination of `for` and `do while` loops is shown below. It traces through the topology of a **form•Z** object, visiting all the segments of all curves.

```
long        i,nface,cindx,chead,
            cnext,sindx,shead,snext;
fz_objt_ptr obj;

   ...

   fz_objt_get_face_count(windex,obj,
               FZ_OBJT_MODEL_TYPE_FACT,nface);

   for(i = 0; i < nface; i++)
   {
      fz_objt_face_get_cindx(windex,obj,i,
                    FZ_OBJT_MODEL_TYPE_FACT,cindx);
      chead = cindx;
      do
      {
         fz_objt_curv_get_sindx(windex,obj,cindx,
                     FZ_OBJT_MODEL_TYPE_FACT,shead);
         sindx = shead;
         do
         {
            ...

            fz_objt_segt_get_next(windex,obj,sindx,
                       FZ_OBJT_MODEL_TYPE_FACT,snext);

         } while ((sindx = snext) != shead && sindx != -1 );

         fz_objt_curv_get_next(windex,obj,cindx,
                      FZ_OBJT_MODEL_TYPE_FACT,cnext);

      } while ((cindx = cnext) != chead);
   }
```

3.2.12 Jump statements

There are three jump statements in FSL: `break`, `continue` and `goto`. A jump statement, when executed, forces a jump to another statement, instead of going to the next statement.

The `break` statement

Syntax:

```
break;
```

The break statement has already been partially discussed in the context of the `switch` statement. It can also be placed inside any of the three loop statements. When executed inside a loop, it forces the loop to terminate immediately without executing any further statements. For example:

```
fz_lite_ptr       lite;

      lite = NULL;
      while ( TRUE )
      {
            fz_lite_get_next_light(windex,lite,lite);
            if ( lite == NULL ) break;

            ...
      }
```

The terminating condition of the while loop is the boolean constant `TRUE`. This will cause the loop to execute forever. The loop, however, has a way of terminating by comparing the `lite` pointer against `NULL`, in which case if it is `NULL`, the `break` statement is executed. This will cause the execution of the script to jump to the statement following the loop. The example shown here is equivalent to the previous `while` loop example. Placing a `break` statement outside the context of a loop or a `switch` statement is not allowed and will cause a compile error.

The `continue` statement

Syntax:

```
continue;
```

The continue statement can only be placed inside the body of a `for`, `while` or `do while` loop. It causes the statements coming after it in the loop body to be skipped. For example:

```
scale = {20, 20, 20};
for(i = 0; i < 10; i++)
{
      if ( i == 5 ) continue;

      origin.x = i * 100.0;
      origin.y = 0.0;
      origin.z = 0.0;
      fz_objt_cnstr_cube(windex,
                         scale,
                         origin,
                         NULL,
                         new_obj);
      fz_objt_add_objt_to_project(windex,new_obj);
}
```

Placing a `continue` statement inside a do while loop may be dangerous, as the terminating expression is skipped as well. The example below would result in an infinite loop, as `i` will never get larger than 5:

```
i = 0;
do
```

```
    {
        if ( i == 5 ) continue;
        ...
    } while ((i = i + 1) < 10);
```

The `goto` **statement**

Syntax:
    `goto label;`

label:

The `goto` statement executes a jump of the program to the line that follows the label identified by the `goto` statement. The label can be placed anywhere inside a function, after the variable declarations. There can be any number of labels in a function, but they must all have unique names, and must be different from the names of any variables declared or passed into the function. The label name must begin with a lower or upper case letters or the _ character. The rest of the label name may contain letters, numbers and the _ character in any combination. The label name cannot be longer than 128 characters. An appropriate use of a `goto` statement in a function is shown below:

```
void  create_cubes(long windex)
{
      long              i,j,err;
      fz_xyz_td         origin,scale;
      fz_objt_ptr       new_obj;

      scale = {10.0,10.0,10.0};
      for(i = 0; i < 3; i++)
      {
            for(j = 0; j < 3; j++)
            {
                  if ( i != 1 && j != 1 )
                  {
                        origin.x = i * 100.0;
                        origin.y = j * 100.0;
                        origin.z = 0.0;
                        err = fz_objt_cnstr_cube(windex,
                                                 scale,
                                                 origin,
                                                 NULL,
                                                 new_obj);
                        fz_objt_add_objt_to_project(windex,new_obj);

                        if ( err != FZRT_NOERR ) goto EXIT;
                  }
            }

      }


EXIT:

      return;
}
```

It is not necessary for the label to be placed after the `goto` statement. Placing the label before the `goto` causes the statements between the label and `goto` to be executed repeatedly. This effectively creates an infinite loop. It is necessary to break this loop through some kind of terminating condition. For example:

```
void  create_cubes(long windex)
{
      long              i,err;
      fz_xyz_td         origin,scale;
      fz_objt_ptr       new_obj;

      scale = {10.0,10.0,10.0};

      i = 0;

REPEAT:
      origin.x = i * 100.0;
      origin.y = 0.0;
      origin.z = 0.0;
      err = fz_objt_cnstr_cube(windex,
                               scale,
                               origin,
                               NULL,
                               new_obj);
      i = i + 1;
      if ( i < 10 && err == FZRT_NOERR ) goto REPEAT;

}
```

It is easy to see that a `for` loop is a much more elegant solution to creating 10 cubes. `goto` statements should be used with caution as they can quickly cause a messy function structure, jumping from statement to statement. In older programming languages, such as FORTRAN, `goto` statements were necessary, as control structures, such as loops or switch statements did not exist. In the C language, upon which FSL is based, `goto` statements are usually only placed to jump from deeply nested loops and if statements to the end of a function, to stop the execution of a function when an error occurred, as shown in the example above. It has also become common practice to give `goto` labels all upper case characters to distinguish them from variables and functions, which usually contain lower case letters.

3.2.13 The `return` statement

Syntax:

`return` *expression*$_{opt}$ `;`

The `return` statement causes the function in which it is placed to return execution to the calling script code. It is usually placed as the last statement in a function. The value of the expression following the statement is returned by the function to the calling script code:

```
fzrt_boolean is_even(long value)
{
      fzrt_boolean      rv;
```

```
	if  ((value / 2) * 2 == value )      rv = TRUE;
	else                                 rv = FALSE;

	return(rv);
}
```

Any function which is declared to have a return type other than void will always return a value. The calling code may or may not use the return value. For example:

```
	fz_objt_cnstr_cube(windex,
	                   scale,
	                   origin,
	                   NULL,
	                   new_obj);
```

and

```
	err = fz_objt_cnstr_cube( windex,
	                          scale,
	                          origin,
	                          NULL,
	                          new_obj);
```

are both valid function calls. The second captures the return value in the variable `err`. If a function does not have a `return` statement, or the expression of the `return` statement is omitted, the return value of the function is arbitrary. It is good practice to always have a `return` statement with the proper expression. It is also common practice to not use multiple `return` statements in a function. While there is no error in doing so, script code is easier to trace if the function exits only in one place. For example the function below is written with one `return` statement. It creates a sphere and returns any possible errors.

```
long  create_sphere(long windex)
{
	long              err;
	fz_xyz_td         org,scale;

	fz_objt_sphr_cnstr_opts_ptr   sphr_opts;
	fz_objt_ptr       obj;
	fzrt_boolean      bval;


	org = {0.0,0.0,0.0};
	scale = {1.0, 1.0, 1.0};
	fz_objt_cnstr_sphr_opts_init(windex,sphr_opts);

	bval = TRUE;
	fz_objt_cnstr_sphr_opts_set(windex,sphr_opts,
	                  FZ_OBJT_SPHR_PARM_PARTIAL,bval);

	if((err = fz_objt_cnstr_sphr(windex,scale,org,
	                  NULL,sphr_opts,NULL,obj)) == FZRT_NOERR)
	{
		fz_objt_add_objt_to_project(windex,obj);
	}

	fz_objt_cnstr_sphr_opts_finit(windex,sphr_opts);
```

```
	return(err);
}
```

The same function could also have been written like this:

```
long	create_sphere(long windex)
{
	long			err;
	fz_xyz_td		org,scale;
	fz_objt_sphr_cnstr_opts_ptr	sphr_opts;
	fz_objt_ptr		obj;
	fzrt_boolean		bval;

	org = {0.0,0.0,0.0};
	scale = {1.0, 1.0, 1.0};
	fz_objt_cnstr_sphr_opts_init(windex,sphr_opts);

	bval = TRUE;
	fz_objt_cnstr_sphr_opts_set(windex,sphr_opts,
				FZ_OBJT_SPHR_PARM_PARTIAL,bval);

	if((err = fz_objt_cnstr_sphr(windex,scale,org,
				NULL,sphr_opts,NULL,obj)) != FZRT_NOERR)
	{
		return (err );
	}

	fz_objt_add_objt_to_project(windex,obj);

	fz_objt_cnstr_sphr_opts_finit(windex,sphr_opts);

	return(FZRT_NOERR);
}
```

The mistake in the function above is that the sphere construction options would not be properly deleted if an error occurred, because the function `fz_objt_cnstr_sphr_opts_finit` would never be executed. In the case of variables that require an init function to be executed, a corresponding finit function must also be executed. In the first version of the function, this would not be the case.

3.2.14 Comments

Comments are text that is not part of the executable part of the script. They are usually provided by script writers to enhance the readability of the code, either for the benefit of the author or for other programmers working on the same script. It is always a good idea to add comments. They help identify important parts of a script, even if they may seem trivial at first.

Comments are structured in two ways. First, the text intended to be a comment can be placed between the start marker characters /* and the end marker characters */. No spaces are allowed between / and *. The text between the start and end markers is considered comment and is ignored by the script when it is compiled and executed.

```
/* THIS IS A COMMENT */
```

```
/*    IT MAY
      even wrap
      AROUND MULTIPLE
      l     i     n     e     s
*/
```

The start and end markers cannot be nested. For example the following comment would result in a compile error:

```
/*    START OF COMMENT
/*    ANOTHER COMMENT  INSIDE A COMMENT  */
      END OF COMMENT */
```

A second way to identify a comment is to place the characters // (double slash) before the comment text. All text following // until the end of the line is considered a comment. Since there is no end marker, the comment stops at the end of the line. For example:

```
// This is a different kind of comment

// If I wrap around the line
   like this it would cause a compile error

// It is quite ok to put /* the other comment marker */ here
```

Comments may be placed anywhere in the code. For example:

```
      if ( /* is_odd(val) */ is_even(val))
      {
            ...
      }
```

is a valid use of a comment. In this case it disables unused code. However a comment cannot split a syntax keyword or variable names.

```
      if ( is/* split */_even(val))
      {
            ...
      }
```

is not correct. This will cause a compile error.

3.2.15 Mixed expressions and their rules

When evaluating an expression, certain rules apply depending on the type of the operands used. In general, when evaluating an expression, the type of the resulting value is the same as the “highest” type of the two operands. The order, low to high, for the FSL types is:

```
fzrt_boolean
long, enum
double
fz_xy_td, fzrt_point
fz_xyz_td, fz_rgb_float_td, fzrt_rect
fz_plane_equ_td
fz_mat3x3_td
```

```
fz_mat4x4_td
fz_map_plane_td
```

For example the expression:

```
15.5 + 5
```

has a floating point and an integer operand. Since `double` is higher than `long`, the expression evaluates to `double`, in this case `20.5`. Expressions involving integer and floating point numbers appear intuitive since they very much resemble our school math. FSL however also allows expressions between higher level operands, which provide a nice shortcut for certain operations. These special relationships between operands of a certain type and the operators involved are documented below:

Multiplying two matrices

Multiplying two 3 by 3 matrices evaluates to a 3 by 3 matrix, where the matrices are multiplied in the same fashion as in the math API function `math_3x3_mulitply_mat_mat`. Therefore

```
fz_mat3x3_td            mat1,mat2,mat3;
...
mat3 = mat1 * mat2;
```

is the same as

```
math_3x3_mulitply_mat_mat(mat,mat2,mat3);
```

The same is the case for 4 by 4 matrices.

Multiplying matrices and `fz_xy_td`, `fz_xyz_td`

Multiplying a 3 by 3 matrix with an `fz_xy_td` evaluates to a `fz_xy_td` value, where the matrix and the `fz_xy_td` are multiplied in the same fashion as in the math API function `math_3x3_mulitply_mat_xy`. Therefore

```
fz_mat3x3_td            mat;
fz_xy_td                pt1,pt2;
...
pt2 = pt1 * mat;
```

is the same as

```
math_3x3_mulitply_mat_xy(mat,pt1,pt2);
```

The same is the case for multiplying a 4 by 4 matrix with a `fz_xyz_td`.

Operating on two structure types

Operating on two operands of a structure type is the same as operating on each of the members individually. For example:

```
fz_xy_td    p1,p2,p3;
...
```

```
p3 = p2 * p1;
```

is the same as

```
p3.x = p1.x * p2.x;
p3.y = p1.y * p2.y;
```

When mixing structure types in the same expression, the extra fields of the higher type are defaulted. For example multiplying an `fz_xy_td` and an `fz_xyz_td` results in an `fz_xyz_td`, where the z value of the result is the same as the z value of the one `fz_xyz_td` operand. For example:

```
fz_xy_td     p1 = { 2.0, 2.0 };
fz_xyz_td    p2 = { 3.0. 3.0, 3.0 },p3;

p3 = p1 * p2;
```

results in the value `{6.0, 6.0, 3.0}` for p3;

Operating on one structure type and one simple type.

When mixing a structure type with a simple type, each field of the structure type operand is operated on with the simple type operand. For example:

```
fz_xyz_td           p1;
double              fval;

p1 *= fval;
```

is the same as

```
p1.x *= fval;
p2.x *= fval;
```

Expressions chains

Several expressions may be chained together by separating them with commas. For example:

```
a = 15 + 7, my_bool = is_even(a), j++
```

In an expression chain, the individual expressions are evaluated left to right. The whole expression chain evaluates to the value of the first expression. For example:

```
if ( is_even(a), j = 0, i = j + 1)
{
      ...
}
```

The if statement uses an expression to determine whether to execute the statements in its body. The expression chain in the example has three individual expressions. They will all be evaluated, but only `is_even(a)` will be used to determine if the `if` clause is `TRUE`. Chaining expressions is not a very common programming practice and should be avoided for code clarity. The only place where chained expressions are common is in the for loop structure (see section 3.2.8). For example:

```
for(i = 0, j = 0;  i < end;  i++, j += 2 )
{
      ...
}
```

3.2.16 Casting values

Casting is a process where a value of one type is forced to become a value of another type. Casting occurs in a number of different situations. For example, in an assignment expression, the value of the operand on the right hand side (the "from" type) is cast to the type of the variable on the left hand side (the "to" type), if the types of the two operands are different. For example:

```
long  my_int;

      my_int = 15.5;
```

When casting from a lower type to a higher type, usually the content of the value is maintained and missing information is substituted. For example when casting an fz_xy_td to an fz_xyz_td, the missing z member is defaulted to `0.0`:

```
fz_xy_td    pt_xy = {2.0, 2.0};
fz_xyz_td   pt_xyz;

      pt_xyz = pt_xy;
```

After the assignment, `pt_xyz` has the value `{2.0, 2.0, 0.0}`. When casting from a higher type to a lower type, some information loss occurs. For example when casting from a `double` to a `long`, the fractional part of the number is lost:

```
long  my_int;

      my_int = 15.5;
```

After the assignment, `my_int` has a value of `15`. Certain types can be cast to other types, where others cannot. The tables below illustrate which casts are allowed, how missing information is substituted and how extra information is lost. If a type does not show up in a table, it is not possible to cast from the "from" type to the "to" type.

Casting from a `fzrt_boolean` to:

	Default / Loss
`fzrt_boolean`	
`long`	integer becomes 0 or 1
`double`	double becomes 0.0 or 1.0
`fz_xy_td`	fz_xy_td becomes {0.0, 0.0} or {1.0, 1.0}
`fz_xyz_td`	xyz becomes {0.0, 0.0, 0.0} or {1.0, 1.0, 1.0}
`fz_rgb_float_td`	rgb becomes {0.0, 0.0, 0.0} or {1.0, 1.0, 1.0}
`fz_plane_equ_td`	the equation becomes {0.0, 0.0, 0.0, 0.0} or {1.0, 1.0, 1.0, 1.0}
`fz_mat3x3_td`	the matrix diagonal is set to all 0.0 or 1.0. All other matrix fields are set to 0.0.
`fz_mat4x4_td`	the matrix diagonal is set to all 0.0 or 1.0. All other matrix

	fields are set to 0.0.
fzrt_ptr	the pointer value is set to NULL if the boolean is FALSE, and is set to 0x00000001 if the boolean is TRUE.

Casting from a `long` to:

	Default / Loss
fzrt_boolean	the boolean becomes FALSE if the long is 0 and TRUE otherwise
long	
double	double uses the value of the long
fz_xy_td	fz_xy_td uses the value of the long in all its fields
fz_xyz_td	fz_xyz_td uses the value of the long in all its fields
fz_rgb_float_td	rgb uses the value of the long in all its fields
fz_plane_equ_td	the equation uses the value of the long in all its fields
fz_mat3x3_td	the matrix diagonal uses the value of the long. All other matrix fields are set to 0.0.
fz_mat4x4_td	the matrix diagonal uses the value of the long. All other matrix fields are set to 0.0.
fzrt_ptr	the pointer value is set to the value of the long

Casting from a `double` to:

	Default / Loss
fzrt_boolean	the boolean becomes FALSE if the double is 0.0 and TRUE otherwise
long	the integer uses the double value with the fractional part truncated. For example -15.5 become -15, 0.1 becomes 0, 999.999 becomes 999.
double	
fz_xy_td	fz_xy_td uses the value of the double in all its fields
fz_xyz_td	fz_xyz_td uses the value of the in double t in all its fields
fz_rgb_float_td	rgb uses the value of the double in all its fields
fz_plane_equ_td	the equation uses the value of the double in all its fields
fz_mat3x3_td	the matrix diagonal uses the value of the double. All other matrix fields are set to 0.0.
fz_mat4x4_td	the matrix diagonal uses the value of the double. All other matrix fields are set to 0.0.

Casting from a `fz_xy_td` to:

	Default / Loss
fzrt_boolean	the boolean becomes FALSE if the average of the x and y members of the fz_xy_td is 0.0 and TRUE otherwise.
long	the integer uses the average of the x and y members of the fz_xy_td with the fractional part truncated. For example {2.0, 5.0} becomes 3.
double	the double uses the average of the x and y members of the fz_xy_td. For example {2.0, 5.0} becomes 3.5.
fz_xy_td	
fz_xyz_td	fz_xyz_td uses the values of the x and y members if the

	fz_xy_td and sets its z value to 0.0. For example {2.0, 2.0} becomes {2.0, 2.0, 0.0}
`fz_rgb_float_td`	rgb uses the x, and y members of the fz_xy_td and sets the b member to 0.0.
`fz_plane_equ_td`	the equation uses the x, and y members of the fz_xy_td for its a and b members and sets the c and d members to 0.0.
`fz_mat3x3_td`	the matrix diagonal uses the average of the x and y members of the fz_xy_td. All other matrix fields are set to 0.0.
`fz_mat4x4_td`	the matrix diagonal uses the average of the x and y members of the fz_xy_td. All other matrix fields are set to 0.0.

Casting from a `fz_xyz_td` to:

	Default / Loss
`fzrt_boolean`	the boolean becomes FALSE if the average of the x, y and z members of the fz_xyz_td is 0.0 and TRUE otherwise.
`long`	the integer uses the average of the x, y and z members of the fz_xyz_td with the fractional part truncated. For example {2.0, 5.0, 4.0} becomes 3.
`double`	the double uses the average of the x, y and z members of the fz_xyz_td. For example {7.5, 5.0, 4.0} becomes 5.5.
`fz_xy_td`	the fz_xy_td uses the x and y member of the fz_xyz_td. The z member of the fz_xyz_td is lost.
`fz_xyz_td`	
`fz_rgb_float_td`	rgb uses the x, y and z members of the fz_xyz_td.
`fz_plane_equ_td`	the equation uses the x, y and z members of the fz_xyz_td for its a, b and c members and sets the d member to 0.0.
`fz_mat3x3_td`	the matrix diagonal uses the average of the x, y and z members of the fz_xyz_td. All other matrix fields are set to 0.0.
`fz_mat4x4_td`	the matrix diagonal uses the average of the x, y and z members of the fz_xyz_td. All other matrix fields are set to 0.0.

Casting from a `fz_rgb_float_td` to:

	Default / Loss
`fzrt_boolean`	the boolean becomes FALSE if the average of the r, g and b members of the rgb is 0.0 and TRUE otherwise.
`long`	the integer uses the average of the r, g and b members of the rgb with the fractional part truncated. For example {1.0, 1.0, 0.0} becomes 0.
`double`	the double uses the average of the r, g and b members of the rgb. For example {1.0, 1.0, 0.0} becomes 0.666666... .
`fz_xy_td`	the fz_xy_td uses the r and b member of the rgb. The b member of the rgb is lost.
`fz_xyz_td`	fz_xyz_td uses the r, g and b members of the rgb.
`fz_rgb_float_td`	
`fz_plane_equ_td`	the equation uses the r, g and b members of the rgb for its a, b and c members and sets the d member to 0.0.
`fz_mat3x3_td`	the matrix diagonal uses the average of the r, g and b members of the rgb. All other matrix fields are set to 0.0.
`fz_mat4x4_td`	the matrix diagonal uses the average of the r, g and b

	members of the rgb. All other matrix fields are set to 0.0.

Casting from a `fz_mat3x3_td` to:

	Default / Loss
`fzrt_boolean`	the boolean becomes FALSE if the average of the matrix fields is 0.0 and TRUE otherwise.
`long`	the integer uses the average of the matrix fields with the fractional part truncated.
`double`	the double uses the average of the matrix fields
`fz_xy_td`	the fz_xy_td uses the average of the matrix fields for each of its members.
`fz_xyz_td`	fz_xyz_td uses the average of the matrix fields for each of its members.
`fz_rgb_float_td`	rgb uses the average of the matrix fields for each of its members.
`fz_plane_equ_td`	the equation uses the average of the matrix fields for each of its members.
`fz_mat3x3_td`	
`fz_mat4x4_td`	the 3 by 3 matrix is copied into the upper part of the 4 by 4 matrix. The lowest row and the right most column of the 4 by 4 matrix is set to all 0.0, except for the lower right field, which is set to 1.0.

Casting from a `fz_mat4x4_td` to:

	Default / Loss
`fzrt_boolean`	the boolean becomes FALSE if the average of the matrix fields is 0.0 and TRUE otherwise.
`long`	the integer uses the average of the matrix fields with the fractional part truncated.
`double`	the double uses the average of the the average of the matrix fields
`fz_xy_td`	the fz_xy_td uses the average of the matrix fields for each of its members.
`fz_xyz_td`	fz_xyz_td uses the average of the matrix fields for each of its members.
`fz_rgb_float_td`	rgb uses the average of the matrix fields for each of its members.
`fz_plane_equ_td`	the equation uses the average of the matrix fields for each of its members.
`fz_mat3x3_td`	The upper part of the 3 by 3 matrix (2x2) is copied into the upper part of the 4 by 4 matrix. The first two fields of the last row of the 3 by 3 matrix is copied into the last row of the 4 by 4 matrix. All other fields of the 4 by 4 matrix remain initialized as in the identity 4 by 4 matrix.
`fz_mat4x4_td`	

Casting from a `fzrt_ptr` (or any of the specific pointer types) to:

	Default / Loss
`fzrt_boolean`	the boolean becomes FALSE if the pointer is NULL and TRUE otherwise.
`long`	the integer uses the pointer value. This is usually a memory address.
`fzrt_ptr`	

Casting from an array of any FSL type to:

	Default / Loss
`fzrt_boolean`	the boolean becomes FALSE if the array has no members, TRUE otherwise
`long`	the integer uses the address of the array in memory
`fzrt_ptr`	the pointer uses the address of the array in memory

Types which are not listed above as "from" types cannot be cast to any other type. Enums of different types cannot be cast to each other. For example:

```
fz_objt_model_type_enum       my_model_type;
fz_lite_type_enum             my_lite_type;

...
my_model_type = my_lite_type; /* THIS IS NOT VALID */
```

Likewise, pointers of different types cannot be cast into each other. However a pointer of a specific type can be cast from and to the generic `fzrt_ptr` type. For example:

```
fzrt_ptr          my_ptr;
fz_objt_ptr       objt;
fz_lite_ptr       lite;

...
objt        = my_ptr;   /* VALID */
my_ptr      = objt;     /* VALID */
lite        = objt;     /* THIS IS NOT VALID */
```

3.2.17 Defining constants

Syntax: `#define` `constant_name` *`value`*

In order to make source code more legible, is it helpful to give a constant value a name placeholder. For example, the square root of 2 has a value of 1.41421. Instead of repeating these numbers each time this value is used in the code, it would be more useful, to use a name instead, which represents 1.41421. This can be done with the `#define` statement. The define statement must be placed before the first use of the constant name. The best place to put it is the top of the file, before the first function is written. It is not allowed to put a `#define` statement inside a function.

```
#define      SQRT_OF_TWO 1.41421

void my_func()
{
```

```
      double      my_val;

      my_val = SQRT_OF_TWO;

      ...
}
```

Any of the types, which have a constant value can be used in a define statement:

```
#define SOME_STRING       "This is a string constant"
#define XYZ_ORIGIN        {0.0, 0.0, 0,0}
#define XY_PLANE          {0.0, 0.0, 1,0, 0.0}
#define NOT_TRUE          FALSE
```

A second advantage of using defined constant names, is that it takes only one modification of the constant definition to have the change take effect for all uses of the constant. For example, if in a revision of the code, the `SQRT_OF_TWO` constant is defined more accurately:

```
#define      SQRT_OF_TWO 1.414213562
```

the change will apply automatically everywhere the constant is used. If the value 1.41421 were used explicitly, the programmer would have to find each use of the value and apply the same change.

3.2.18 Including scripts in other scripts

Syntax: `#include "script_file_name"`

With the `#include` statement, the source code of the script identified by the script's file name is added to the script code, as if the content of the included script file were present directly in the script it is included from. This allows a developer to reuse commonly used functions, without having to retype the code. It also keeps the commonly used code in one place, so that if a change is made, all other script which include that script will be updated with the change the next time they are compiled.

A common use of the `#include` statement would be to include a script file which contains utility function, that could be used by other scripts. The `#include` statement should be placed at the top of a script. It is not allowed to be placed inside a function.The script file, which is included by other scripts, should not be tagged with the `script_type` header. For example:

utility_funcs.fsl

```
long utility_func1()
{
	...
}

long utility_func2()
{
	...
}
```

my_script.fsl

```
script_type   FZ_UTIL_PROJ_EXTS_TYPE

#include "utility_funcs.fsl"

long fz_util_cbak_proj_main(long windex)
{
	long   val1, val2;

	val1 = utility_func1();
	val2 = utility_func2();

	...

}
```

is the same as:

my_script.fsl

```
script_type   FZ_UTIL_PROJ_EXTS_TYPE

long utility_func1()
{
	...
}

long utility_func2()
{
	...
}

long fz_util_cbak_proj_main(long windex)
{
	long   val1, val2;

	val1 = utility_func1();
	val2 = utility_func2();

	...

}
```

3.3 Script File Structure

A script consists of two parts, a header and the body. The header tells **form•Z** the type of script. The body contains one or more functions.

3.3.1 Script header

The minimum script header consists of a single line of code with two key words:

```
script_type EXTENSION_TYPE_UUID
```

`script_type` must be the first non comment key word in a script, followed by the script type identifier. The identifier indicates what type the script is, which can be a color, reflection, transparency bump, depth effect or background shader. A script can also define a modeling tool, project or system command and a project or system utility. Each of these script types has a different identifier. The required and optional functions of each script type are discussed in further detail in their respective sections. If a script were a color shader script, the first line in the script source file would look like:

```
script_type FZ_SHDR_COLR_EXTS_TYPE
```

A second header identifier, called `script_debug`, is optional:

```
script_debug      boolean
```

It defines, whether a script can be executed in debug mode or not. In order to test a script, **form•Z** offers the script author to display the source code of the script as it is executed. This is also known as debugging. The author may step through the code one statement at a time, observe the content of variables and check the flow of the execution of the script. In order to turn this mode on, the script needs to be enabled for debugging. This is done with the `script_debug` identifier. If it is followed by the boolean value `TRUE`, debugging is enabled. If it is followed by `FALSE`, or the `script_debug` identifier is not present in the script header, debugging is disabled. In addition to the `script_debug` identifier, the Use Script Debugger item in the Extensions menu in **form•Z** main menu bar needs to be selected. When a script is executed, **form•Z** will stop at the first statement of each script function and bring up the debugger environment. This is described in more detail in section 3.8.2.

3.3.2 Script Body

Depending on the type of script, different functions must be implemented. Each function must have a specific name, required by the type of script and the function arguments must also match those required by the script type. For example, a RenderZone color shader script must define at least two functions. This is described in more detail in section 3.7.3, but is summarized again below. These two functions constitute the basic functionality of a shader. They are:

A function which defines the name of the shader:

```
long fz_shdr_cbak_colr_name( mod fz_string_td name, long max_len);
```

A function which gives a pixel a color:

```
fz_rgb_float_td   fz_shdr_cbak_colr_pixel();
```

In addition to these required functions, there are optional functions. If they are not defined in a script, **form•Z** will substitute a default behavior. For example, the color shader has an optional function which returns the average color, representing the multi color pattern generated by the shader:

```
fz_rgb_float_td   fz_shdr_cbak_colr_avg();
```

This function is used to determine the color with which to draw objects in rendering modes which use a single solid color, such as Wireframe or Surface Render. If this function is not defined by a color shader, **form•Z** will substitute a 50% gray. There is one optional function, which is common to all shaders, except the project and system utiltity. It is always defined as follows:

```
long fz_script_cbak_info(mod fz_string_td uuid,
                         mod fz_string_td title,
                         mod fz_string_td vendor,
                         mod long         version);
```

This is the init function for a script and is called once, when **form•Z** first loads the script, usually at startup time. While it is optional, it is recommended, that a developer who intends to distribute the script to other users implements this function. Specifically, the uuid (unique identifier) should be defined, as it will avoid collisions of two scripts with the same id. The other arguments of this function are as follows:

`title`: this string returned by the function is displayed as the title of the script in **form•Z** when opening the **Extensions** dialog.
`vendor`: this string returned by the function is displayed as the script vendor in **form•Z** when opening the **Extensions** dialog.
`version`: This is the version assigned to the script. It is used in the Extensions dialog to indicate which version of the script is loaded. It may also be used by the respective script type to keep track of the script's data when it is written to file. This is, for example, the case with shader scripts and is discussed in more detail in section 3.7.3.

A simple, complete color shader script, including the init function, would look like this:

```
script_type FZ_SHDR_COLR_EXTS_TYPE

long fz_script_cbak_info(mod fz_string_td uuid,
                         mod fz_string_td title,
                         mod fz_string_td vendor,
                         mod long         version)
{
   uuid =
   "\xb5\xd7\xfb\x7a\x0c\x28\x48\x71\x92\x71\x34\x29\xf0\x78\x1e\x29";
   title   = "Simple Black and White Checker Color Shader";
   vendor  = "auto·des·sys Inc";
   version = 0;
}

long fz_shdr_cbak_colr_name( mod fz_string_td   name, long max_len )
{
   name = "Simple Checker";
   return (FZRT_NOERR);
}

fz_rgb_float_td fz_shdr_cbak_colr_pixel()
```

```
{
   fz_rgb_float_td   color;
   fz_xy_td       st;
   double            s,t;

   fz_shdr_get_tspace_st(st);
   s = fz_shdr_saw_tooth(st.x,1.0);
   t = fz_shdr_saw_tooth(st.y,1.0);
   if (  s < 0.5 && t < 0.5 ||
         s > 0.5 && t > 0.5 ) color = {0.0, 0.0, 0.0};
   else                       color = {1.0, 1.0, 1.0};
   return(color);
}
```

Naturally, functions can be defined in a script body, which are called from within the script. These functions can have any name and any number of arguments, as long as they don't conflict with the required or optional functions defined by a specific script type or any of the API functions offered by **form•Z**.

3.4. Using form•Z API and callback functions

Much of the functionality of a script is derived from calling functions that **form•Z** provides and which execute a large variety of operations. The functions are referred to as **API** functions. A complete listing of all API functions can be found in the on line API reference (section 5.0). The API reference describes each function, its prototypes, and provides some examples for using them.

Most **form•Z** API functions are designed to be called by scripts, so that certain functionality offered by **form•Z** can be executed by a script. Likewise, a script must offer certain functions to **form•Z**, so that the functionality defined by a script can be executed by **form•Z**. These functions are referred to as **callback** functions. Depending on which kind of script is implemented, the callback functions vary. For each script type there is a fixed number of required and a fixed number of optional callback functions. In each script type, the name, return type and arguments of the callback functions are different, but must match the name, return type and arguments required by **form•Z**. For each script type the callback functions are listed and described in more detail in section 3.7.

3.5 Interface

The **form•Z** API includes support for common interface features such as dialogs, alerts, palettes, wait cursor, key cancel detection and progress bars. The **form•Z** user interface manager (FUIM) manages these interfaces. The prefix `fz_fuim_` is used for all of the FUIM API entities (functions, types, constants, etc.).

The layout of interface elements (buttons, menus, text, etc.) found in dialogs and palettes is called a **FUIM template**. The template contains the definition of the interface elements, the definition of dependencies between the elements, and the connection to data storage (variables) in the extension. The **form•Z** template manager handles the graphic layout of the template automatically and deals with all platform specific issues. The template definition is hierarchically organized in the form of a tree. That is, each element has a parent element and may have multiple sibling elements and child elements. The interface elements are implicitly dependent on their parent. That is, if the parent element is disabled, all of its descendents are also disabled.

Templates are defined through a **FUIM template function** that is provided to **form•Z** by the extension. The template function defines the template by calling **form•Z** API functions to create the interface elements, define relationships between items, and bind the data storage (variables) from the extension to the elements. The template function is provided to **form•Z** when a dialog is invoked through a dialog driver, or through specific call back functions provided by **form•Z**. These call back functions vary by the type of extension and are discussed in section 2.7.

Note that for clarity the strings in the example in this section are shown directly in the code rather than using the recommended method of retrieving the strings from .fzr files, as described in section 1.4.2.

3.5.1 Alerts

Alerts are simple dialogs that get the user's attention by beeping and presenting information or posing questions. They are frequently used for error notification or for asking the user to make decisions at critical times. Alerts usually consist of a simple message and one or more buttons for the user to select the desired response. An icon is shown in the alert to indicate that the alert represents an error, a question or just useful information. The alert is closed when the user selects one of its buttons. A set of standard alerts is provided and custom alerts can be created using a set of functions to build and display the alert as follows:

Standard confirmation alert

```
long  fz_fuim_alrt_std_confirm(
      fz_string_td           prmt_str,
      fz_fuim_std_conf_enum  confirm_flags
      );
```

This alert contains a single prompt text string and up to two buttons. This is useful for posting a simple notification or asking a simple OK/Cancel or Yes/No question. The `prmt_str` parameter is the prompt text for the alert. The `confirm_flags` parameter indicates which buttons the alert should have as follows:

`FZ_FUIM_ALRT_CONFIRM_OK`: The alert has a single button with a title of OK.
`FZ_FUIM_ALRT_CONFIRM_OK_CANCEL`: The alert has a button with a title of OK and a button with a title of Cancel.

`FZ_FUIM_ALRT_CONFIRM_YES_NO`: The alert has a button with a title of Yes and a button with a title of No.

The alert remains on the screen until the user selects one of the buttons in the alert. The function returns `FZRT_STD_OK` if an OK or Yes button is pressed or `FZRT_STD_CANCEL` if a Cancel or No button is pressed. The following as and example of a standard confirmation alert used to ask the user if they wish to proceed with an operation.

```
long        rv;

rv = fz_fuim_alrt_std_confirm(
            "Are you sure you want to proceed?",
            FZ_FUIM_ALRT_CONFIRM_OK_CANCEL);

if(rv == FZRT_STD_OK)
{
      /* perform action here */
}
```

Standard name alert

```
long fz_fuim_alrt_std_name (
      fz_string_td            prmt_str,
      fz_string_td            name,
      long                    max_len
      );
```

This alert contains a single prompt text string, an editable name text field and the standard OK and Cancel buttons. This is useful for asking the user for simple text input. The `prmt_str` parameter is the prompt text for the alert. The `name` parameter is the string shown in the edit field. This parameter contains the desired default or current value for the name string. When the dialog is dismissed, this parameter contains the string that was entered in the text field. The `max_len` parameter is the length of the name string (in bytes). The alert remains on the screen until the user selects one of the buttons in the alert. The function returns `FZRT_STD_OK` if the OK button is pressed or `FZRT_STD_CANCEL` if the Cancel button is pressed. The following as an example of a standard name alert used to change an object name for a given object (`obj`) of a project window (`windex`);.

```
long              rv;
fz_string_td      name;

if(fz_objt_attr_get_objt_name (windex, obj, name) == FZRT_NOERR)
{     rv = fz_fuim_alrt_std_name (
            "New object name:",
            name,
            256);

      if(rv == FZRT_STD_OK)
      {
            fz_objt_attr_set_objt_name(windex, obj, name);
      }
}
```

Standard error alert

```
fzrt_boolean fz_fuim_alrt_std_error(
      long              err_id,
      long              where_id,
      fz_string_td      where_str
);
```

This alert is used for displaying error messages. This is used for posting error messages returned from **form•Z** API functions or errors in an extension that registered the error with the `fzrt_error_set` function. **form•Z** will post error messages for extensions that return errors from their call back functions, however, there are times where it may be desirable for an error alert to be displayed from an extension directly.

The alert contains a single prompt text string and the standard OK button. The `err_id` parameter is the error value returned from a **form•Z** API function or `fzrt_error_set` function call in an extension. The `where_id` parameter is a numeric indicator of where in the extension the error occurred. Each call to the `fz_fuim_alrt_std_error` function should have a unique numeric value in this parameter so that the location in the extension code where the error occurred can be identified. The `where_str` is an optional parameter that complements `where_id`. This string can be used to give additional details of where in the extension the error occurred.). The alert remains on the screen until the user selects the OK button in the alert.

```
err = fz_objt_attr_set_objt_name(windex, obj, name);

if(err != FZRT_NOERR)
{     fz_fuim_alrt_std_error(err,   1,
             "Attempting to change name");
}
```

Custom alerts

Custom alerts are constructed by initializing an alert pointer, then adding prompt text item(s) and button item(s). The alert is then displayed to the user and disposed when it is closed. The alert remains on the screen until the user selects one of the buttons in the alert.

Custom alert initialization

```
long  fz_fuim_alrt_ptr_init (
      mod fz_fuim_alrt_ptr    fuim_alrt,
      fz_fuim_alrt_flag_enum  flags,
      fz_fuim_alrt_icon_enum  alrt_icon,
      fz_string_td            alrt_title
);
```

This function creates the alert pointer. The alert pointer is a form•Z opaque data structure used to manage alerts. The pointer is returned in the `fuim_alrt` parameter. The `flags` parameter indicates optional control for the display of the alert. The default value for no options is `FZ_FUIM_ALRT_FLAG_NONE`. The value `FZ_FUIM_ALRT_FLAG_BVRT` can be used to indicate that the buttons in the alert should appear vertically stacked rather than the default horizontal layout. The `alrt_icon` parameter tells form•Z which standard icon should be shown in the alert. The valid values are `FZ_FUIM_ALRT_ICON_STOP`, `FZ_FUIM_ALRT_ICON_ASK` and `FZ_FUIM_ALRT_ICON_INFO`. The `alrt_title` parameter is the text for the title of the alert. This is shown in the title bar of the alert dialog. This parameter is optional.

Custom alert strings

```
long fz_fuim_alrt_ptr_add_str(
      fz_fuim_alrt_ptr        fuim_alrt,
      long                    flags,
      fz_string_td            str
      );
```

This function adds a string to the alert. The `fuim_alrt` parameter is the alert pointer created by the `fz_fuim_alrt_ptr_init` function. The `flags` parameter is currently not used and should always be set to 0. The `str` parameter is the text for the string that is to be shown in the alert.

Custom alert buttons

```
long fz_fuim_alrt_ptr_add_button(
      fz_fuim_alrt_ptr              fuim_alrt,
      long                          button_id,
      fz_fuim_alrt_butn_opts_enum   button_opts,
      fz_fuim_alrt_button_enum      button_kind,
      fz_string_td                  str
      );
```

This function adds a button to the alert. The `fuim_alrt` parameter is the alert pointer created by the `fz_fuim_alrt_ptr_init` function. The `button_id` should be set to a unique numeric value for each button. This value is used to identify which button the user selects when the alert is displayed on the screen. The `button_opts` parameter indicates optional control for the button. The value `FZ_FUIM_ALRT_BUT_NONE` is used to indicate no options. The value `FZ_FUIM_ALRT_BUT_DEF` can be used to indicate that the button is the default button. The default button is the button that is selected if the return or enter key is pressed while the alert is displayed on the screen. The value `FZ_FUIM_ALRT_BUT_DEF_CANCEL` can be used to indicate that the button is the cancel button. The cancel button is the button that is selected if the escape (esc) key (or any user defined cancel key shortcut) is pressed while the alert is displayed on the screen. The `button_kind` parameter indicates what title should be used for the button. The following values are available:

`FZ_FUIM_ALRT_BUTTON_OK`: Button is named “OK”.
`FZ_FUIM_ALRT_BUTTON_CANCEL`: Button is named “Cancel”.
`FZ_FUIM_ALRT_BUTTON_YES`: Button is named “Yes”.
`FZ_FUIM_ALRT_BUTTON_NO`: Button is named “No”.
`FZ_FUIM_ALRT_BUTTON_QUIT`: Button is named “Quit”.
`FZ_FUIM_ALRT_BUTTON_CUSTOM`: The title is specified in the `str` parameter.

Custom alert display

```
long fz_fuim_alrt_driver (
      fz_fuim_alrt_ptr              fuim_alrt
      );
```

This function displays the alert on the screen. The `fuim_alrt` parameter is the alert pointer created by the `fz_fuim_alrt_ptr_init` function. The alert remains on the screen until the user selects one of the buttons in the alert. The value returned from this function is the ID of the user sleeted button. The ID is the value of the `button_id` parameter that was used to create the button with the `fz_fuim_alrt_ptr_add_button` function.

Custom alert disposal

```
void fz_fuim_alrt_ptr_finit(
      mod fz_fuim_alrt_ptr          fuim_alrt
      );
```

This function disposes the alert pointer and all memory used by the alert .

Example

The following example shows a custom alert that asks the user if they want to delete selected objects. Note that for clarity the strings in this example are shown directly rather than the preferred method of storing them in .fzr files as described in section 1.4.2.

```
fz_fuim_alrt_ptr        fuim_alrt;
long                    hit;

      /* initalize the alert */
      fz_fuim_alrt_ptr_init(fuim_alrt, FZ_FUIM_ALRT_FLAG_NONE,
                              FZ_FUIM_ALRT_ICON_STOP, NULL);

      /* add the message */
      fz_fuim_alrt_ptr_add_str(fuim_alrt, 0,
            "Are you sure you want to delete the selected objects?");

      /* add the "Delete" and "Keep" buttons */
      fz_fuim_alrt_ptr_add_button(fuim_alrt, 1,
            FZ_FUIM_ALRT_BUT_DEF,
            FZ_FUIM_ALRT_BUTTON_CUSTOM, "Delete");
      fz_fuim_alrt_ptr_add_button(fuim_alrt, 2,
            FZ_FUIM_ALRT_BUT_DEF_CANCEL,
            FZ_FUIM_ALRT_BUTTON_CUSTOM, "Keep");

      /* display the alert to the user */
      hit = fz_fuim_alrt_driver(fuim_alrt);

      /* dispose the alert */
      fz_fuim_alrt_ptr_finit(fuim_alrt);

      /* handle the users choice */
      if(hit == 1)
      {
            /* Delete  objects here */
      }
```

3.5.2 Dialogs

Dialogs are invoked by calling a dialog driver function. The driver creates the window for the dialog and calls a FUIM **template function** provided by the script to create the content of the dialog. The driver displays the dialog on the screen and the user dismisses handles user interaction with the template until the dialog.

There are two dialog driver functions that work in identical fashion. The two dialog driver variants correspond to the two variants of template functions available as described in the next section. By default the driver functions return `FZRT_STD_OK` if an OK button is pressed or `FZRT_STD_CANCEL` if a Cancel button is pressed to dismiss the dialog. The two driver functions are as follows.

```
long fz_fuim_script_run_dialog(
      fz_string_td            tmpl_func_name
      );


long fz_fuim_script_run_dialog_windex)(
      long                    windex,
      fz_string_td            tmpl_func_name
      );
```

The difference between the two is that the second function uses a project window index as the first parameter. If the content of the dialog is dependent of any kind of project data, this dialog function should be used. The other argument is the name of a template callback function. **form•Z** will call this function to construct the items in the dialog. This function can have any name, but must fit the required return type and arguments. The function definition for the callback function passed to `fz_fuim_script_run_dialog` is:

long *my_dialog_func*(fz_fuim_tmpl_ptr fuim_tmpl);

The `tmpl_ptr` parameter is an opaque pointer that is created by form•Z and used to manage the template. The template pointer parameter is used as the first parameter to all FUIM API functions. This function should return `FZRT_NOERR` if the template is successfully created. Any other return value indicates that template creation failed.

For `fz_fuim_script_run_dialog_windex` the template function is:

long *my_dialog_windex_func*(long windex, fz_fuim_tmpl_ptr fuim_tmpl);

This is the same as the first template function with the addition of the `windex` parameter. This parameter is the project window index to be used for project references in the template function. This template function variant is used when operating on project or window level data where the `windex` is needed to access project or window data. The value for `windex` supplied by the function that that is driving the template.

This dialog template callback function should first initialize the template by calling `fz_fuim_script_tmpl_init`. One or more dialog items can be created with the respective `fz_fuim_script_new_...` functions and variables can be attached to an item with the `fz_fuim_script_item_range_...` functions. The dialog callback function should return `FZRT_NOERR` if it succeeds, and an error if it does not. An example of a utility script, which posts a simple dialog, is shown below. This script is also available as source code in the Scripts/Samples/Utilities folder.

```
script_type FZ_UTIL_PROJ_EXTS_TYPE

fz_objt_model_type_enum test_model_type = FZ_OBJT_MODEL_TYPE_FACT;

long  dialog_test_function( long windex, fz_fuim_tmpl_ptr  fuim_tmpl)
```

```
{
   long  err = FZRT_NOERR;
   long  tab_gindx,g1;

   /* INIT THE TEMPALTE */
   if((err = fz_fuim_script_tmpl_init(fuim_tmpl,"Example",0,NULL,0))
           == FZRT_NOERR )
   {
      /* MAKE A TAB GROUP */
      tab_gindx = fz_fuim_script_new_tab_group(fuim_tmpl,FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE);

      /* FIRST TAB HAS THE STANDARD MODEL TYPE OPTIONS */
      g1 = fz_fuim_script_new_text_static(fuim_tmpl,tab_gindx,
                  FZ_FUIM_FLAG_NONE,"Test Options");
      fz_fuim_model_type_group(fuim_tmpl,g1,test_model_type);

      /* SECOND TAB IS THE DISPLAY RESOLUTION ATTRIBUTE */
      fz_fuim_disp_res_surf_group(fuim_tmpl, tab_gindx, NULL);

      /* THIRD TAB IS THE STATUS OF OBJECTS */
      fz_fuim_status_of_objt_group(fuim_tmpl, tab_gindx);

   }

   return(err);
}


long fz_util_cbak_proj_main(long windex)
{
   long  rv;

   rv =
fz_fuim_script_run_dialog_windex(windex,"dialog_test_function");

   return(FZRT_NOERR);
}
```

For the tool script, one of the optional callback functions is also a template function. This is described in more detail in section 3.7.4.

3.5.3 Template Function

The first function that should be called inside of a template function is `fz_fuim_script_tmpl_init`.

```
long fz_fuim_tmpl_script_init(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      fz_string_td      titl_str,
      long              tmpl_flags,
      fzrt_UUID_td      uuid,
      long              version
      );
```

This function initializes the template definition. The `fuim_tmpl` parameter is the template pointer. The `titl_str` parameter is the name of the template. For dialogs, this is the tittle that appears in the title bar of the dialog window. This parameter is not used for palettes. The `tmpl_flags` parameter is currently unused and should always be 0. The `uuid` parameter is the ID of the template. This is an optional parameter. When a UUID is provided, the **form•Z** template manager stores information about the state of the template for reuse each time the template is used. This includes remembering which tab is active for tab elements and items that are collapsed in palettes. The version parameter complements the UUID and is only used when a UUID is provided. This number informs the **form•Z** template manager what version of the template is in use. This number should be set to zero for the first implementation of a template and then increased when changes are made to the implementation of the template (i.e. elements changed, removed or added). This version change informs the temple manager that the template has changed and that it should no longer use the saved state from the previous implementation.

3.5.3.1 Element creation and variable association

Each interface element in the template is referred to as a template **item**. Items are referenced by their **ID**, which is the value returned by any of the item creation functions. All items except groups and dividers have are said to have a **value**. The value can be a **specific** numeric value or a **range** of values depending on the interface element. Items that have values can associate a script **variable** with the item. When the user changes the interface element, the associated variable is updated to the defined value.

The next section describes the common aspects of template item creation. The following section describes how variables are associated with items. The remainder of the sections describes each type of element, the function that is used to create the item and what types of association are supported.

Item creation

There is a single function for creating an item of each type of interface element. All of the creation functions return the ID of the new item. If the item can not be created, the value `FZ_FUIM_NONE` is returned. All of the item creation functions start with `fz_fuim_script_new_` and contain the following common parameters:

```
fz_fuim_tmpl_ptr  fuim_tmpl
```

The `fuim_tmpl` parameter is the template pointer.

```
long              parent
```

The `parent` parameter is the ID of the parent item of the item being created. The value `FZ_FUIM_ROOT` should be used if the item is at the top of the template's hierarchy.

```
long              flags
```

The `flags` parameter is a bit encoded parameter that specifies optional control for the item being created. These values should be combined using the bitwise or (|)operator (e.g. `FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_SMAL)`. The following values are supported:

`FZ_FUIM_FLAG_NONE`: Indicates no flags.

`FZ_FUIM_FLAG_HORZ`: Indicates that the child items of the new item should have a horizontal layout. If this is not specified, they have the default vertical layout.
`FZ_FUIM_FLAG_BRDR`: Indicates that the item should be drawn with a boarder around it.
`FZ_FUIM_FLAG_INDT`: Indicates that the item's position should be indented from the position of its parent. The indentation moves the item towards the right if it is in a vertical layout and towards the bottom if it is in a vertical layout.
`FZ_FUIM_FLAG_GFLT`: Indicates that the sibling items of the new item should have a horizontal layout next to the new item.
`FZ_FUIM_FLAG_HTOP`: Items in a horizontal layout are by default center aligned. If this value is provided, all of the child items that are in a horizontal layout will be bottom aligned. Should not be used with `FZ_FUIM_FLAG_HBOT`.
`FZ_FUIM_FLAG_HBOT`: Items in a horizontal layout are by default center aligned. If this value is provided, all of the child items in a horizontal layout will be bottom aligned. Should not be used with `FZ_FUIM_FLAG_HTOP`.
`FZ_FUIM_FLAG_VCNT`: Items in a vertical layout are by default left aligned. If this value is provided, all of the child items in a vertical layout will be center aligned. Should not be used with `FZ_FUIM_FLAG_VRGT`.
`FZ_FUIM_FLAG_VRGT`: Items in a vertical layout are by default left aligned. If this value is provided, all of the child items in a vertical layout will be right aligned. Should not be used with `FZ_FUIM_FLAG_VCNT`.
`FZ_FUIM_FLAG_SMAL`: Indicates that the item should be shown in a reduced width.
`FZ_FUIM_FLAG_EQSZ`: Indicates that all of the child item should be shown made to be the same size. The size of the largest child is calculated and all child items are set to be the same size.
`FZ_FUIM_FLAG_JRGT`: Indicates that the new item should be right justified. If this is not set then the default left justification is used.
`FZ_FUIM_FLAG_DIMM`: Indicates that the item should be shown always dimmed and inactive.
`FZ_FUIM_FLAG_FRAM`: Indicates that a boarder should be drawn around all of the child items of the new item.
`FZ_FUIM_FLAG_PASS`: This is a special flag only used by text items. It indicates that the text is a password field and it should not show the text directly. When this option is selected, the text is shown with a "*" for each character in the string.

Most of the functions also contain a `titl_str` parameter. This string is the title of the item in the template. It is recommended that the strings be stored in .fzr files and loaded from this file so that they can be localized.

Variable association

The variables can be associated value can be a specific numeric value or a range of values.
Items that have values
Unary

Specific values

Specific values are used for interface elements that are binary. That is, they only have two states: on (TRUE or 1) and off (FALSE or 0). These are check boxes and radio buttons. There are 2 functions that are used to associated a specific value. The TRUE value is supplied by the script.as a function argument. The FALSE value is any value other than the TRUE value.

```
fz_fuim_script_item_unary_bool
fz_fuim_script_item_unary_long
```

Both functions have the same parameters and work identically. Each is provided for the type of the variable that is being associated (long and boolean). For example if the script variable is a long, then the function `fz_fuim_script_item_unary_long` is used.

```
void   fz_fuim_script_item_unary_long(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              item_id,
      mod long          lval,
      long              true_value
      );
```

The `fuim_tmpl` parameter is the template pointer. The `item_id` parameter is the ID of the item that is being associated . The `lval` parameter is the script variable that is being associated. The type for this variable matches the type specified in the function name. The `true_value` parameter is the value that the variable (`lval`) must have for the element to be in its TRUE state. That is when `lval == true_value`, the items value is TRUE and when `lval != true_value`, the items value is FALSE.

Range values

Range association is used for interface elements that can represent more than a single specific value. These are menus, sliders, tabs, frames and text fields. There are 3 functions that are used to associate a specific value to an item.:

```
fz_fuim_script_item_range_long
fz_fuim_script_item_range_double
fz_fuim_script_item_range_str
```

Each variant is provided for the type of the variable that is being associated. For example if the script variable is a long, then the function `fz_fuim_script_item_range_long` is used.

```
void fz_fuim_script_item_range_long(
      fz_fuim_tmpl_ptr        fuim_tmpl,
      long                    item_id,
      mod long                lval,
      long                    min_value,
      long                    max_value,
      fz_fuim_format_int_enum format,
      long                    flags
      );
```

The `fuim_tmpl` parameter is the template pointer. The `item_id` parameter is the ID of the item that is being associated. The `lval` parameter is the pointer to the script variable that is being associated. The `min_value` parameter is the minimum value for the range and `max_value` parameter is the maximum value. The `format` parameter is used if the associated item contains a text string. There is currently only one value for this parameter (`FZ_FUIM_FORMAT_INT_DEFAULT`). The `flags` parameter can be used to add additional control as follows:

`FZ_FUIM_RANGE_NONE`: no flags (default).
`FZ_FUIM_RANGE_MIN`: Clamp input to the specified minimum value in text fields.
`FZ_FUIM_RANGE_MIN_INCL`: The specified minimum value is inclusive. If this is not set it is exclusive.
`FZ_FUIM_RANGE_MAX`: Clamp input to the specified maximum value in text fields.
`FZ_FUIM_RANGE_MAX_INCL`: The specified maximum value is inclusive. If this is not set it is exclusive.

The function used for floating point values is:

```
void fz_fuim_script_item_range_double(
      fz_fuim_tmpl_ptr              fuim_tmpl,
      long                          item_id,
      mod double                    dval,
      double                        min_value,
      double                        max_value,
      fz_fuim_format_float_enum     format,
      long                          flags
      );
```

All of the parameters are the same as the integer function except for `format`. The `format` parameter is used if the associated item contains a text string. The following are currently supported:

`FZ_FUIM_FORMAT_FLOAT_DEFAULT`: The floating-point value is displayed as a fraction, with the whole and fractional part of the number separated by a decimal point.
`FZ_FUIM_FORMAT_FLOAT_DISTANCE`: floating point value is displayed as a distance value. The formatting is determined by the setting in the Working Units dialog. For example, when English units are selected the default linear distances are displayed with the feet and inch notation.
`FZ_FUIM_FORMAT_FLOAT_ANGLE`: The floating-point value is displayed as an angle. The variable's value is expected to be in radians. The display of an angle is shown in degrees in the text field.
`FZ_FUIM_FORMAT_FLOAT_PERCENT`: The floating-point value is displayed as a percentage value. That is, the variable's value is multiplied by 100 before it is displayed in the text field. This allows a value to be stored in a variable in a normalized range (0.0 to 1.0) but display it to the user as a percentage (0.0 to 100.0).

The function for a string is:

```
void fz_fuim_script_item_range_string(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              item_id,
      mod fz_string_td  str_val,
      long              max_value
      );
```

The `fuim_tmpl` parameter is the template pointer. The `item_id` parameter is the ID of the item that is being associated . The `str_val` parameter is the string variable that is being associated. The `max_value` parameter is the maximum number of characters allowed.

Check box

```
long fz_fuim_script_new_check(
```

```
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str
      );
```

A check box is an interface element that can be in either an "on" (true/1) or "off" (false/0) state. Clicking on a check box changes its state from "on" to "off", or from "off" to "on". The title string is shown to the right of the check box graphic. Variables are associated with check box items using the `fz_fuim_script_item_unary_long` or `fz_fuim_script_item_unary_bool` functions.

The following is a example of a check box with a long value associated with it such that the check is on when the variable is 2 and off when the variable is anything else.

```
long  item;
long  my_variable;

/* Create a check box item */
item = fz_fuim_script_new_check(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Check Box");

/* Associate my_variable with the item, */
/* my_variable == 2 for check on, my_variable != 2 for off */
fz_fuim_script_item_unary_long(fuim_tmpl, item,  my_variable, 2);
```

Radio button

```
long fz_fuim_script_new_radio(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str
      );
```

Radio buttons are like checkboxes except that they are used in a set and are mutually exclusive in the set: when one is switched "on", all others in the set are switched "off". This function creates a single radio button. A set of radio buttons is defined by the creation of each button in the set and then associating them with the same variable (see next section on binding). The title string is shown to the right of the radio button graphic. Variables are associated with radio items using the `fz_fuim_script_item_unary_long` or `fz_fuim_script_item_unary_bool` functions.

The following is an example of three radio buttons with a long variable associated with them such that the radio buttons are mapped to the values of 2, 3, and 7. That is when the first button is selected, the variable is set to 2, when the second is selected the variable is set to 3 and when the third is selected the variable is set to 7.

```
long  item;
long  my_variable;

/* Create a radio button box item and variable with the item with a
value of 2 */
item = fz_fuim_script_new_radio(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Radio 1");
fz_fuim_script_item_unary_long(fuim_tmpl, item,  my_variable, 2);
```

```
/* Create a radio button box item and variable with the item with a
value of 3 */
item = fz_fuim_script_new_radio(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, “My Radio 2”);
fz_fuim_script_item_unary_long(fuim_tmpl, item,  my_variable, 3);

/* Create a radio button box item and variable with the item with a
value of 7 */
item = fz_fuim_script_new_radio(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, “My Radio 3”);
fz_fuim_script_item_unary_long(fuim_tmpl, item,  my_variable, 7);
```

Button

```
long fz_fuim_script_new_button(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str,
      fz_string_td      item_func_name
      );
```

Buttons are interface items that perform an action when they arc clicked on. The action is handled in the by the function identified by the `item_func_name` argument, as described below. The title string is shown in graphics of the button. This item can not be associated with a variable as it does not change in value.

The following is an example of a button.

```
/* global variable */
my_button_id      item;

...
      /* inside a template creating function */
      /* Create a button item */
      my_button_id = fz_fuim_script_new_button(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, “My Button”, "my_button_func");
```

With the following item function to handle the click in the button.

```
long my_button_func (
      fz_fuim_tmpl_ptr              fuim_tmpl,
      long                          item_id
 )
{
      long rv = FALSE;

      if ( item_id == my_button_id )
      {
            /* Handle hit here */

            rv = TRUE;
      }

      return(rv);
}
```

The button callback function can be used for more than one button. Each time the function is invoked by **form•Z**, the id of the button which was clicked on is passed into the function via the item_id argument. In the template setup function, the return value of `fz_fuim_script_new_button` should be stored in a global variable. Inside the button callback function, it can be compared against the item id passed in, as shown in the example above. The button callback function should return TRUE, if the hit was handled by the function and FALSE otherwise.

Static text

```
long fz_fuim_script_new_text_static(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str
      );
```

Static text items are single line strings that are used for information, labels, or titles for sub-groups in a template. The user can not change static text items. This item can not be associated with a variable as it does not change in value.

The following is an example of static text.

```
long  item;

/* Create static text item */
item = fz_fuim_script_new_text_static(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Static Text");
```

Editable text

```
long fz_fuim_script_new_text_edit(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str
      );
```

Editable text items are strings that can be changed by the user. They are used for numeric fields and string fields. If a numeric variable is associated with the edit text item, then the edit text will shown a numeric value and accept numeric input. If a character variable is associated with the edit text item, then the edit text will shown the string and accept character input. The title for the edit text is shown to the left with the editable area in a box to the right. Variables are associated with editable text items using `fz_fuim_script_item_range_...` functions.

The following is an example of editable text for a long variable with a range of 0 to 20.

```
long  item;
long  my_variable;

/* Create editable text item */
item = fz_fuim_script_new_text_edit(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Edit Text");
fz_fuim_script_item_range_long(fuim_tmpl, item, my_variable, 0, 20,
                  FZ_FUIM_FORMAT_INT_DEFAULT,
                  FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
```

```
                  FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL );
```

The following is an example of editable text for a double variable which must be greater than zero.

```
long  item;
double      my_variable;

/* Create editable text item */
item = fz_fuim_script_new_text_edit(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Edit Text");
fz_fuim_script_item_range_double(fuim_tmpl, item, my_variable, 0.0,
0.0,
                  FZ_FUIM_FORMAT_FLOAT_DEFAULT,
                  FZ_FUIM_RANGE_MIN);
```

The following is an example of editable text for a string.

```
long  item;
fz_string_td      my_string;

/* Create editable text item */
item = fz_fuim_script_new_text_edit (fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Edit Text");
fz_fuim_script_item_range_string(fuim_tmpl, item, my_string);
```

Note

```
long fz_fuim_script_new_note(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str
      );
```

A note is like a static text item except that it supports multiple lines. Note are used for detailed information. The user can not change these items. This item can not be associated with a variable as it does not change in value.

```
/* Create note item */
item = fz_fuim_script_new_note(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, "My Note");
```

Menu

```
long fz_fuim_script_new_menu (
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fzrt_boolean      is_pop,
      fz_string_td      titl_str,
      fz_string_td      menu_items[],
      long              nitems
      );
```

A menu is a list of items form which items can be selected. A menu can be a regular menu or a pop-up menu. In regular menu, the menu has one active item. The active item is shown in the template and when the item is selected, the entire menu is displayed so that a new active item can be selected. A pop-up menu is shown in the template as a small triangle. When the triangle is selected, the menu is displayed and one of the items can be selected. As there is no active item, this type of menu is useful when the selection of the item performs an action (like loading preset values) or if the menu contains a series of on/off settings and the selection of an item toggles its state.

If the `is_pop` parameter is set to `TRUE`, then the menu is a pop-up menu, and when it is set to `FALSE`, it is a regular menu. The names of the menu items are supplied via the `menu_items[]` argument, which is an array of strings. The number of items is passed via the `nitems` argument. If a menu item string is a single '-' (dash) character, the menu item is formed as a separator line, which cannot be selected.

Variables are associated with menu items using integer `fz_fuim_item_range_*` functions. The following is an example of menu variable with a range of 0 to 6. Menus are implicitly clamped at the range limits if one uses the `FZ_FUIM_RANGE_NONE` range flag. Otherwise, only the inclusive range flags are useful for menus (`FZ_FUIM_RANGE_MIN_INCL` and `FZ_FUIM_RANGE_MAX_INCL`).

```
long              item;
long              my_variable;
fz_string_td      item_names[7]

/* set menu item names */
item_names[0] = "Veggies";
item_names[1] = "Meat";
item_names[2] = "Dairy";
item_names[3] = "-";
item_names[4] = "Beer";
item_names[5] = "Juice";
item_names[6] = "Wine";

/* create menu */
item = fz_fuim_script_new_menu(fuim_tmpl, FZ_FUIM_ROOT,
                  FZ_FUIM_FLAG_NONE, “My Edit Menu”, item_names, 7);
fz_fuim_script_item_range_long(fuim_tmpl, item, my_variable, 0, 6,
                  FZ_FUIM_FORMAT_INT_DEFAULT,
                  FZ_FUIM_RANGE_NONE);
```

Slider

```
long fz_fuim_script_new_slider(
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags,
      fz_string_td      titl_str
      );
```

A slider is a graphic control useful for setting a value that has a specific range. The slider has an indicator that shows the current value of the slider. The user changes the value of the slider to the desired value by dragging the indicator. Variables are associated with slider items using either the integer or floating-point `fz_fuim_script_item_range_...` functions.

Group

```
long fz_fuim_script_new_group (
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags
      );
```

Groups are invisible items that are used to organize items. This item can not be associated with a variable since it does not change in value. To associate items within the same group, the group id should be passed as the parent id to FUIM items created after the group. An example of a useful flag for a group is one that organizes its items vertically (default) or horizontally, or puts a border around the group. Groups can be organized hierarchically as well, having a group be a parent to many child groups and other items.

Tab

```
long fz_fuim_script_new_tab_group (
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags
      );
```

A tab is used to organize information in a template into categories such that only one of the categories is shown at a given time. Each of the categories is represented by a title that is placed in a tab at the top of the interface element. The tab is a graphic that mimics the tab that would be found on a file folder. When a tab is clicked on, its contents are shown in the body of the tab interface element. This function simply creates the tab group. To construct the tab, the descendents of this item must be created in a certain fashion. Each child item of the tab item establishes an entry in the tab element. The children of the tab entries, are the contents of each tab. A long integer variable should be associated with the tab group to determine which tab is actively viewable.

Frame

```
long fz_fuim_script_new_frame_group (
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags
      );
```

A frame functions like a tab group except that it does not have any graphics. That is, there are a number of categories of information in the frame that are all displayed in the same area of the template. The selection of the active frame is driven by another interface element such as a menu or radio button. A long integer variable should be associated with the frame group to determine which frame is actively viewable.

Divider

```
long fz_fuim_script_new_divider (
      fz_fuim_tmpl_ptr  fuim_tmpl,
      long              parent,
      long              flags
```

```
);
```

A divider is a graphic line drawn across the item. By default a divider is drawn horizontally. If the value `FZ_FUIM_FLAG_HORZ` is set in the `flags` parameter, then the line is drawn vertically. This item cannot be associated with a variable as it does not change in value.

Combination items

There are a number of convenience functions that combine more than one FUIM item. Effectively, they create each of the component items, and align them in a horizontal group, and link them to the same variable. This means that when one of the items is updated the other item is updated as well. For example, a slider and edit field combo item has both a slider and an editable text field. If one were to edit the text field by supplying a new number, the slider would be updated with a new slider position and vice versa. The combination item functions are:

`fz_fuim_script_new_slider_edit_long` (slider with editable long field).
`fz_fuim_script_new_slider_edit_double` (slider with editable double field).
`fz_fuim_script_new_slider_edit_pcent_double` (slider with editable double field represented as a percentage).

The following combination functions disable the use of their edit fields when they are turned off:

`fz_fuim_script_new_check_text_edit` (check box with an editable field – use a range function to associate a variable with edit field).
`fz_fuim_script_new_radio_text_edit` (radio button with an editable field – use a range function to associate a variable with edit field).

3.5.4 Interface for time consuming tasks

Scripts that could potentially take a while to execute should implement the **wait cursor**, **key cancel**, and where possible a **progress bar**. These interface elements provide feedback to the user and allow the user to interrupt long or unintended tasks.

Wait cursor

The cursor should be changed to the wait cursor to indicate to the user when a task is being performed. On the Macintosh, this cursor is a spinning circle with alternating black and white quadrants. On Windows, the wait cursor is an animated hourglass. The function `fz_fuim_curs_wait` should be called to update the wait cursor during the processing of a task. This function takes a single parameter with the following three values:

> `FZ_FUIM_CURS_WAIT_START`: This value is used once at the start of the task. The cursor is changed to the wait cursor.
> `FZ_FUIM_CURS_WAIT_TURN`: This value is used during the processing of the task. The animated cursor is updated (turned). The function should be called with this value inside loops and other places where the flow of the extension will consume its time. Performance is not an issue with this value because the cursor is only updated every 1/4 second regardless of how frequently the function is called. Note that, if it is not called frequently enough, the cursor will appear jumpy.
> `FZ_FUIM_CURS_WAIT_END`: This value is used once at the end of a time consuming task. The cursor is changed back to the state it was in prior to the start of the task.

It is important to have exactly one start and end call so that the cursor display stays balanced. This allows for nesting of the wait cursor in a case where one time consuming extension invokes another time consuming extension.

Cancel

The user should be able to cancel any time consuming task. A script can check to see if the user has pressed the key shortcut for cancel by calling the function `fz_fuim_key_cancel`. This function returns TRUE if the cancel key shortcut has been pressed and FALSE if it has not. Note that the user can program a variety of key combinations for the cancel key shortcut using the **Shortcuts** dialog, however, extensions do not need to make any adjustments for this as it is all handled by the one function.

Progress bar

A progress bar gives the user feedback on the progress of a task. A progress bar is a small window that displays graphic and optionally descriptive textual feedback on how far a task has progressed. A progress bar is divided into stages so that task sub-portions can be identified to the user. The progress bar is updated by the extension through the use of a variable in the extension that tracks the task's progress. Loop counters are often good indication of progress through a task as shown in the example at the end of this section.

form•Z offers normal and extended styles of the progress bar as shown below. The difference between them is that the extended has much larger areas for text. Both styles have two text areas referred to as the **info** and **detail** strings. The info string is usually used to display a title for the detail string. The detail string usually is used to give some information about the task progress. In the normal progress bar the info and detail strings are short and appear next to each other. This is the style of progress bar used throughout most of **form•Z**. In the extended style, the text fields are on top of each other and they are much larger. The space for the detail string supports multiple lines. This style of progress bar is used in **form•Z** during animation generation.

There are a number of functions in the FUIM for working with progress bars. They all start with `fz_fuim_prog_`. The basic required functions for implementing a progress bar are described here and in the example at the end of the section. The remainder of the function descriptions can be found in HTML API reference (chapter 5).

The function `fz_fuim_prog_init` is called once at the start of the task to initialize the progress bar.

```
long fz_fuim_prog_init(
      long                          stages,
      fz_fuim_prog_kind_enum        kind,
      fzrt_boolean                  use_clock
      );
```

The `stages` parameter indicates how many stages the progress bar will have. There are two types of progress bars indicated by the `kind` parameter. The normal progress bar has a graphic progress indicator, a short information field and a short detail field. The expanded progress bar has a graphic progress indicator and a single line information field and a multi-line detail field. If the `use_clock` parameter is `TRUE`, then the graphic progress indicator is redrawn every 1/4 second (if there has been any progress since the last redraw). If this value is `FALSE`, then the progress bar is updated (redrawn) each time that the progress bar indicator changes. To avoid performance degradation from the progress bar, it is recommended that `TRUE` be used for this parameter.

The function fz_fuim_prog_stage_init is called to indicate the start of a task stage.

```
long  fz_fuim_prog_stage_init(
      fz_string_td       name,
      long               min,
      long               max
      );
```

The name parameter is the title of the stage that is shown in the tittle bar of the progress bar window. The min and max parameters define the range of the progress indicator during the stage. That it is, the progress indicator will move from min to max during the stage with min indicating 0% completion and max indicating 100% completion.

The function fz_fuim_prog_stage_set_current is used during the processing of a stage to update the progress bar to indicate the current progress.

```
long  fz_fuim_prog_stage_set_current(
      long         current
      );
```

The current parameter is the value of the progress indicator and must have a value between the min and max parameters used in the most recent fz_fuim_prog_stage_init function call.

The function fz_fuim_prog_stage_set_strings is used during the processing of a stage to update the info or detail strings in the progress window.

```
long fz_fuim_prog_stage_set_strings(
      fz_string_td       prog_info,
      fz_string_td       prog_detail
      );
```

The prog_info parameter is the string for the info field of the progress window. If this string is not provided, the string is not changed. The prog_detail parameter is the string for the detail field of the progress window. If this string is not provided, the string is not changed.

The function fz_fuim_prog_stage_finit should be called to indicate the completion of a stage.

```
long  fz_fuim_prog_stage_finit();
```

The function fz_fuim_prog_finit should be called to indicate the completion of the entire task. This function removes the progress bar window from the screen.

```
long  fz_fuim_prog_finit();
```

The following example shows the implementation of the wait cursor, key cancel and multi-stage progress bar in two loops of a script.

```
fzrt_boolean       canceled = FALSE;
long               i;
fz_string_td       str;
double             done;
```

```
/* start wait cursor */
fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_START);

/* initalize progress bar with 2 stages */
fz_fuim_prog_init(2, FZ_FUIM_PROG_KIND_NORMAL, TRUE);

/* start the first stage */
fz_fuim_prog_stage_init("Loop 1", 1, 100);
fz_fuim_prog_stage_set_strings("Completed:", "0 %");
for(i=1; i<=100; i++)
{
      /* do task first stage processing here */

      /* check for key cancel short cut */
      if(fz_fuim_key_cancel())
      {     canceled = TRUE;
            break;
      }
      /* update the progress bar indicator */
      fz_fuim_prog_stage_set_current(i);

      /* update the progress bar detail text */
      done = i;
      sprint_float(str,done,0,0);
      strcat(str, " %");
      fz_fuim_prog_stage_set_strings(NULL, str);

      /* update the wait cursor */
      fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_TURN);
}
/* complete the first stage */
fz_fuim_prog_stage_finit();

if(!canceled)
{
      /* start the second stage */
      fz_fuim_prog_stage_init("Loop 2", 1, 2000);
      fz_fuim_prog_stage_set_strings("Completed:", "0 %");
      for(i=1;i<=2000;i++)
      {

            /* do second stage processing here */

            /* check for key cancel short cut */
            if(fz_fuim_key_cancel())
            {     canceled = TRUE;
                  break;
            }
            /* update the progress bar indicator */
            fz_fuim_prog_stage_set_current(i);

            /* update the progress bar detail text */
            done = floor((i/2000.0) * 100.0);
            sprint_float(str,done,0,0);
            strcat(str, " %");
            fz_fuim_prog_stage_set_strings(NULL, str);
```

```
            /* update the wait cursor */
            fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_TURN);

            /* complete the second stage */
            fz_fuim_prog_stage_finit();
      }
}

/* complete the progress bar */
fz_fuim_prog_finit();

/* complete the wait cursor */
fz_fuim_curs_wait(FZ_FUIM_CURS_WAIT_END);
```

3.6 Notification

The **form•Z** notification manager is used to notify scripts when certain events occur. The events include changes in **form•Z** project data like objects, lights and layers. All scripts, except project and system utility scripts (see section 3.7.5) can receive these notifications by implementing notification callback functions. These callbacks are invoked by **form•Z** when the respective event occurs. Care should be used when implementing these functions because notification functions are called throughout **form•Z** and a poor implementation can lead to performance issues or crashes. Likewise only necessary functions should be implemented, as even empty "shell" functions will cause some performance degradation. The notification functions can be included in any script, except system and project utility scripts.

Notification call back functions

Notifications functions are implemented by giving them a specific name. They all start with `fz_notf_cbak_`. All of the functions are optional. **form•Z** will only call the functions if the script writer provides them.

The system function (optional)

```
long fz_notf_cbak_syst (
      fz_notf_syst_enum       syst_notf
      );
```

This function is called by **form•Z** when one of the actions specified by `fz_notf_syst_enum` occurs. This function is provided so that scripts can be notified when one of the actions occurs and the script can make any adjustments in reaction to the action.

```
long fz_notf_cbak_syst(
      fz_notf_syst_enum       syst_notf
      )
{
      long        err = FZRT_NOERR;


      /** Handle notification here **/

      return(err);
}
```

The project function (optional)

```
long fz_notf_cbak_proj (
      long                    windex,
      fz_notf_proj_enum       proj_notf
      );
```

This function is called by **form•Z** when one of the actions specified by `fz_notf_proj_enum` occurs in the specified project. This function will be called for each project in which the action occurs. This function is provided so that scripts can be notified when one of the actions occurs and the script can make any adjustments in reaction to the action.

```
long  fz_notf_cbak_proj (
      long                    windex,
      fz_notf_proj_enum       proj_notf
      )
```

```
{
      long        err = FZRT_NOERR;


      /** Handle project notification here **/

      return(err);
}
```

The window function (optional)

```
long fz_notf_cbak_wind (
      long                    windex,
      fz_notf_wind_enum       wind_notf,
      fz_notf_proj_enum       proj_notf
      );
```

This function is called by **form•Z** when one of the actions specified by fz_notf_wind_enum occurs in the specified project. This function will be called for each window in which the action occurs. This function is provided so that scripts can be notified when one of the actions occurs and the script can make any extension specific adjustments in reaction to the action.

This function is also called for each window in a project when a project notification happens (ie `fz_notf_cbak_proj` is called). In this situation `wind_notf == FZ_NOTF_WIND_PROJ` and `proj_notf` is the value of the project level notification.

```
long fz_notf_cbak_wind (
      long                    windex,
      fz_notf_wind_enum       wind_notf,
      fz_notf_proj_enum       proj_notf
      )
{
      long        err = FZRT_NOERR;


      /** Handle window notification here **/

      return(err);
}
```

The system units function (optional)

```
long fz_notf_cbak_syst_units (
      fz_unit_type_enum             pref_units,
      fz_unit_scale_enum            pref_scale
      );
```

This function is called when the current unit type (English/Metric) or unit scale (large/medium/small/miniture) changes. This happens when the user changes the settings in the Working Units dialog , the function `fz_proj_units_set_parm_data` is called to change the settings or when the active window is changed to a project with different Working units settings. When this notification is received, all system level (global) dimensional values should be converted to a reasonable setting for the current settings.

It is recommended that the function fz_fuim_unit_convert be used to get proper dimensional values (units and data scale) from default values for the specified pref_units and pref_scale. The fz_fuim_unit_convert function sets a double value to the current pref_units and pref_scale given an English and metric default unit values for a specified scale.

The following example establishes a default English value of 12.0 inches and a metric default value of 25 cm for the medium scale for the global variable my_distance.

```
double my_distance;

...

long fz_notf_cbak_syst_units (
      fz_unit_type_enum             pref_units,
      fz_unit_scale_enum            pref_scale
      )
{
      long        err = FZRT_NOERR;
      double      my_distance = 10;

      err = fz_fuim_unit_convert(12.0, 25.0, FZ_UNIT_SCAL_MEDIUM,
            pref_units, pref_scale, my_distance);

      return(err);
}
```

The project units function (optional)

```
long fz_notf_cbak_proj_units (
      long                          windex,
      fz_unit_type_enum             pref_units,
      fz_unit_scale_enum            pref_scale
      );
```

This function is called when the unit type (English/Metric) or unit scale (large/medium/small/miniture) for a project is changed. This happens when the user changes the settings in the Working Units dialog or the function fz_proj_units_set_parm_data is called to change the settings. When this notification is received, all project level dimensional values should be converted to a reasonable setting for the current settings.

It is recommended that the function fz_fuim_unit_convert be used to get proper dimensional values (units and data scale) from default values for the specified pref_units and pref_scale. The fz_fuim_convert_units function sets a double value to the current pref_units and pref_scale given an English and metric default unit values for a specified scale.

The following example establishes a default English value of 12.0 inches and a metric default value of 25 cm for the medium scale for the global array my_proj_distance.

```
double my_proj_distance[];

...

long fz_notf_cbak_proj_units (
```

```
      long                          windex,
      fz_unit_type_enum             pref_units,
      fz_unit_scale_enum            pref_scale
      )
{
      long        err = FZRT_NOERR;
      double      my_distance = 10;

      err = fz_fuim_unit_convert(12.0, 25.0, FZ_UNIT_SCAL_MEDIUM,
                        pref_units,pref_scale,my_proj_distance[windex])
                        ;

      return(err);
}
```

The window units function (optional)

```
long fz_notf_cbak_wind_units (
      long                    windex,
      fz_unit_type_enum       pref_units,
      fz_unit_scale_enum      pref_scale
      );
```

This function is called for each project window when the unit type (English/Metric) or unit scale (large/medium/small/miniature) for a project changes. This happens when the user changes the settings in the Working Units dialog or the function `fz_proj_units_set_parm_data` is called to change the settings. When this notification is received, all project level dimensional values should be converted to a reasonable setting for the current settings.

It is recommended that the function fz`_fuim_unit_convert` be used to get proper dimensional values (units and data scale) from default values for the specified `pref_units` and `pref_scale`. The `fz_fuim_unit_convert` function sets a double value to the current `pref_units` and `pref_scale` given an English and metric default unit values for a specified scale.

The following example establishes a default English value of 12.0 inches and a metric default value of 25 cm for the medium scale for the global array `my_wind_distance`.

```
double my_wind_distance[];

...

long fz_notf_cbak_wind_units (
      long                    windex,
      fz_unit_type_enum       pref_units,
      fz_unit_scale_enum      pref_scale
      )
{
      long        err = FZRT_NOERR;
      double      my_distance = 10;

      err = fz_fuim_unit_convert(12.0, 25.0, FZ_UNIT_SCAL_MEDIUM,
            pref_units, pref_scale, my_wind_distance[windex]);

      return(err);
}
```

The object function (optional)

```
long fz_notf_cbak_objt (
      long                          windex,
      fz_notf_objt_enum             objt_notf,
      fz_objt_ptr                   objt
      );
```

This function is called to notify that an object has changed. The `objt_`notf parameter indicates what change occurred.

```
long fz_notf_cbak_objt (
      long                          windex,
      fz_notf_objt_enum             objt_notf,
      fz_objt_ptr                   objt
      )
{
      long        err = FZRT_NOERR;


      /** Handle object notification here **/

      return(err);
}
```

The Light function (optional)

```
long fz_notf_cbak_lite (
      long                          windex,
      fz_notf_lite_enum             lite_notf,
      fz_lite_ptr                   lite
      );
```

This function is called to notify that a light has changed. The `lite_notf` parameter indicates what change occurred.

```
long fz_notf_cbak_lite (
      long                          windex,
      fz_notf_lite_enum             lite_notf,
      fz_lite_ptr                   lite
      )
{
      long        err = FZRT_NOERR;


      /** Handle light notification here **/

      return(err);
}
```

The Layer function (optional)

```
long fz_notf_cbak_layr (
      long                          windex,
      fz_notf_layr_enu              layr_notf,
      fz_layr_ptr                   layr
```

```
	);
```

This function is called to notify that an layer has changed. The `layr_notf` parameter indicates what change occurred.

```
long fz_notf_cbak_layr (
	long				windex,
	fz_notf_layr_enum		layr_notf,
	fz_layr_ptr			layr
	)
{
	long		err = FZRT_NOERR;


	/** Handle layer notification here **/

	return(err);
}
```

The view function (optional)

```
long fz_notf_cbak_view (
	long				windex,
	fz_notf_view_enum		view_notf,
	fz_view_ptr			view
	);
```

This function is called to notify that a view has changed. The `view_notf` parameter indicates what change occurred.

```
long fz_notf_cbak_view (
	long				windex,
	fz_notf_view_enum		view_notf,
	fz_view_ptr			view
	)
{
	long		err = FZRT_NOERR;


	/** Handle view notification here **/

	return(err);
}
```

The preference defaults function (optional)

```
long fz_notf_cbak_pref_default (
	fz_unit_type_enum		pref_units,
	fz_unit_scale_enum		pref_scale
	);
```

The default function is called by **form•Z** called once at startup and when a user resets the preferences to defaults in the preferences dialog. This function is provided so that scripts can establish default values for private data. All private data should be set to its default values and dimensional values should be set to the specified `pref_units` and `pref_scale`. It is recommended that the function `fz_fuim_unit_convert` should be used to get proper dimensional values (units and data scale) from default values for the specified `pref_units` and pref_scale. The fz_fuim_unit_convert function sets a double value to the current pref_units and pref_scale given English and metric default unit values for a specified scale.

```
double my_distance;

...

long fz_notf_cbak_pref_default(
      fz_unit_type_enum            pref_units,
      fz_unit_scale_enum           pref_scale
      )
{
      long        err = FZRT_NOERR;

      err = fz_fuim_unit_convert(12.0, 25.0,
                  FZ_UNIT_SCAL_MEDIUM, pref_units, pref_scale,
                  my_distance);

      return(err);
}
```

The preference model type function (optional)

```
long fz_notf_cbak_pref_model_type (
      fz_objt_model_type_enum      model_type
      );
```

The preference model type function is called by **form•Z** when the model type preference is changed. This function notifies the script to change its internal preference to facetted (FZ_OBJT_MODEL_TYPE_FACT) or smooth modeling (FZ_OBJT_MODEL_TYPE_SMOD) as indicated by the model_type parameter. This function is useful for tool scripts, which support both facetted and smooth modeling.

```
fz_objt_model_type_enum my_command_model_type;

...

long fz_notf_cbak_pref_model_type(
      fz_objt_model_type_enum      model_type
      )
{
      long        err = FZRT_NOERR;

      my_command_model_type = model_type;

      return(err);
}
```

3.7 Script Types (classes)

There are 5 types of scripts: **commands**, **palettes**, RenderZone **shaders**, **tools**, and **utilities**. Scripts are organized into types based on the functionality they provide and how they implement it. Some types of scripts are flexible and can add functionality to various areas of **form•Z**. Other types of scripts add very specific functionality to a certain area of the program. The command and utility script types are examples of more flexible scripts while the RenderZone shader script type is very specific.

There is also a distinction between system and project level scripts. System scripts are not dependent on the active window or project hence the call back functions for system scripts do not receive the active project window `windex` as a parameter. Project level scripts work on the active project window, and therefore do receive `windex` as a parameter.

3.7.1 Command Scripts

A command in **form•Z** is an action that is invoked from a menu item, icon in the command palette or a key shortcut. Command scripts are extensions that complement the **form•Z** commands and behave consistently with the **form•Z** commands. Command scripts are available in ***system*** and ***project*** levels. A system command is global in nature and does not require a project window index. These are typically utility actions for which it is desirable to have access to the utility in the **form•Z** interface. A project command requires a project or window index and are expected to operate on the project information for a provided project. Project commands are unavailable when there is no open project window.

Commands are described as **states** and **actions**. A state reflects a setting that has a specific set of selectable values (states) and a single current setting (or active state). For example, the **Show Grid** item in the **Windows** menu is a **form•Z** command that reflects the state of the grid display (on or off). When this item is selected, the state is changed and the check mark in the menu is updated to reflect the current state.

An action command is a command that performs a task when it is selected. The task is linear in nature in that **form•Z** waits for the task to be completed before anything else can be done. An action command is very flexible as virtually any **form•Z** API function can be called during the execution of the task.

There is no explicit distinction between actions and states in the **form•Z** call back functions. For a command to function properly as a state, it should implement the active function described below. This tells **form•Z** that the command in its active state and that the check mark should be drawn in the menu or the icon drawn active in the command palette.

Command script type

Command scripts are defined by tagging the script in its header with the `script_type` keyword and the proper identifier as follows:

```
script_type FZ_CMND_SYST_EXTS_TYPE
```

for a system level command script and

```
script_type FZ_CMND_PROJ_EXTS_TYPE
```

for a project level command script.

3.7.1.1 System Command

System command scripts are implemented by defining a set of callback functions. There are 13 possible callback functions. Note that some of these functions are optional hence a script would rarely implement all functions. All callback functions, if implemented, must match exactly the required name, return type and arguments as described below. As with all other script types, the system command script may implement the `fz_script_cbak_info` callback function, which defines basic information about the script. This is discussed in more detail in section 3.3.

The initialization function (optional)

```
long fz_cmnd_cbak_syst_init();
```

This function is called by **form•Z** once when the script is successfully loaded and registered. The initialization function is where the script should initialize any data that may be needed by the other functions in the function set.

```
long fz_cmnd_cbak_syst_init()
{
	long		err = FZRT_NOERR;

	/* Do initialization here */

	return(err);
}
```

The finalization function (optional)

```
long fz_cmnd_cbak_syst_finit();
```

This function is called by **form•Z** once when the script is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to any necessary cleanup.

```
long fz_cmnd_cbak_syst_finit()
{
	long		err = FZRT_NOERR;

	/* Perform cleanup here */

	return(err);
}
```

The name function (recommended)

```
long fz_cmnd_cbak_syst_name(
	mod fz_string_td	name,
	long		max_len
	);
```

This function is called by **form•Z** to get the name of the command. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the command name string is stored in a .fzr file so that it is localizable. This function is recommended for all command scripts. If this function is not provided, the name of the script file is used.

```
long fz_cmnd_cbak_syst_name(
	mod fz_string_td	name,
	long			max_len
	)
{
	long		err = FZRT_NOERR;

	/* Get the title string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, name);

	return(err);
}
```

The help function (recommended)

```
long fz_cmnd_cbak_syst_help(
	mod fz_string_td	help,
	long			max_len
	);
```

This function is called by **form•Z** to display a help string that describes the detail of what the command does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` characters. It is recommended that the command name is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 256 bytes (characters).

```
long fz_cmnd_cbak_syst_help(
	mod fz_string_td	help,
	long			max_len
	)

{
	long		err = FZRT_NOERR;

	/* Get the help string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, help);

	return(err);
}
```

The available function (recommended)

```
long fz_cmnd_cbak_syst_avail(
	mod long	rv
	);
```

This function is called by **form•Z** at various times to see if the command is available. This is useful if the command is dependent on certain conditions and it is desirable to restrict its use when the conditions are not currently satisfied. If the command is not available, then it is shown as inactive (dimmed) in the **form•Z** interface (menu, icon or palette). Key shortcuts are also disabled for the command when it is not available. If this function is not provided then the command is always available.

Availability is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is available, a value of 0 indicates that the command is unavailable.

```
long fz_cmnd_cbak_syst_avail(
	mod long	rv
	)
{
	long		err = FZRT_NOERR;

	/* return 1 for available, 0 for not available */
	rv = 1;

	return(err);
}
```

The active function (Optional)

```
long fz_cmnd_cbak_syst_active(
	mod long	rv
	);
```

This function is called by **form•Z** at various times to see if the command is active. This function is needed to implement a state command where the interface element indicates the current state. This If the command is active, then it is shown selected in the **form•Z** interface. Active commands in a menu are indicated with a check mark in front of the command name. Active commands in command palettes are indicated with a highlighted icon.

Activity is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is active, a value of 0 indicates that the command is inactive. The following example shows the active function for a state command.

```
long fz_cmnd_cbak_syst_active(
	mod long	rv
	)
{
	long		err = FZRT_NOERR;

	/* check if state is active */
	if(my_command_value1 == 1) rv = 1;
	else			   rv = 0;

	return(err);
}
```

The select function (required)

```
long fz_cmnd_cbak_syst_select();
```

This function is called by **form•Z** when an action or state command is selected from the interface (menu, icon or palette) or when a key shortcut for the command is invoked. The select function is where the real execution for the command takes place. For action commands the desired action should be performed in this function. For state commands, the state should be changed and the appropriate actions should be taken. After the select function is executed, **form•Z** will call the active function to check for active states.

Action command example:

```
long fz_cmnd_cbak_syst_select()
{
	long		err = FZRT_NOERR;

	/* perform  command action here */

	return(err);
}
```

State command example:

```
long fz_cmnd_cbak_syst_select()
{
	long		err = FZRT_NOERR;

	/* toggle state */
	my_command_value1  = !my_command_value1;

	return(err);
}
```

The menu function (Optional)

```
long fz_cmnd_cbak_syst_menu (
      fz_fuim_menu_ptr                  menu_ptr,
      fzrt_UUID_td                      extensions_uuid,
      mod fzrt_UUID_td                  group_uuid,
      mod long                          position
      );
```

This function is called by **form•Z** to add the command to the Extensions menu. System commands are grouped at the top of the extensions menu. The presence of this function places the command in the menu. If this function is not provided, then the command does not appear in the menu. Assigning values to the parameters of the function provides control over the placement of items in the menu. The name that appears in the menu is the name returned in the `fz_cmnd_cbak_syst_name` function.

A group of items can be placed into a pop-out hierarchical menu rather than in the extensions menu itself. Calling the function `fz_fuim_exts_menu` creates a pop-out menu in the extensions menu. The `menu_ptr` and `extensions_uuid` parameters provided to the `fz_cmnd_cbak_syst_menu` function are used in the creation of the pop-out menu. The UUID of the new menu should be assigned to the `group_uuid` parameter. The pop-out menu should be created in each `fz_cmnd_cbak_syst_menu` call back function for the group so that if the user has disabled one of the scripts, the menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists that would occur if all the scripts are enabled.

form•Z will group together all commands in the extensions menu that have the same `group_uuid`. That is, all `fz_cmnd_cbak_syst_menu` implemented functions that return the same `group_uuid` parameter are placed together in the extensions menu in a group separated from other items by a menu separator. The `position` parameter specifies the order of the items. The items in the group are sorted from lowest to highest position. If `position` is set to Zero, the items are placed in alphabetic order.

The following is an example of a menu function with a pop-out menu.

```
#define MY_GRUP_ID "\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

long fz_cmnd_cbak_syst_menu (
      fz_fuim_menu_ptr          menu_ptr,
      fzrt_UUID_td              extensions_uuid,
      mod fzrt_UUID_td          group_uuid,
      mod long                  position
      )
{
   long      err = FZRT_NOERR;
   fz_string_td     my_str;

   /* Get the title string “My Group” from the script’s resource file */
   if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) == FZRT_NOERR)
   {
      /* create the menu group */
      err = fz_fuim_exts_menu(menu_ptr, extensions_uuid, my_str, MY_GRUP_ID);

      if(err == FZRT_NOERR)
      {
         fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
         position = 1;
      }
   }
   return(err);
}
```

Nested menus can be created up to 3 levels of hierarchy by passing the uuid of another pop-out menu to the `fz_fuim_cmnd_new_menu` function. The following is an example of a nested pop-out menu.

```
#define MY_GRUP_ID_NEST "\x24\xf6\x35\x41\x6b\xab\x7f\xb4\xa5\x6a\xd5\xaa\x65\x36\xfb\xe0"

long fz_cmnd_cbak_syst_menu (
        fz_fuim_menu_ptr            menu_ptr,
        fzrt_UUID_td                extensions_uuid,
        mod fzrt_UUID_td            group_uuid,
        mod long                    position
        )
{
    long        err = FZRT_NOERR;
    fz_string_td        my_str;

    /* Get the title string "My Group" from the script's resource file */
    if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) == FZRT_NOERR)
    {
        /* create the menu group */
        if((err = fz_fuim_exts_menu (menu_ptr, extensions_uuid,
                    my_str, MY_GRUP_ID)) == FZRT_NOERR)
        {

            /* Get title string "My Nested Group" from the resource file */
            err = fzrt_fzr_get_string(my_rfzr_refid, 1, 3, my_str);

            if(err == FZRT_NOERR)
            {
                /* create the nested menu group */
                err = fz_fuim_exts_menu (menu_ptr, MY_GRUP_ID,
                            my_str, MY_GRUP_ID_NEST);

                if(err == FZRT_NOERR)
                {   fzrt_UUID_copy(MY_GRUP_ID_NEST, group_uuid);
                    position = 1;
                }
            }
        }
    }
    return(err);
}
```

By default menu items are enabled. The `fz_cmnd_cbak_syst_avail` function can be used to disable the command and make its menu item shown dimmed. Menu items for state commands are shown with a check mark when the `fz_cmnd_cbak_syst_active` function indicates that the state for the command is active.

The icon menu function (Optional, mutually exclusive with `fz_cmnd_cbak_syst_icon_menu_adjacent`)

```
long fz_cmnd_cbak_syst_icon_menu (
        fzrt_UUID_td                        icon_menu_uuid,
        mod fzrt_UUID_td                    group_uuid,
        mod fz_fuim_icon_group_enum         group_pos,
        mod long                            group_row,
        mod long                            group_col
        );
```

This function is called by **form•Z** to add the command to the system command icon menu palette. The presence of this function places the command in the palette. If no other parameters are set then the command will get added to a group of icons at the bottom (end) of the icon menu. Note that this only adds the position to the icon palette. The function `fz_cmnd_cbak_syst_icon_file` must be provided to add custom graphics for the icon. If it is not provided, **form•Z** uses a generic icon graphic.

The `group_uuid` parameter is assigned to all commands that should be grouped together. That is, all `fz_cmnd_cbak_syst_icon_menu` implemented functions that return the same `group_uuid` parameter are placed together in the system icon menu in the same group (pop-out tool menu). This group is added to the bottom (end) of the menu. The placement of the item in the group is controlled by the `group_pos` parameter. A value of `FZ_FUIM_ICON_GROUP_START` places the item at the start of the group and a value of `FZ_FUIM_ICON_GROUP_END` places it at the end of the group. Note that these may not always yield constant results because script load order can vary hence multiple uses of `FZ_FUIM_ICON_GROUP_END` my note build the icon palette in the expected order. When `FZ_FUIM_ICON_GROUP_CUSTOM` is selected, then the `group_row` and `group_col` parameters specify the position of the item in the tool menu group.

```
#define MY_GRUP_ID "\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

long fz_cmnd_cbak_syst_icon_menu (
       fzrt_UUID_td                    icon_menu_uuid,
       mod fzrt_UUID_td                group_uuid,
       mod fz_fuim_icon_group_enum     group_pos,
       mod long                        group_row,
       mod long                        group_col
       )
{
       long          err = FZRT_NOERR;

       fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
       group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
       group_row = 1;
       group_col = 1;

       return(err);
}
```

The function `fz_fuim_exts_icon_group` can be called to better control the group containing the set of commands. This adds the ability to name the group and insert the pop-out menu group in the existing menu groups. The icon pop-out menu can be created in each `fz_cmnd_cbak_syst_icon_menu` so that if the user has disabled one of the scripts, the icon menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists. That would occur if all the scripts are enabled. The following is an example of a pop-out menu.

```
long fz_cmnd_cbak_syst_icon_menu (
       fzrt_UUID_td                    icon_menu_uuid,
       mod fzrt_UUID_td                group_uuid,
       mod fz_fuim_icon_group_enum     group_pos,
       mod long                        group_row,
       mod long                        group_col
       )
{
       long          err = FZRT_NOERR;

       err = fz_fuim_exts_icon_group(
             "My  Group", MY_GRUP_ID, icon_menu_uuid,
             FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE,
```

```
                FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE);

        if(err = FZRT_NOERR)
        {       fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
                group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
                group_row = 1;
                group_col = 1;
        }
        return(err);
}
```

The icon menu adjacent function (Optional, mutually exclusive with `fz_cmnd_cbak_syst_icon_menu`)

```
long fz_cmnd_cbak_syst_icon_menu_adjacent(
        fzrt_UUID_td                        icon_menu_uuid,
        mod fzrt_UUID_td                    adjacent_uuid,
        mod fz_fuim_icon_adjacent_enum      where
        );
```

This function is called by **form•Z** to add the command to the command icon menu palette. It serves the same purpose as the `fz_cmnd_cbak_syst_icon_menu` function, however it specifies the location of the icon item quite differently. The location is identified by referencing another command in the icon menu. The `adjacent_uuid` parameter is the UUID of the command to which the icon should be added adjacent. The `where` parameter specifies to which side of the adjacent icon the icon should be added. The available options are `FZ_FUIM_ICON_ADJACENT_TOP, FZ_FUIM_ICON_ADJACENT_BOTTOM, FZ_FUIM_ICON_ADJACENT_LEFT, FZ_FUIM_ICON_ADJACENT_RIGHT`. The default action is specified by `FZ_FUIM_ICON_ADJACENT_DEFAULT` which currently is the same as `FZ_FUIM_ICON_ADJACENT_RIGHT`. New pop-out groups can not be created with this function. The following example adds the icon to the right of the **form•Z** save command.

```
long fz_cmnd_cbak_syst_icon_menu_adjacent(
        fzrt_UUID_td                        icon_menu_uuid,
        mod fzrt_UUID_td                    adjacent_uuid,
        mod fz_fuim_icon_adjacent_enum      where
        )
{
        long            err = FZRT_NOERR;

        fzrt_UUID_copy(CMND_SAVE, adjacent_uuid);
        where = FZ_FUIM_ICON_ADJACENT_RIGHT;

        return(err);
}
```

The icon file function (Optional)

```
long fz_cmnd_cbak_syst_icon_file (
        fz_fuim_icon_enum                   which,
        fzrt_floc_ptr                       floc,
        mod long                            hpos,
        mod long                            vpos,
        fzrt_floc_ptr                       floc_mask,
        mod long                            hpos_mask,
        mod long                            vpos_mask
        );
```

This function is called by **form•Z** to get an icon for the command from an image file. The icon image can be in any of the **form•Z** supported image file formats or a format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. **form•Z** will request an icon when the command is displayed in a tool menu using `fz_cmnd_cbak_syst_icon_menu` or `fz_cmnd_cbak_syst_icon_menu_adjacent`.

form•Z supports 3 styles of icon display. Recall that these are selectable by the user from the Icon Style menu in the Icons Customization dialog. The first two options (White and Gray) are generated from a black and white source graphic with different treatments at drawing time. The third option is generated from a color source graphic. The first two options are older icon styles that are provided for backward compatibility. The color icons became the default with v 4.0. Note that if an icon of one type or the other (or both) is not provided, then **form•Z** uses a generic icon graphic.

The `which` parameter indicates the type of source graphic icon that is needed by **form•Z**. For each type of icon source (black and white and color), there are two possible sizes. The full size icon is the size that is used in the main tool palettes and tear off tool palettes. The black and white source full size is 30 x 30 pixels and indicated by `FZ_FUIM_ICON_MONOC`. The color source is 32 x 32 pixels and indicated by `FZ_FUIM_ICON_COLOR`. The alternate size is the smaller size used for window icons that are drawn in the lower margin of the window. The alternate size for both black and white and color sources is 20 x 16 pixels and indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively.

The `floc` parameter should be filled with the file name and location of the file that contains the icon graphic. The `hpos` and `vpos` parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the script file. This makes it simple to find the file. The location of the plugin file can be retained using the `fz_script_file_get_floc` function.

The `floc_mask` parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the `floc` parameter). The icon mask defines the transparent areas of the icon. The `hpos_mask` and `vpos_mask` parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided than the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of commands by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

```
long fz_cmnd_cbak_syst_icon_file (
   fz_fuim_icon_enum   which,
   fzrt_floc_ptr       floc,
   mod long            hpos,
   mod long            vpos,
   fzrt_floc_ptr       floc_mask,
   mod long            hpos_mask,
   mod long            vpos_mask
   )
{
   long   err = FZRT_NOERR;

   switch(which)
   {
      case FZ_FUIM_ICON_MONOC:
         err = fz_script_file_get_floc(floc);
         if(err == FZRT_NOERR)
         {  err = fzrt_file_floc_set_name(floc,"my_icon_bw.tif");
            hpos = 0;
```

```
            vpos = 0;
         }
      break;

      case FZ_FUIM_ICON_COLOR:
         err = fz_script_file_get_floc(floc);
         if(err == FZRT_NOERR)
         {  err = fzrt_file_floc_set_name(floc,"my_icon_col.tif");
            hpos = 0;
            vpos = 0;
         }
      break;
   }
   return(err);
}
```

The preferences IO function (optional)

```
long fz_cmnd_cbak_syst_pref_io (
      fz_iost_ptr                     iost,
      fz_iost_dir_td_enum             dir,
      mod long                        version,
      long                            size
      );
```

form•Z calls this function to read and write any command specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the plugin data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the plugin, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_cmnd_cbak_syst_pref_io (
   fz_iost_ptr            iost,
   fz_iost_dir_td_enum            dir,
   mod long               version,
   long                   size
   )
{
   long      err = FZRT_NOERR;
```

```
	if ( dir == FZ_IOST_WRITE ) version = 1;

	err = fz_iost_one_long(iost,my_command_value1);
	if(err == FZRT_NOERR)
	{	err = fz_iost_one_long(iost,my_command_value2);
		if(err == FZRT_NOERR)
		{	err = fz_iost_one_long(iost,my_command_value3);
			if(err == FZRT_NOERR)
			{	err = fz_iost_one_long(iost,my_command_value4);

				if(version >= 1)
				{	err = fz_iost_one_long(iost,my_command_value5);
				}
			}
		}
	}

	return(err);
}
```

3.7.1.2 Project Commands

Project command scripts are implemented by defining a set of callback functions. There are 17 possible callback functions. Note that some of these functions are optional hence a script would rarely implement all functions. All callback functions, if implemented, must match exactly the required name, return type and arguments as described below. As with all other script types, the project command script may implement the `fz_script_cbak_info` callback function, which defines basic information about the script. This is discussed in more detail in section 3.3.

The initialization function (optional)

```
long fz_cmnd_cbak_proj_init();
```

This function is called by **form•Z** once when the script is successfully loaded and registered. The initialization function is where the script should initialize any data that may be needed by the other functions in the function set.

```
long fz_cmnd_cbak_proj_init()
{
		long		err = FZRT_NOERR;

		/* Do initialization here */

		return(err);
}
```

The finalization function (optional)

```
long fz_cmnd_cbak_proj_finit();
```

This function is called by **form•Z** once when the script is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to perform any necessary cleanup.

```
long fz_cmnd_cbak_proj_finit()
{
		long		err = FZRT_NOERR;

		/* Perform cleanup here */
```

```
	return(err);
}
```

The info function (required)

```
long fz_cmnd_cbak_proj_info(
	mod fz_proj_level_enum	level
	);
```

This function is called by **form•Z** once when the script is successfully loaded to determine the kind of command that is implemented by the callback functions.

The `level` parameter indicates the context of the command. **form•Z** uses the value in this parameter to determine when the command should be shown and when it should be updated. The following are the available values:

`FZ_PROJ_LEVEL_MODEL`: Indicates that the command operates on the projects modeling content (objects for example).
`FZ_PROJ_LEVEL_MODEL_WIND`: Indicates that the command operates on modeling window specific content (views for example) of modeling windows.
`FZ_PROJ_LEVEL_DRAFT`: Indicates that the command operates on the projects drafting content (elements for example).
`FZ_PROJ_LEVEL_DRAFT_WIND`: Indicates that the command operates on drafting window specific content (views for example) of drafting windows.

```
long fz_cmnd_cbak_proj_info(
	mod fz_proj_level_enum	level
	)
{
	long		err = FZRT_NOERR;

	/* indicate modeling level */
	level = FZ_PROJ_LEVEL_MODEL;

	return(err);
}
```

The name function (recommended)

```
long fz_cmnd_cbak_proj_name(
	mod fz_string_td	name,
	long			max_len
	);
```

This function is called by **form•Z** to get the name of the command. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the command name string is stored in a .fzr file so that it is localizable. This function is recommended for all command scripts. If this function is not provided, the name of the script file is used.

```
long fz_cmnd_cbak_proj_name(
	mod fz_string_td	name,
	long			max_len
	)
```

```
{
	long		err = FZRT_NOERR;
	fz_string_td my_str;

	/* Get the title string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, my_str);

	return(err);
}
```

The help function (optional)

```
long fz_cmnd_cbak_proj_help(
	mod fz_string_td	help,
	long			max_len
	);
```

This function is called by **form•Z** to display a help string that describes the detail of what the command does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a memory block (string) which can handle up to `max_len` characters. It is recommended that the command name is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 256 characters.

```
long fz_cmnd_cbak_proj_help(
	mod fz_string_td	help,
	long			max_len
	)

{
	long		err = FZRT_NOERR;

	/* Get the help string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, help);

	return(err);
}
```

The available function (optional)

```
long fz_cmnd_cbak_proj_avail(
	long			windex,
	mod long		rv
	);
```

This function is called by **form•Z** at various times to see if the command is available. This is useful if the command is dependent on certain conditions and it is desirable to restrict its use when the conditions are not currently satisfied. If the command is not available, then it is shown as inactive (dimmed) in the **form•Z** interface (menu, icon or palette). Key shortcuts are also disabled for the command when it is not available. If this function is not provided then the command is always available.

Availability is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is available, a value of 0 indicates that the command is unavailable.

```
long fz_cmnd_cbak_proj_avail(
	long			windex,
	mod long		rv
	)
{
```

```
    long          err = FZRT_NOERR;

    /* return 1 for available, 0 for not available */
    rv = 1;

    return(err);
}
```

The active function (Optional)

```
long fz_cmnd_cbak_proj_active(
    long                windex,
    mod long            rv
    );
```

This function is called by **form•Z** at various times to see if the command is active. This function is needed to implement a state command where the interface element indicates the current state. This If the command is active, then it is shown selected in the **form•Z** interface. Active commands in a menu are indicated with a check mark in front of the command name. Active commands in command palettes are indicated with a highlighted icon.

Activity is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the command is active, a value of 0 indicates that the command is inactive. The following example shows the active function for a state command.

```
long fz_cmnd_cbak_proj_active(
    long                windex,
    mod long            rv
    )
{
    long          err = FZRT_NOERR;

    /* check if state is active */
    if(my_command_value1 == 1) rv = 1;
    else                       rv = 0;

    return(err);
}
```

The select function (required)

```
long fz_cmnd_cbak_proj_select(
    long                windex
    );
```

This function is called by **form•Z** when an action or state command is selected from the interface (menu, icon or palette) or when a key shortcut for the command is invoked. The select function is where the real execution for the command takes place. For action commands the desired action should be performed in this function. For state commands, the state should be changed and the appropriate actions should be taken. After the select function is executed, **form•Z** will call the active function to check for active states.

Action command example:

```
long fz_cmnd_cbak_proj_select(
    long                windex
    )
{
    long          err = FZRT_NOERR;
```

```
        /* perform  command action here */

        return(err);
}
```

State command example:

```
long fz_cmnd_cbak_proj_select(
        long                windex
        )
{
        long          err = FZRT_NOERR;

        /* toggle state */
        my_command_value1 = !my_command_value1;

        return(err);
}
```

The menu function (Optional)

```
long fz_cmnd_cbak_proj_menu (
        fz_fuim_menu_ptr                    menu_ptr,
        fzrt_UUID_td                        extensions_uuid,
        mod fzrt_UUID_td                    group_uuid,
        mod long                            position
        );
```

This function is called by **form•Z** to add the command to the Extensions menu. Project commands are grouped at the top of the extensions menu. The presence of this function places the command in the menu. If this function is not provided, then the command does not appear in the menu. Assigning values to the parameters of the function provides control over the placement of items in the menu. The name that appears in the menu is the name returned in the `fz_cmnd_cbak_proj_name` function.

A group of items can be placed into a pop-out hierarchical menu rather than in the extensions menu itself. Calling the function `fz_fuim_exts_menu` creates a pop-out menu in the extensions menu. The `menu_ptr` and `extensions_uuid` parameters provided to the `fz_cmnd_cbak_proj_menu` function are used in the creation of the pop-out menu. The UUID of the new menu should be assigned to the `group_uuid` parameter. The pop-out menu should be created in each `fz_cmnd_cbak_proj_menu` call back function for the group so that if the user has disabled one of the scripts, the menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists. That would occur if all the scripts are enabled.

form•Z will group together all commands in the extensions menu that have the same `group_uuid`. That is, all `fz_cmnd_cbak_proj_menu` implemented functions that return the same `group_uuid` parameter are placed together in the extensions menu in a group separated from other items by a menu separator. The `position` parameter specifies the order of the items. The items in the group are sorted from lowest to highest position. If `position` is set to Zero, the items are placed in alphabetic order.

The following is an example of a menu function with a pop-out menu.

```
#define MY_GRUP_ID "\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"
```

```
long fz_cmnd_cbak_proj_menu (
      fz_fuim_menu_ptr          menu_ptr,
      fzrt_UUID_td              extensions_uuid,
      mod fzrt_UUID_td          group_uuid,
      mod long                  position
      )
{
   long      err = FZRT_NOERR;
   fz_string_td    my_str;

   /* Get the title string “My Group” from the script’s resource file */
   if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) == FZRT_NOERR)
   {
      /* create the menu group */
      err = fz_fuim_exts_menu(menu_ptr, extensions_uuid, my_str, MY_GRUP_ID);

      if(err == FZRT_NOERR)
      {
         fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
         position = 1;
      }
      return(err);
   }
}
```

Nested menus can be created up to 3 levels of hierarchy by passing the uuid of another pop-out menu to the `fuim_cmnd_new_menu` function. The following is an example of a nested pop-out menu.

```
#define MY_GRUP_ID_NEST "\x24\xf6\x35\x41\x6b\xab\x7f\xb4\xa5\x6a\xd5\xaa\x65\x36\xfb\xe0"

long fz_cmnd_cbak_proj_menu (
      fz_fuim_menu_ptr          menu_ptr,
      fzrt_UUID_td              extensions_uuid,
      mod fzrt_UUID_td          group_uuid,
      mod long                  position
      )
{
   long      err = FZRT_NOERR;
   fz_string_td    my_str;

   /* Get the title string “My Group” from the script’s resource file */
   if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, my_str)) == FZRT_NOERR)
      {
      /* create the menu group */
      if((err = fz_fuim_exts_menu ( menu_ptr, extensions_uuid,
                   my_str, MY_GRUP_ID)) == FZRT_NOERR)
      {

         /* Get title string “My Nested Group” from the resource file */
         err = fzrt_fzr_get_string(my_rfzr_refid, 1, 3, my_str);

         if(err == FZRT_NOERR)
                   {
            /* create the nested menu group */
                     err = fz_fuim_exts_menu (menu_ptr, MY_GRUP_ID,
                                 my_str, MY_GRUP_ID_NEST);

            if(err == FZRT_NOERR)
            {  fzrt_UUID_copy(MY_GRUP_ID_NEST, group_uuid);
               position = 1;
            }
         }
```

```
        }
    }
    return(err);
}
```

By default menu items are enabled. The `fz_cmnd_cbak_proj_avail` function can be used to disable the command and make its menu item shown dimmed. Menu items for state commands are shown with a check mark when the fz_`cmnd_cbak_proj_active` function indicates that the state for the command is active.

The icon menu function (Optional, mutually exclusive with `fz_cmnd_cbak_proj_icon_menu_adjacent`)

```
long fz_cmnd_cbak_proj_icon_menu (
        fzrt_UUID_td                        icon_menu_uuid,
        mod fzrt_UUID_td                    group_uuid,
        mod fz_fuim_icon_group_enum         group_pos,
        mod long                            group_row,
        mod long                            group_col
        );
```

This function is called by **form•Z** to add the command to the commands icon menu palette. The presence of this function places the command in the icon menu palette. If no other parameters are set then the command will get added to a group of icons at the bottom (end) of the icon menu. Note that this only adds the position to the tool menu. The function `fz_cmnd_cbak_proj_icon_file` must be provided to add custom graphics for the icon. If it is not provided, **form•Z** uses a generic icon graphic.

The `group_uuid` parameter is assigned to all commands that should be grouped together. That is, all `fz_cmnd_cbak_proj_icon_menu` implemented functions that return the same `group_uuid` parameter are placed together in the system icon menu in the same group (pop-out tool menu). This group is added to the bottom (end) of the menu. The placement of the item in the group is controlled by the `group_pos` parameter. A value of `FZ_FUIM_ICON_GROUP_START` places the item at the start of the group and a value of `FZ_FUIM_ICON_GROUP_END` places it at the end of the group. Note that these may not always yield constant results because plugin load order can vary hence multiple uses of `FZ_FUIM_ICON_GROUP_END` my note build the menu in the expected order. When `FZ_FUIM_ICON_GROUP_CUSTOM` is selected, then the `group_row` and `group_col` parameters specify the position of the item in the tool menu group.

```
#define MY_GRUP_ID "\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

long fz_cmnd_cbak_proj_icon_menu (
        fzrt_UUID_td                        icon_menu_uuid,
        mod fzrt_UUID_td                    group_uuid,
        mod fz_fuim_icon_group_enum         group_pos,
        mod long                            group_row,
        mod long                            group_col
        )
{
        long          err = FZRT_NOERR;

        fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
        group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
        group_row = 1;
        group_col = 1;

        return(err);
}
```

The function `fz_fuim_exts_icon_group` can be called to better control the group containing the set of commands. This adds the ability to name the group and insert the pop-out menu group in the existing menu groups. The icon pop-out menu can be created in each `fz_cmnd_cbak_proj_icon_menu` so that if the user has disabled one of the scripts, the icon menu will still be formed properly. **form•Z** ignores attempts to create a menu when the UUID already exists. That would occur if all the scripts are enabled. The following is an example of a pop-out menu.

```
long fz_cmnd_cbak_proj_icon_menu (
	fzrt_UUID_td				icon_menu_uuid,
	mod fzrt_UUID_td			group_uuid,
	mod fz_fuim_icon_group_enum	group_pos,
	mod long				group_row,
	mod long				group_col
	)
{
	long		err = FZRT_NOERR;

	err = fz_fuim_exts_icon_group(
		"My  Group", MY_GRUP_ID, icon_menu_uuid,
		FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE,
		FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE);

	if(err = FZRT_NOERR)
	{	fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
		group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
		group_row = 1;
		group_col = 1;
	}
	return(err);
}
```

The icon menu adjacent function (Optional, mutually exclusive with `fz_cmnd_cbak_proj_icon_menu`)

```
long fz_cmnd_cbak_proj_icon_menu_adjacent (
	fzrt_UUID_td				icon_menu_uuid,
	mod fzrt_UUID_td			adjacent_uuid,
	mod fz_fuim_icon_adjacent_enum	where
	);
```

This function is called by **form•Z** to add the command to the system icon menu. It serves the same purpose as the `fz_cmnd_cbak_proj_icon_menu` function, however it specifies the location of the icon item quite differently. The location is identified by referencing another command in the icon menu. The `adjacent_uuid` parameter is the UUID of the command to which the icon should be added adjacent. The `where` parameter specifies to which side of the adjacent icon the icon should be added. The available options are `FZ_FUIM_ICON_ADJACENT_TOP, FZ_FUIM_ICON_ADJACENT_BOTTOM, FZ_FUIM_ICON_ADJACENT_LEFT, FZ_FUIM_ICON_ADJACENT_RIGHT`. The default action is specified by `FZ_FUIM_ICON_ADJACENT_DEFAULT` which currently is the same as `FZ_FUIM_ICON_ADJACENT_RIGHT`. New pop-out groups can not be created with this function. The following example adds the icon to the right of the **form•Z** save command.

```
long fz_cmnd_cbak_proj_icon_menu_adjacent (
	fzrt_UUID_td				icon_menu_uuid,
	mod fzrt_UUID_td			adjacent_uuid,
```

```
        mod fz_fuim_icon_adjacent_enum    where
        )
{
        long          err = FZRT_NOERR;

        fzrt_UUID_copy(CMND_SAVE, adjacent_uuid);
        where = FZ_FUIM_ICON_ADJACENT_RIGHT;

        return(err);
}
```

The icon file function (Optional)

```
long fz_cmnd_cbak_proj_icon_file (
        fz_fuim_icon_enum                 which,
        fzrt_floc_ptr                     floc,
        mod long                          hpos,
        mod long                          vpos,
        fzrt_floc_ptr                     floc_mask,
        mod long                          hpos_mask,
        mod long                          vpos_mask
        );
```

This function is called by **form•Z** to get an icon for the command from an image file. The icon image can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. **form•Z** will request an icon when the command is displayed in a command menu using `fz_cmnd_cbak_proj_icon_menu` or `fz_cmnd_cbak_proj_icon_menu_adjacent`.

form•Z supports 3 styles of icon display. Recall that these are selectable by the user from the Icon Style menu in the Icons Customization dialog. The first two options (White and Gray) are generated from a black and white source graphic with different treatments at drawing time. The third option is generated from a color source graphic. The first two options are older icon styles that are provided for backward compatibility. The color icons became the default with v 4.0. Note that if an icon of one type or the other (or both) is not provided, then **form•Z** uses a generic icon graphic.

The `which` parameter indicates the type of source graphic icon that is needed by **form•Z**. For each type of icon source (black and white and color), there are two possible sizes. The full size icon is the size that is used in the main tool palettes and tear off tool palettes. The black and white source full size is 30 x 30 pixels and indicated by `FZ_FUIM_ICON_MONOC`. The color source is 32 x 32 pixels and indicated by `FZ_FUIM_ICON_COLOR`. The alternate size is the smaller size used for window icons that are drawn in the lower margin of the window. The alternate size for both black and white and color sources is 20 x 16 pixels and indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively.

The `floc` parameter should be filled with the file name and location of the file that contains the icon graphic. The `hpos` and `vpos` parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the script file. This makes it simple to find the file. The location of the script file can be acquired using the `fz_script_file_get_floc` function.

The `floc_mask` parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the `floc` parameter). The icon mask defines the transparent areas of the icon. The `hpos_mask` and `vpos_mask` parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided than the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of commands by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

```
long fz_cmnd_cbak_proj_icon_file (
    fz_fuim_icon_enum       which,
    fzrt_floc_ptr           floc,
    mod long                hpos,
    mod long                vpos,
    fzrt_floc_ptr           floc_mask,
    mod long                hpos_mask,
    mod long                vpos_mask
    )
{
    long   err = FZRT_NOERR;

    switch(which)
    {
        case FZ_FUIM_ICON_MONOC:
            err = fz_script_file_get_floc (floc);
            if(err == FZRT_NOERR)
        {   err = fzrt_file_floc_set_name(floc,"my_icon_bw.tif");
                hpos = 0;
                vpos = 0;
            }
        break;
        case FZ_FUIM_ICON_COLOR:
            err = fz_script_file_get_floc (floc);
            if(err == FZRT_NOERR)
            {   err = fzrt_file_floc_set_name(floc,"my_icon_col.tif");
                hpos = 0;
                vpos = 0;
            }
        break;
    }
    return(err);
}
```

The preferences IO function (optional)

```
long fz_cmnd_cbak_proj_pref_io (
        fz_iost_ptr                 iost,
        fz_iost_dir_td_enum                 dir,
        mod long                    version,
        long                        size
        );
```

form•Z calls this function to read and write any command specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the plugin data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the script that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_cmnd_cbak_proj_pref_io (
    fz_iost_ptr             iost,
    fz_iost_dir_td_enum     dir,
    mod long                version,
    long                    size
    )
{
    long      err = FZRT_NOERR;

    if ( dir == FZ_IOST_WRITE ) version = 1;

    err = fz_iost_one_long(iost,my_command_value1);
    if(err == FZRT_NOERR)
    {   err = fz_iost_one_long(iost,my_command_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_command_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_command_value4);

                if(version >= 1)
                {   err = fz_iost_one_long(iost,my_command_value5);
                }
            }
        }
    }

    return(err);
}
```

The project data IO function (optional)

```
long fz_cmnd_cbak_proj_data_io (
        long                        windex,
        fz_iost_ptr                 iost,
        fz_iost_dir_td_enum                 dir,
        mod long                    version,
        long                        size
        );
```

form•Z calls this function to read and write any command specific project data to a **form•Z** project file. This function is called once when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** project file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the script data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the script that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_cmnd_cbak_proj_data_io (
   long                   windex,
   fz_iost_ptr            iost,
   fz_iost_dir_td_enum    dir,
   mod long               version,
   long                   size
   )
{
   long      err = FZRT_NOERR;


   if ( dir == FZ_IOST_WRITE ) version = 1;

   err = fz_iost_one_long(iost,my_command_value1);
   if(err == FZRT_NOERR)
   {  err = fz_iost_one_long(iost,my_command_value2);
      if(err == FZRT_NOERR)
      {  err = fz_iost_one_long(iost,my_command_value3);
         if(err == FZRT_NOERR)
         {  err = fz_iost_one_long(iost,my_command_value4);

            if(version >= 1)
            {  err = fz_iost_one_long(iost,my_command_value5);
            }
         }
      }
   }

   return(err);
}
```

The project window data IO function (optional)

```
long fz_cmnd_cbak_proj_wind_data_io (
      long                       windex,
      fz_iost_ptr                iost,
      fz_iost_dir_td_enum                dir,
      mod long                   version,
      long                       size
      );
```

form•Z calls this function to read and write any command specific project window data to a **form•Z** project file. This function is called once for each window in the project when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** Project file and should be used in all IO Stream function calls. The IO Stream functions are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the plugin to maintain version changes of the script data. In the following example, in its first release, a commands data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the command preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the script that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_cmnd_cbak_proj_wind_data_io (
    long                    windex,
    fz_iost_ptr             iost,
    fz_iost_dir_td_enum     dir,
    mod long                version,
    long                    size
    )
{
    long      err = FZRT_NOERR;

    if ( dir == FZ_IOST_WRITE ) version = 1;

    err = fz_iost_one_long(iost,my_command_value1);
    if(err == FZRT_NOERR)
    {   err = fz_iost_one_long(iost,my_command_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_command_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_command_value4);

                if(version >= 1)
                {   err = fz_iost_one_long(iost,my_command_value5);
                }
            }
        }
    }

    return(err);
}
```

3.7.2 Palette Scripts

A palette is a floating window that contains an interface for a feature or set of related features. The interface is composed of a variety of interface elements (buttons, radio buttons, check boxes, etc.) provided by the **form•Z** interface manager (fuim). Palette scripts are extensions that complement the **form•Z** palettes and behave consistently with the **form•Z** palettes.

Palettes are available in ***system*** and ***project*** levels. System palettes are global in nature and do not require a project window index while project palettes require a project or window index and are expected to operate on project information for a provided project, Palettes are flexible extensions as a lot of functionality can be included in a palette. The interface of the palette is defined by the extension through a fuim template. A description of fuim templates can be found in section 3.5 and in the **form•Z** API reference.

The names of palette scripts are added to a group near the bottom of the Palettes menu. As with all other palette names in this menu, selecting a palette name toggles the visibility of the palette. That is, if the palette is visible, then it is hidden and vice versa. Palettes that are visible are indicated by a check mark in the menu before the name. All palettes appear in the Key Shortcuts Manager dialog so that they may have key shortcuts assigned to them to open and close the palette. Note that if it is desirable to have the ability for the user to assign a key shortcut for individual items within the interface of the palette, then a separate command script must be implemented for this action.

The Samples directory in the Scripts folder contains a folder named Palettes that contains an example of a palette script named palt_my_view.fsl. This example creates a project palette with buttons for selecting a standard view type. This sample can be very valuable as both starting points for development as well as examples of how the functions work.

Palette script type

Palette scripts are defined by tagging the script in its header with the `script_type` keyword and the proper identifier as follows:

```
script_type FZ_PALT_SYST_EXTS_TYPE
```

for a system level palette script and

```
script_type FZ_PALT_PROJ_EXTS_TYPE
```

for a project level palette script.

3.7.2.1 System Palette

System palette scripts are implemented by defining a set of callback function. Only one is required, while others are optional, but should be implemented to enable certain functionality. All callback functions, if implemented, must match exactly the required name, return type and arguments as described below. As with all other script types, the system palette script may implement the `fz_script_cbak_info` callback function, which defines basic information about the script. This is discussed in more detail in section 3.3.

The initialization function (optional)

```
long fz_palt_cbak_syst_init();
```

This function is called by **form•Z** once when the script is successfully loaded and registered. The initialization function is where the script should initialize any data that may be needed by the other functions in the script.

```
long   fz_palt_cbak_syst_init()
{
       long   err = FZRT_NOERR;

       /** Do initialization here **/

       return(err);
}
```

The finalization function (optional)

```
long fz_palt_cbak_syst_finit();
```

This function is called by **form•Z** once when the script is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to perform any cleanup that may be necessary.

```
long fz_palt_cbak_syst_finit()
{
       long          err = FZRT_NOERR;

       /** clean up here **/

       return(err);
}
```

The name function (recommended)

```
long fz_palt_cbak_syst_name (
       mod fz_string_td    name,
       long                max_len
       );
```

This function is called by **form•Z** at various times to get the name of the palette. It is recommended that the name is stored in a .fzr file so that it is localizable. The name is the name that is added to the palette menu and is used as the tittle for the palette.

```
long fz_palt_cbak_syst_name (
       mod fz_string_td    name,
       long                max_len
       )
{
       long          err = FZRT_NOERR;

       /* Get the title string “My Palette” from the script's resource file */
       err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, name);

       return(err);
}
```

The help function (recommended)

```
long fz_palt_cbak_syst_help (
```

```
	mod fz_string_td	help,
	long			max_len
	);
```

This function is called by **form•Z** to display a help string that describes the detail of what the palette does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a string which can handle up to `max_len` characters. It is recommended that the help string is stored in .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 256 characters.

```
long fz_palt_cbak_syst_help (
	mod fz_string_td	help,
	long			max_len
	)

{
	long		err = FZRT_NOERR;

	/* Get the help string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, help);

	return(err);
}
```

The interface template function (required)

```
long fz_palt_cbak_syst_iface_tmpl (
	fz_fuim_tmpl_ptr		tmpl_ptr
	);
```

This function is called by **form•Z** when the interface for the palette is needed. The **form•Z** interface template functions should be called to construct the interface of the palette in this function. Please see section 3.5 for more details on the fuim template functions that are available for scripts. As scripts are more limited in scope than plugins, the range of fuim functions is smaller and only certain dialog interface items can be constructed by a palette script.

The following sample is a template for 3 buttons grouped inside a border with a title.

```
#define MY_STRINGS		1

#define MY_STRING_NAME	1
#define MY_STRING_TYPE	2
#define MY_STRING_1		3
#define MY_STRING_2		4
#define MY_STRING_3		5

long fz_palt_cbak_syst_iface_tmpl (
	fz_fuim_tmpl_ptr		tmpl_ptr
	)
{
	long			err;
	long			gindx;
	fz_string_td		str;

	/* get the options title from script's resource file */
	fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS, MY_STRING_NAME, str);
	if((err = fz_fuim_script_tmpl_init(tmpl_ptr, str,0,
			MY_PALETTE_TMPL_UUID, 0)) == FZRT_NOERR)
	{
		/* create a static text item */
		fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS,
```

```
                MY_STRING_TYPE, str);

            gindx = fz_fuim_script_new_text_static(tmpl_ptr, FZ_FUIM_ROOT,
                        FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_EQSZ, str);

                /* create a button */
                fzrt_fzr_get_string(my_rfzr_refid,
                        MY_STRINGS, MY_STRING_1, str);
                fz_fuim_script_new_button(tmpl_ptr, gindx,
                        FZ_FUIM_FLAG_NONE, str, "my_button_func1");

                /* create a button */
                fzrt_fzr_get_string(my_rfzr_refid,
                        MY_STRINGS, MY_STRING_2, str);
                fz_fuim_script_new_button(tmpl_ptr, gindx,
                        FZ_FUIM_FLAG_NONE, str, "my_button_func2");

                /* create a button */
                fzrt_fzr_get_string(my_rfzr_refid,
                        MY_STRINGS, MY_STRING_3, str);
                fz_fuim_script_new_button(tmpl_ptr, gindx,
                        FZ_FUIM_FLAG_NONE, str, "my_button_func3");
        }

        return (err);
}
```

Note, that the fuim function `fz_fuim_script_new_button` receives the name of a function, which is called by **form•Z**, when the button is pressed by the user. This function must be defined in the same script. It can have any name, but must have a return type of `long` and must have one argument, which is is a pointer of type `fz_fuim_tmpl_ptr`. The return value must be TRUE, if the function executed any statements, which represent the action assigned to the button. It should return FALSE, if the pressing of the button did not execute anything. One of the button functions used above is shown below:

```
long my_button_func1(
        fz_fuim_tmpl_ptr            tmpl_ptr
        )
{

        /* Add code here which executes when button was pressed */

        return(TRUE);
}
```

The preferences IO function (optional)

```
long fz_palt_cbak_syst_pref_io (
        fz_iost_ptr                 iost,
        fz_iost_dir_td_enum                 dir,
        mod long                    version,
        long                        size
        );
```

form•Z calls this function to read and write any palette specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions available for scripts are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the script data. In the following example, in its first release, a palette data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette preference data, **form•Z** will pass in the version number of the palette data when it was written, in this case 0. This indicates to the script, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_palt_cbak_syst_pref_io (
    fz_iost_ptr         iost,
    fz_iost_dir_td_enum dir,
    mod long            version,
    long                size
    )
{
    long      err = FZRT_NOERR;


    if ( dir == FZ_IOST_WRITE ) version = 1;

    err = fz_iost_one_long(iost,my_palette_value1);
    if(err == FZRT_NOERR)
    {   err = fz_iost_one_long(iost,my_palette_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_palette_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_palette_value4);

                if(version >= 1)
                {   err = fz_iost_one_long(iost,my_palette_value5);
                }
            }
        }
    }

    return(err);
}
```

3.7.2.2 Project Palette

Project palette scripts are implemented by defining a set of callback function. Only two are required, while others are optional, but should be implemented to enable certain functionality. All callback functions, if implemented, must match exactly the required name, return type and arguments as described below. As with all other script types, the project palette script may implement the `fz_script_cbak_info` callback function, which defines basic information about the script. This is discussed in more detail in section 3.3.

The initialization function (optional)

```
long fz_palt_cbak_proj_init ();
```

This function is called by **form•Z** once when the script is successfully loaded and registered. The initialization function is where the script should initialize any data that may be needed by the other functions in the function set.

```
long fz_palt_cbak_proj_init ()
{
	long		err = FZRT_NOERR;

	/* Do initialization here */

	return(err);
}
```

The finalization function (optional)

```
long fz_palt_cbak_proj_finit();
```

This function is called by **form•Z** once when the script is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to perform any cleanup.

```
long fz_palt_cbak_proj_finit()
{
	long		err = FZRT_NOERR;

	/* perform cleanup here  */

	return(err);
}
```

The information function (required)

```
long fz_palt_cbak_proj_info (
	mod fz_proj_level_enum	level
	);
```

This function is called by **form•Z** once when the script is successfully loaded and registered immediately after the initialization function (if provided). The `level` parameter indicates the context of the palette. `FZ_PROJ_LEVEL_MODEL` indicates that the palette operates on the project's modeling content (objects for example). `FZ_PROJ_LEVEL_MODEL_WIND` indicates that the palette operates on window specific content (views for example) of modeling windows. `FZ_PROJ_LEVEL_DRAFT` indicates that the palette operates on the projects drafting content (elements for example). `FZ_PROJ_LEVEL_DRAFT_WIND` indicates that the palette operates on window specific content (views for example) of drafting windows. **form•Z** uses the value in this parameter to determine when the palette should be shown and when it should be updated.

```
long fz_palt_cbak_proj_info (
	mod fz_proj_level_enum	level
	)
{
	long		err = FZRT_NOERR;
```

```
	level = FZ_PROJ_LEVEL_MODEL;

	return(err);
}
```

The name function (recommended)

```
long fz_palt_cbak_proj_name (
	mod fz_string_td	name,
	long			max_len
	);
```

This function is called by **form•Z** at various times to get the name of the palette. It is recomended that the name is stored in a .fzr file so that it is localizable. The name is the name that is added to the palette menu and is used as the tittle for the palette.

```
long fz_palt_cbak_proj_name (
	mod fz_string_td	name,
	long			max_len
	)
{
	long		err = FZRT_NOERR;

	/* Get the title string “My Palette” from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, name);

	return(err);
}
```

The help function (recommended)

```
long fz_palt_cbak_proj_help (
	mod fz_string_td	help,
	long			max_len
	);
```

This function is called by **form•Z** to display a help string that describes the detail of what the palette does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a string which can handle up to `max_len` characters. It is recommended that the help string is stored in a .fzr file so that it is localizable. The display area for help is limited so **form•Z** currently will ask for no more than 256 characters.

```
long fz_palt_cbak_proj_help (
	mod fz_string_td	help,
	long			max_len
	)
{
	long		err = FZRT_NOERR;

	/* Get the help string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, help);

	return(err);
}
```

The interface template function (required)

```
long fz_palt_cbak_proj_iface_tmpl (
	long				windex,
	fz_fuim_tmpl_ptr		tmpl_ptr
```

```
	);
```

This function is called by **form•Z** when the interface for the palette is needed. The **form•Z** interface template functions should be called to construct the interface of the palette in this function. Please see section 3.5 for more details on the fuim template functions. The full fuim template documentation can be found in the API reference.

The following sample is a template for 3 buttons grouped inside a boarder with a title.

```
#define MY_STRINGS		1

#define MY_STRING_NAME		1
#define MY_STRING_TYPE		2
#define MY_STRING_1			3
#define MY_STRING_2			4
#define MY_STRING_3			5

long fz_palt_cbak_proj_iface_tmpl (
	long				windex,
	fz_fuim_tmpl_ptr		tmpl_ptr
	)
{
	long			err;
	long			gindx;
	fz_string_td			str;

	/* get the options title from script's resource file */
	fzrt_fzr_get_string(my_rfzr_refid, MY_STRINGS, MY_STRING_NAME, str);
	if((err = fz_fuim_script_tmpl_init(tmpl_ptr, str,0,
			MY_PALETTE_TMPL_UUID, 0)) == FZRT_NOERR)
	{
		/* create a static text item */
		fzrt_fzr_get_string(my_rfzr_refid,
			MY_STRINGS,MY_STRING_TYPE, str);

		gindx = fz_fuim_script_new_text_static(tmpl_ptr, FZ_FUIM_ROOT,
				FZ_FUIM_FLAG_BRDR | FZ_FUIM_FLAG_EQSZ, str);

			/* create a button */
			fzrt_fzr_get_string(my_rfzr_refid,
				MY_STRINGS, MY_STRING_1, str);
			fz_fuim_script_new_button(tmpl_ptr, gindx,
				FZ_FUIM_FLAG_NONE, str, "my_button_func1");

			/* create a button */
			fzrt_fzr_get_string(my_rfzr_refid,
				MY_STRINGS, MY_STRING_2, str);
			fz_fuim_script_new_button(tmpl_ptr, gindx,
				FZ_FUIM_FLAG_NONE, str, "my_button_func2");

			/* create a button */
			fzrt_fzr_get_string(my_rfzr_refid,
				MY_STRINGS, MY_STRING_3, str);
			fz_fuim_script_new_button(tmpl_ptr, gindx,
				FZ_FUIM_FLAG_NONE, str, "my_button_func3");
	}

	return (err);
}
```

The preferences IO function (optional)

```
long fz_palt_cbak_plat_proj_pref_io (
	fz_iost_ptr			iost,
	fz_iost_dir_td_enum		dir,
	mod long			version,
	long				size
	);
```

form•Z calls this function to read and write any palette specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions available for scripts are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the script data. In the following example, in its first release, a palette's data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette data, **form•Z** will pass in the version number of the data when it was written, in this case 0. This indicates to the script, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value..

```
long fz_palt_cbak_plat_proj_pref_io (
    fz_iost_ptr         iost,
    fz_iost_dir_td_enum dir,
    mod long            version,
    long                size
    )
{
    long      err = FZRT_NOERR;


    if ( dir == FZ_IOST_WRITE ) version = 1;

    err = fz_iost_one_long(iost,my_palette_value1);
    if(err == FZRT_NOERR)
    {   err = fz_iost_one_long(iost,my_palette_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_palette_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_palette_value4);

                if(version >= 1)
                {   err = fz_iost_one_long(iost,my_palette_value5);
                }
            }
        }
    }
```

```
    return(err);
}
```

The project data IO function (optional)

```
long fz_palt_cbak_proj_data_io (
        long                        windex,
        fz_iost_ptr                 iost,
        fz_iost_dir_td_enum                 dir,
        mod long                    version,
        long                        size
        );
```

form•Z calls this function to read and write any palette specific project data to a **form•Z** project file. This function is called once when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** project file and should be used in all IO Stream function calls. The IO Stream functions available to scripts are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the script data. In the following example, in its first release, a palette's project data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette's project data, **form•Z** will pass in the version number of the data when it was written, in this case 0. This indicates to the script, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_palt_cbak_proj_data_io (
    long                windex,
    fz_iost_ptr         iost,
    fz_iost_dir_td_enum dir,
    mod long            version,
    long                size
    )
{
    long      err = FZRT_NOERR;


    if ( dir == FZ_IOST_WRITE ) version = 1;

    err = fz_iost_one_long(iost,my_palette_value1);
    if(err == FZRT_NOERR)
    {   err = fz_iost_one_long(iost,my_palette_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_palette_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_palette_value4);
```

```
            if(version >= 1)
            {   err = fz_iost_one_long(iost,my_palette_value5);
            }
        }
    }
  }

  return(err);
}
```

The project window data IO function (optional)

```
long fz_palt_cbak_proj_wind_data_io (
        long                            windex,
        fz_iost_ptr                     iost,
        fz_iost_dir_td_enum                     dir,
        mod long                        version,
        long                            size
        );
```

form•Z calls this function to read and write any palette specific project window data to a **form•Z** project file. This function is called once for each window in the project when reading and writing **form•Z** project files. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the **form•Z** Project file and should be used in all IO Stream function calls. The IO Stream functions available to scripts are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that was written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written in the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the script data. In the following example, in its first release, a palette's window data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the palette's window data, **form•Z** will pass in the version number of the data when it was written, in this case 0. This indicates to the script, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_palt_cbak_proj_wind_data_io (
    long                windex,
    fz_iost_ptr         iost,
    fz_iost_dir_td_enum dir,
    mod long            version,
    long                size
    )
{
    long      err = FZRT_NOERR;


    if ( dir == FZ_IOST_WRITE ) version = 1;

    err = fz_iost_one_long(iost,my_palette_value1);
    if(err == FZRT_NOERR)
```

```
    {   err = fz_iost_one_long(iost,my_palette_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_palette_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_palette_value4);

                if(version >= 1)
                {   err = fz_iost_one_long(iost,my_palette_value5);
                }
            }
        }
    }

    return(err);
}
```

3.7.3 RenderZone Shaders

The shader pipeline

When a pixel in an image is rendered, the shaders needed to compute the final pixel color are executed in a specific order. This order is referred to as the shader pipeline. The sequence of the shader pipeline for each pixel is as follows:
1. The color shader of the material assigned to the surface on which the pixel lies is executed. This defines the unshaded pixel color.
2. The bump shader of the material assigned to the surface on which the pixel lies is executed. This defines a new normal direction at the pixel, which is important for the reflection calculation that comes next.
3. The reflection shader of the material assigned to the surface on which the pixel lies is executed. The unshaded pixel color, generated by the color shader is augmented with shading information from all lights in the scene. If a bump shader other than None was used, the altered surface normal direction will be used to create bump patterns from the shading calculation. The shaded color is returned by the reflection shader.
4. The transparency shader of the material assigned to the surface on which the pixel lies is executed. The transparency of the pixel is returned by the shader and retained by **form•Z**.
5. If the transparency value from step 4 is more than 0.0 (i.e. there is some level of transparency) the background shader is executed. The color from the background shader and the shaded color from step 3 are mixed using the transparency value and returned by the shader.
6. The depth effect shader is executed. It uses the color from step 5. A new color is calculated using the depth information of the current pixel. This color is returned and becomes the final pixel color in the image.

Any of the six shaders contained in the shader pipeline can be extended through a script. Color, reflection, transparency and bump extension shaders are added to the respective menus in the Surface Style Parameters dialog. Background and Depth Effect script shaders are added in the RenderZone Options dialog. A Background script shader also becomes available as an Environment shader.

Shader script type

Each of the six shader types is identified by a different keyword in the header portion of the script. To identify a color shader the first line in the scrip should be:

```
script_type FZ_SHDR_COLR_EXTS_TYPE
```

Similarily, the other shader types are identified as follows:

```
script_type FZ_SHDR_REFL_EXTS_TYPE
script_type FZ_SHDR_TRNS_EXTS_TYPE
script_type FZ_SHDR_BUMP_EXTS_TYPE
script_type FZ_SHDR_BGND_EXTS_TYPE
script_type FZ_SHDR_FGND_EXTS_TYPE
```

Shader call back functions

Shader scripts are implemented by defining a number of call back functions. Of the eight callback functions of a color shader, only some are required, while others are optional. When an optional callback is defined, the respective functionality of the shader is disabled. For example, if the `fz_shdr_cbak_colr_avg` callback function is not provided, **form•Z** will substitute a 50% gray for the color, whenever a single solid color is used, such as in wireframe drawing. The required callback functions for a color shader are:

```
fz_shdr_cbak_colr_name
fz_shdr_cbak_colr_pixel
```

Optional functions are:

```
fz_script_cbak_info
fz_shdr_cbak_colr_set_parameters
fz_shdr_cbak_colr_pre_render
fz_shdr_cbak_colr_post_render
fz_shdr_cbak_colr_get_avg
```

The functions shown below are taken from the Sine Wave shader scripts, which are available as samples in the **form•Z** SDK.

The following section gives a detailed description of each of the shader functions and what task each function is expected to perform. The functions are explained in detail for the color shader. Any differences for the equivalent function of the other shaders are noted where necessary.

The script init function (recommended)

```
long fz_script_cbak_info(mod fzrt_UUID_td uuid,
                         mod fz_string_td title,
                         mod fz_string_td vendor,
                         mod long         version);
```

This function defines a unique identifier and returns basic information about the script. It is described in more detail in section 3.3. If this function is implemented, it needs to return the version of the shader. It is up to the developer to assign a version number to the shader. When a **form•Z** project file is saved with a script shader, the version of the shader is saved as well. If the project is opened later and a newer version of the shader exists at that time, **form•Z** will reset the parameters of the shader to default values. A shader developer must increase the version number when, during ongoing development of the shader, the parameters of the older shader do not match the parameters of the newer shader. If the shader is changed so that saved shader parameters are still meaningful, and are aligned with the current shader parameters, then the version does not need to be changed. Assume, for example, that a shader is originally defined with 2 color and 2 integer parameters. The version assigned to the shader initially was 0. In the second release of the shader, the developer adds a 5th parameter. This requires that the version be increased to 1. In a third release of the shader, the first integer parameter, which originally could take on values between 0 and 10, can now take on values from 0 to 20. This does not require a version change.

The name function (required)

```
long  fz_shdr_cbak_colr_name(
      mod fz_string_td name,
      long maxlen
      );
```

The name function must assign a string to the name argument. The length of the string assigned cannot exceed max_len characters. This string appears as the shader's name in the respective menu. It is recommended that the name is stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example below, this step is omitted for the purpose of simplicity. A script name function would look like this:

```
long  fz_shdr_cbak_colr_name(mod fz_string_td name, long maxlen)
{
      name = "Sine Wave";
      return(FZRT_NOERR);
}
```

The set parameters function (optional)

```
long  fz_shdr_cbak_colr_set_parameters();
```

The set parameters function is called once at startup. It needs to establish the number and types of parameters for the shader. Based on the parameters set up in this function, **form•Z** automatically builds the content of the shader's option dialog, which can be invoked by clicking on the Options... botton next to the shader menu, as usual. Setting the shader's parameters is accomplished with a number of **form•Z** API function calls. There are standard parameters which can be set up automatically, such as scale or noise. Custom parameters can be created individually, such as colors, floating point values with sliders or check boxes. If the shader is a color, transparency or bump shader, fhe first **form•Z** API call in the set parameters function should identify the shader as a 2d (wrapped) or 3d (solid) shader. This is done with the API call:

```
      fz_shdr_set_wrapped(TRUE);
```

if the shader is 2d, and

```
      fz_shdr_set_solid(TRUE);
```

if the shader is 3d. Note, not calling these functions is equivalent to calling either function with the argument set to `FALSE`. It is also possible to call both function with `TRUE`, in which case the shader would be labeled as a 2d and 3d shader. While this is rarely the case, it is conceivable, that a shader creates a pattern based on 2d and 3d texture space mapping. Mirror, background and depth effect shaders do not need to call this API function.

Shaders which create a pattern should present the standard scale parameter to a user. This parameter is set up with the API call:

```
      fz_shdr_set_scale_parm (1.0);
```

The function argument 1.0 sets the default value of the scale parameter to 100%. This function call will automatically add the Scale field in the shader options dialog. **form•Z** will apply the current scale factor to the 2d or 3d texture space coordinate, which is used in the pixel function to calculate the shader's pattern.

If a shader uses any of the noise functions, which create random patterns, the standard noise parameters can be added to the shader with the API call:

```
      fz_shdr_set_noise_parm(FZ_SHDR_TURB_TYPE_BETTER,3);
```

This will add the Noise menu and # of Impulses field to the shader option dialog. The current setting of these parameters may be retrieved in the pre-render function and used in a call to any of the noise functions in the shader's pixel function.

Most procedural shaders that create some kind of pattern suffer from strong moire artifacts, when the pattern becomes very small. With an area sampling technique, these artifacts can be avoided.

Automatic area sampling can be added to a color, transparency, or bump shader by adding the standard shader parameter with the API function call:

```
fz_shdr_set_area_sample_parm(FALSE);
```

The argument in the API function call sets the default value of area sampling to TRUE or FALSE (FALSE should be the default). The standard "Area Sampling" check box will be added by **form•Z** in the shader dialog. If this API call is not made in the set parameters callback, the shader will not have area sampling. Note, that his call only applies in the set parameters function of color, transparency and bump shaders. For all other shader types, this API call is ignored.

If the shader is a reflection shader, additional standard parameters can be set up. They define the six shading parameters: ambient, diffuse, specular, mirror, transmission and glow:

```
fz_shdr_set_ambient_parm (1.0);
fz_shdr_set_diffuse_parm (0.75);
fz_shdr_set_specular_parm (0.5,0.1);
fz_shdr_set_specular_color_parm (col);
fz_shdr_set_mirror_parm (0.5);
fz_shdr_set_transmission_parm (0.5,1.0);
fz_shdr_set_glow_parm (0.0);
```

When the respective setup call is made, the shader options dialog will add the Factor field, Map menu and map Options button. Not all reflection parameters need to be offered. Any combination of them can be selected and mixed with custom parameters.

Custom parameters are created with the API calls:

```
fz_shdr_set_pct_parm("Value 1", 0.5, 1, 1, SHDR_VAL1_ID);
fz_shdr_set_col_parm("Color 1", col, SHDR_COL_ID);
fz_shdr_set_sld_flt_parm("Value 2", 0.5,1,1, SHDR_VAL2_ID);
fz_shdr_set_sld_int_parm("Value 3", 5,1,10,1,1, SHDR_VAL3_ID);
fz_shdr_set_flt_parm("Value 4", 0.5, 0.0, 1.0, 1, 1, SHDR_VAL4_ID);
fz_shdr_set_int_parm("Value 5", 5, 1, 10, 1, 1, SHDR_VAL5_ID);
fz_shdr_set_bool_parm("Boolean", TRUE, SHDR_BOOL_ID);
```

Each of these calls creates a shader parameter of the respective type, with the given title, default values, allowable range and range checking. The last parameter to each function is an integer id, which must be unique. This id is used when retrieving the current value of a parameter in the pre render function. It is possible to pass a value of -1 for the id argument. In this case **form•Z** will generate a unique id and pass it back through the function's return value. For example:

```
id = fz_shdr_set_col_parm("Color 1", col, -1);
```

Since the **form•Z** generated id must be used to retrieve the parameter value in the pre render function, it must be a global variable.

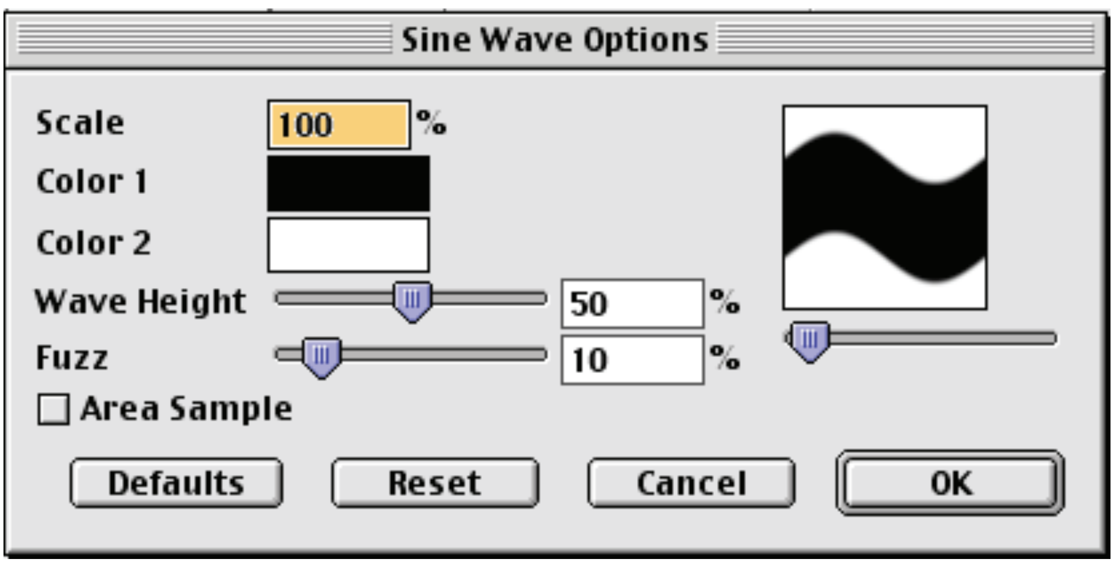
A user may edit the preset and custom values in the options dialog. In the pre render function the current values of the custom parameters should be retrieved and passed on to the pixel function, where they are used to compute the shader pattern.

The set parameters function for the Sine Wave color shader in a script is:

```
#define     PARAM_ID_1        1
#define     PARAM_ID_2        2
#define     PARAM_ID_HEIGHT   3
#define     PARAM_ID_FUZZ     4

long  fz_shdr_cbak_colr_set_parameters()
{
      fz_rgb_float_td   col;

      // 2d texture mapping
      fz_shdr_set_wrapped(TRUE);
```

```
        // allow scaling
        fz_shdr_set_scale_parm(1.0);
        // allow area sampling
        fz_shdr_set_area_sample_parm(TRUE);

        // color of first part of sine wave
        col = {0.0,0.0,0.0};
        fz_shdr_set_col_parm("Color 1", col, PARAM_ID_1);
        // color of second part of sine wave
        col = {1.0,1.0,1.0};
        fz_shdr_set_col_parm("Color 2", col, PARAM_ID_2);
        // amplitude of sine wave
        fz_shdr_set_sld_flt_parm("Wave Height", 0.5, 1, 2, PARAM_ID_HEIGHT);
        // fuzziness of boundary
        fz_shdr_set_sld_flt_parm("Fuzz", 0.1, 1, 1, PARAM_ID_FUZZ);
}
```

The dialog resulting from these shader parameters is shown below:



There is one important detail to the use of the custom parameters API functions, such as `fz_shdr_set_sld_flt_parm`. The first parameter to this API is the name of the parameter as it will appear in the shader dialog. A transparency shader is also used to define an equivalent shader in the reflection map menus of a reflection shader, which uses any of the six standard reflection parameters. When the dialog for this shader (if used as a diffuse map for eample) is invoked, the parameter, which would be called, for example, "Background Transparency" in the transparency shader options dialog, is called "Background Diffuse" in the diffuse map options dialog. This automatic adjustment of the parameter name can be achieved by substituting %s in the name parameter of the API, for those calls of the API which would use the word "Transparency" in the dialog. The same mechanism also works for color and bump shaders, although they are not used in any other context. For example the API call to define a color in the set parameters function of a color shader can be written in two different ways:

```
        fz_shdr_set_col_parm("Color 1",def_col1, PARAM_ID_COLOR1);
```

or

```
        fz_shdr_set_col_parm("%s 1",def_col1, PARAM_ID_COLOR1);
```

While it is not necessary to substitute the %s in color and bump shaders, it is necessary to do so in transparency shaders, in order to get the correct parameter title, when the transparency shader is also used in the context of a reflection map shader.

The pre render function (recommended)

```
long  fz_shdr_cbak_colr_pre_render();
```

This function is called once before the start of each rendering. It is expected to precompute information that will be used by the pixel function. Using the pre render function can significantly speed up the execution of a shader. Certain information, that is needed during the calculation of the shader pattern does not change during the rendering. For example, a shader may use a floating point value from a shader parameter, but really needs the inverse (1.0 / value) during the pixel calculation. Instead of computing 1.0 / value each time during the execution of the pixel function, the value can be computed once in the pre render function and then be reused in the pixel function. Any of the shader parameters defined in the set parameters function can be retrieved in the pre render function. For the standard parameter function, there are the equivalent functions which get the current value of a standard parameter. They are:

```
fz_shdr_get_noise_type
fz_shdr_get_noise_impulses
```

Note, that there is no function to get the scale parameter. **Form•Z** automatically applies the scale factor, if it exists, to the texture space or 3D coordinate before it is used by the pixel function. For custom parameters, a single API call retrieves the value of a given parameter:

```
fz_shdr_get_parm(PARAM_ID,value);
```

The parameter is identified by the first argument to the function, which is the id used when the parameter was defined, or the id generated by **form•Z**, if -1 was passed for the id. The standard reflection parameters for ambient, diffuse, specular, mirror, transmission and glow should not be retrieved in the pre render function but in the pixel function. This is described in more detail later in this section.

The pre render function typically will use global variables and fill them with precomputed information. The pixel function may then use the global variables for its own computations. The pre_render function for the Sine Wave color shader is shown below.

```
fz_rgb_float_td   col1;
fz_rgb_float_td   col2;
double            min_left, min_right, max_left, max_right, amplitude;

long  fz_shdr_cbak_colr_pre_render()
{
      double fuzz;

      fz_shdr_get_parm(PARAM_ID_1, col1);
      fz_shdr_get_parm(PARAM_ID_2, col2);
      fz_shdr_get_parm(PARAM_ID_HEIGHT, amplitude);
      fz_shdr_get_parm(PARAM_ID_FUZZ, fuzz);

      // keep sine wave within boundaries
      amplitude = amplitude * 0.25;
      if (amplitude < 0.0)
```

```
		amplitude = 0.0;

	// scale fuzziness
	fuzz = fuzz * 0.25;
	if (fuzz < 0.0 ) fuzz = 0.0;
	if (fuzz > 0.25) fuzz = 0.25;

	// set where to start and end the sine wave
	// boundaries based on the fuzziness.
	min_left  = 0.25 - fuzz;
	min_right = 0.25 + fuzz;
	max_left  = 0.75 - fuzz;
	max_right = 0.75 + fuzz;

	return(FZRT_NOERR);
}
```

The pixel function (required)

The pixel function is called during a rendering one or more times for each pixel. Depending on which kind of shader is written, the pixel function needs to compute different types of information.

The color pixel function

```
fz_rgb_float_td fz_shdr_cbak_colr_pixel();
```

For a color shader, the pixel function needs to compute and return the rgb color of the surface pixel, based on the 2d or 3d texture coordinate. This coordinate is retrieved via a **form•Z** API call:

```
	fz_shdr_get_tspace_st(st);
```

for 2d shaders or

```
	fz_shdr_get_tspace_pnt(pnt);
```

for 3d shaders. Note, that the scale factor, set up in the set parameters function does not need to be applied to the 2d or 3d texture space coordinate, as **form•Z** already has performed this step. Together with the shader parameters, the point's coordinates can be transformed into a color pattern. A number of **form•Z** API function are offered to facilitate the computation of regular and random patterns. This is described in further detail in later in this section. The pixel function of the Sine Wave color shader is shown below:

```
fz_rgb_float_td fz_shdr_cbak_colr_pixel()
{
	double			ss,tt;
	fz_rgb_float_td	col;
	fz_xy_td		st;

	// get current texture space coordinate
	fz_shdr_get_tspace_st(st);

	// apply saw tooth filter w/sine function
	ss = fz_shdr_saw_tooth(st.x, 1.0);
	tt = fz_shdr_saw_tooth(st.y, 1.0) + sin(ss * FZ_2PI) * amplitude;
	tt = fz_shdr_saw_tooth(tt, 1.0);

	// apply fuzziness
	tt =		fz_shdr_smooth_step(min_left, min_right, tt) *
```

```
            (1.0 - fz_shdr_smooth_step(max_left, max_right, tt));

        // apply linear interpolation
        col = col1 * tt + (1.0 - tt) * col2;

        return(col);
}
```

Note, in the context of shader scripts one can run in to potential multi-threaded or multi-processor system-dependent problems. Global variables should not be used inside the pixel shading function if one is changing the value in the global variable in this call back function. This will result in noisy images where different threads or processors have computed different parts of the image with the same global variable that had been changing under their feet.

The reflection pixel function

```
fz_rgb_float_td fz_shdr_cbak_refl_pixel();
```

For a reflection shader, the pixel function is expected to take the pixel color, computed by the color shader and apply shading to it, based on the lighting conditions in the scene. The unshaded pixel color can be retrieved with the API call:

```
        fz_shdr_get_col(color);
```

If the reflection shader uses any of the standard reflection parameter setup function in the set parameters function, the current value of each parameter needs to the retrieved in the pixel function. Since any of the standard reflection parameters may be altered by a reflection map, the value of a reflection parameter may vary on a surface. Therefore, it cannot be retrieved in the pre render function and stored in a global variable. For example, consider the set parameters function of a reflection shader to define the standard diffuse reflection shader:

```
long    fz_shdr_cbak_refl_set_parameters()
{
        ...
        fz_shdr_set_diffuse_parm(0.75);
        ...
}
```

The pixel function of the same reflection shader would retrieve the current value of the diffuse parameter:

```
fz_rgb_float_td fz_shdr_cbak_refl_pixel()
{
        ...
        fz_shdr_get_diffuse_param(df);
        ...
}
```

`df` will then contain the diffuse factor at the current pixel, taking into account the value of the diffuse factor entered by the user and a possible diffuse map, which will alter the user's value based on the diffuse map's pattern. In addition to obtaining the diffuse factor for a pixel, it is also necessary to perform the actual diffuse illumination. **form•Z** offers API function which perform this task, as well as illumination for ambient, specular, mirror and transmission. Of course, it is up to the script writer to implement a custom illumination algorithm, if desired. The illumination function offered by **form•Z** are the same used by the RenderZone display mode. To calculate the diffuse illumination of a pixel the **form•Z** API

```
        fz_shdr_get_diffuse_term(dcol);
```

can be called. The color returned is the illumination from all lights, including shadows. Typically, this color is multiplied (filtered) with the unshaded pixel color, created by the color shader of a surface style to create the final diffuse shaded pixel. The classic shading algorithm computes the final pixel shading using ambient, diffuse and specular illumination with the following algorithm:

```
col_out = col_in * (af * acol + df * dcol) + sf * scol;
```

Where col_in is the unilluminated pixel color, `af` is the ambient factor, `acol` is the ambient color (the result of `fz_shdr_get_ambient_term`), `df` is the diffuse factor, `dcol` is the diffuse color (the result of `fz_shdr_get_diffuse_term`), `sf` is the specular factor and `scol` is the specular color (the result of `fz_shdr_get_specular_term`). The full pixel function for such a standard reflection shader would look like this:

```
fz_rgb_float_td fz_shdr_cbak_refl_pixel()
{
	double			af,df,sf;
	fz_rgb_float_td	col,acol,dcol,scol;

	fz_shdr_get_ambient_factor(af);
	fz_shdr_get_diffuse_factor(df);
	fz_shdr_get_specular_factor(sf);

	fz_shdr_get_ambient_term(acol);
	fz_shdr_get_diffuse_term(dcol);
	fz_shdr_get_specular_term(inv_roughness,scol);

	fz_shdr_get_col(col);
	col.r = col.r * (af*acol.r + df*dcol.r) + sf*scol.r;
	col.g = col.g * (af*acol.g + df*dcol.g) + sf*scol.g;
	col.b = col.b * (af*acol.b + df*dcol.b) + sf*scol.b;

	return(col);
}
```

Note, that the original color is filtered (multiplied) by the ambient and diffuse shading component and the specular color is added on top of it.

Adding raytraced effects.

In addition to the simple shading calculations shown above, it is possible to add reflection effects through raytracing. In the standard reflection shaders offered by **form•Z**, these effects create mirrored and transmission reflections. To add mirrored reflections, a **form•Z** API function can be called:

```
fz_shdr_raytrace_reflected(world_pt,mirr_vec,mf,mirr_col);
```

This function takes the following arguments: `world_pt` is the point where the reflected ray starts on the rendered surface. This point can be retrieved with the API call:

```
fz_shdr_get_world_pnt(world_pt);
```

which is the point on the surface where the current pixel is rendered. `mirr_vec` is the direction of the reflected ray as it bounces off the surface. For a true mirror surface, this direction is the direction of the view vector, reflected about the normal direction of the surface. The following API functions can be used to calculate this mirror direction:

```
fz_shdr_get_world_shading_normal(norm);
fz_shdr_get_view_dir(view_vec);
fz_shdr_ray_reflect(view_vec,norm,mirr_vec);
```

The mirror factor argument `mf` tells the `fz_shdr_raytrace_reflected` API function how much of the calculated mirror color will be added to the final shaded color. If the mirror factor is small, the raytracing can stop earlier, because the added mirror color only makes up a small component of the final pixel color, and it would not make any visible difference to let the raytraced ray bounce longer between other mirroring surfaces. However, if the mirror factor is large, such as in a perfect mirror, the reflected ray needs to bounce longer between other mirroring surfaces to compute accurate reflections. Recall that the termination of raytraced rays is determined through the options set in the Raytrace Options dialog, which is invoked from the RenderZone Options dialog.

To create transmission effects, which simulate glasslike materials, a similar API function can be called:

```
fz_shdr_raytrace_refracted(world_pt,mirr_vec,tf,mirr_col);
```

The arguments are the same as to `fz_shdr_raytrace_reflected`. The transmission factor argument `tf` acts in the same manner as the mirror factor argument. It determines how long refracted rays are allowed to bounce between transmissive and reflective surfaces. In order to calculate the vector with which a refracted ray enters a glass like material, the API function `fz_shdr_ray_refract` can be called. It bends an incoming ray, usually the view direction vector, about the surface normal, using the index of refraction of a material. Thus a complete calculation of a transmission effect can be written like this:

```
if ( tf > 0.0 )
{       fz_shdr_get_world_pnt(world_pt);
        fz_shdr_get_world_shading_normal(norm);
        fz_shdr_get_view_dir(view_vec);

        if(fz_shdr_ray_refract(view_vec,norm,eta,mirr_vec) == TRUE)
        {       fz_shdr_raytrace_refracted(world_pt,mirr_vec,tf,mirr_col);
                col.r += mcol.r * mf;
                col.g += mcol.g * mf;
                col.b += mcol.b * mf;

                if ( fz_shdr_ray_inside_solid() == TRUE ) mf = 0.0;
        }
}
```

Note, that the API `fz_shdr_ray_refract` returns a boolean value, which is TRUE, if the incoming ray is bent so that it enters the surface. When the incoming ray is angled in such a way, that with the given index of refraction, it would bounce off the surface rather than enter it, the API function returns FALSE. In this case no transmission needs to be calculated. Raytracing usually causes a recursive call to the shading pipeline. For example, a ray which is spawned through the call `fz_shdr_ray_reflect` as shown above, may hit another surface. The color of that point on the surface needs to be calculated through the same shader calls as the original surface pixel on the screen. As a result, the same pixel function may be invoked again in a nested fashion. Consider two parallel opposing mirrors. A ray bouncing off one mirror in an exact perpendicular direction would bounce between the two mirrors infinitely. **form•Z** will pre-empt this process at a given time, when a satisfactory accuracy of the color to be calculated is achieved. It is quite possible that there may be as many as 10 or more rays before this occurs. In this case, the pixel function of the mirror reflection shader would be called in a stack of 10 nestings. The same may be the case with `fz_shdr_ray_refract`. A typical glass like material is both refractive and reflective. This means that both raytrace API functions are called. If the ray from a refraction calculation is currently bouncing inside a solid material, such as the wall of a glass bottle, it is only necessary to spawn off another refracted ray when the ray exits the material on the other side. Only when the ray enters the material is it necessary to compute refraction and reflection. In the code example above, the API `fz_shdr_ray_inside_solid()` is called to determine, whether the current ray is inside or outside a solid material. If it is inside, the mirror factor for the subsequent reflection calculation is set to 0.0, effectively disabling mirroring for this ray. Putting all shading components together, a complete reflection shader can be written as shown below. This is actually the code that is used to implement the Generic reflection shader offered by **form•Z**.

```
fz_rgb_float_td fz_shdr_cbak_refl_pixel()
{
	double		af,df,sf,mf,tf,gf;
	fz_rgb_float_td		col,acol,dcol,scol,mcol,gcol;
	fz_xyz_td		world_pt,norm,view_vec,mirr_vec;


	fz_shdr_get_col(col);
	gcol = col;  /* SAVE UNSHADED SURFACE COLOR FOR GLOW LATER */


	/* GET REFLECTION FACTORS */
	fz_shdr_get_ambient_factor(af);
	fz_shdr_get_diffuse_factor(df);
	fz_shdr_get_specular_factor(sf);
	fz_shdr_get_mirror_factor(mf);
	fz_shdr_get_transmission_factor(tf);
	fz_shdr_get_glow_factor(gf);

	/* CALCULATE BASIC SHADING */
	fz_shdr_get_ambient_term(acol);
	fz_shdr_get_diffuse_term(dcol);
	fz_shdr_get_specular_term(inv_roughness,scol);

	col.r = col.r * (af*acol.r + df*dcol.r) + sf*scol.r;
	col.g = col.g * (af*acol.g + df*dcol.g) + sf*scol.g;
	col.b = col.b * (af*acol.b + df*dcol.b) + sf*scol.b;

	/* CALCULATE RAYTRACE EFFECTS */
	if ( mf > 0.0 || tf > 0.0 )
	{	fz_shdr_get_world_pnt(world_pt);
		fz_shdr_get_world_shading_normal(norm);
		fz_shdr_get_view_dir(view_vec);

		/* CALCULATE REFRACTED RAYS */
		if(tf > 0.0 &&
		   fz_shdr_ray_refract(view_vec,norm,eta,mirr_vec) == TRUE)
		{
			fz_shdr_raytrace_refracted(world_pt,mirr_vec,tf,mcol);
			col.r += mcol.r * tf;
			col.g += mcol.g * tf;
			col.b += mcol.b * tf;

			if ( fz_shdr_ray_inside_solid() == TRUE ) mf = 0.0;
		}

		/* CALCULATE REFLECTED RAYS */
		if ( mf > 0.0 )
		{	fz_shdr_ray_reflect(view_vec,norm,mirr_vec);
			fz_shdr_raytrace_reflected(world_pt,mirr_vec,mf,mcol);
			col.r += mcol.r * mf;
			col.g += mcol.g * mf;
			col.b += mcol.b * mf;
		}
	}

	/* NOW ADD GLOW, IF ANY */
	if ( gf > 0.0 )
	{
		col.r += gcol.r * gf;
		col.g += gcol.g * gf;
		col.b += gcol.b * gf;
	}
```

```
	return(col);
}
```

The transparency pixel function

```
double		fz_shdr_cbak_trns_pixel();
```

The pixel function of a transparency shader is expected to return the level of transparency of the current pixel towards the background. If a value of 0.0 is returned, the pixel is considered completely opaque. If 1.0 is returned, the pixel is considered completely transparent. Values less than 0.0 and larger than 1.0 are not accepted and are clamped to the respective limit. As with a color shader, the transparency shader can compute the pixel transparency based on a pattern. All utility function that can be used by a color shader also apply to a transparency shader. In addition, a transparency shader may compute transparency based on surface geometry. The Neon shader offered by **form•Z** is such a shader. It uses the angle between the surface normal and the view direction to compute the transparency. As such, it is not tagged as a 2d or 3d shader and therefore shows up in the correct section in the Transparency menu in the Surface Style Parameters dialog. The sine wave transparency shader pixel function is shown below:

```
double fz_shdr_cbak_trns_pixel()
{
	fz_xy_td	st;
	double		ss,tt;
	double		trn;


	shdr_get_tspace_st(st);

	ss = fz_shdr_saw_tooth(st.x,1.0);
	tt = fz_shdr_saw_tooth(st.y,1.0) + sin(ss * FZ_2PI)*amplitude;
	tt = fz_shdr_saw_tooth(tt,1.0);

	tt = fz_shdr_smooth_step(min_left, min_right, tt) *
		(1.0 - fz_shdr_smooth_step(max_left, max_right, tt));

	trn = val1 * tt + (1.0 - tt) * val2;

	return(trn);
}
```

The bump pixel function

```
double		fz_shdr_cbak_bump_pixel();
```

The pixel function of a bump shader is expected to return the bump amplitude (height) of the current pixel. Values should be in the range of 0.0 to 1.0, but may be smaller and larger. The pixel function of a bump shader is actually called more than once per pixel. A number of calls to this function determine how the surface bends around the area of the pixel. This information is then used to alter the normal direction used for the shading calculation during the pixel function of the reflection shader or a surface style. Bump shaders are usually either 2d or 3d and should therefore be tagged as such in the set parameters function. Special care should be taken when writing a bump shader that is based on a pattern. The transition of high and low areas in the pattern should be gradual and smooth for best bump results. For example, the sine wave shader shown below creates a "fuzzy" zone between the wave and background part of the pattern. This is achieved via the fuzz parameter using the `fz_shdr_smooth_step` utility API, which is described in further detail later in this section. If the transition between the wave and the background area would be sharp, the bumps would not be as pronounced, even with a large amplitude parameter. The sine wave bump shader pixel function is shown below:

```
double fz_shdr_cbak_bump_pixel()
{
	fz_xy_td	st;
	double		ss,tt;
	double		ampl;


	shdr_get_tspace_st(st);

	ss = fz_shdr_saw_tooth(st.x,1.0);
	tt = fz_shdr_saw_tooth(st.y,1.0) + sin(ss * FZ_2PI)*amplitude;
	tt = fz_shdr_saw_tooth(tt,1.0);

	tt = fz_shdr_smooth_step(min_left, min_right, tt) *
		(1.0 - fz_shdr_smooth_step(max_left,max_right,tt));

	ampl = val1 * tt + (1.0 - tt) * val2;

	return(ampl);
}
```

The background pixel function

```
fz_rgb_float_td   fz_shdr_cbak_bgnd_pixel();
```

The pixel function of a background shader is expected to calculate the color of a pixel in the background of the scene. A background pixel is a part of the image, which is not covered by a surface, or which may be visible through a transparent surface. No tagging as 2d or 3d is necessary for this shader in the set parameters function. The coordinate of the current background pixel can be retrieved with the API call:

```
	fz_shdr_get_ispace_xy(bg_pixel);
```

The coordinate for the upper left corner of the pixel would be x = 0.0, y = 0.0, the lower right corner is x = 1.0, y = 1.0 regardless of the image pixel resolution. The sine wave background shader pixel function is shown below:

```
fz_rgb_float_td		fz_shdr_cbak_bgnd_pixel()
{
	fz_xy_td		st;
	double			ss,tt;
	fz_rgb_float_td		col;

	fz_shdr_get_ispace_xy(st);

	ss = fz_shdr_saw_tooth(st.x,1.0);
	tt = fz_shdr_saw_tooth(st.y,1.0) + sin(ss * FZ_2PI)*amplitude;
	tt = fz_shdr_saw_tooth(tt,1.0);

	tt = fz_shdr_smooth_step(min_left,min_right, tt) *
		(1.0 - fz_shdr_smooth_step(max_left,max_right, tt) );

	col.r = col1.r * tt + (1.0 - tt) * col2.r;
	col.g = col1.g * tt + (1.0 - tt) * col2.g;
	col.b = col1.b * tt + (1.0 - tt) * col2.b;

	return(col);
}
```

Note, that this function is the same as the pixel function of the sine wave color shader, with the exception of the API call to get the pixel coordinate. The color pixel function uses `fz_shdr_get_tspace_xy(st)` to get the texture space coordinate, whereas the background pixel function uses `fz_shdr_get_ispace_xy(st)` to get the image space coordinate. Similar to the color shader pixel function, the standard scale factor is already contained in the image space coordinate.

The depth effect (foreground) pixel function

```
fz_rgb_float_td   fz_shdr_cbak_fgnd_pixel();
```

The pixel function of a depth effect shader is expected to change the color of a pixel based on the depth of the surface pixel in the scene. The depth effect shader is the last shader invoked in the shader pipeline. The API function `fz_shdr_get_dist_eye_world_pnt` can be called to get the distance of the pixel's world coordinate point to the eye point. If the current pixel is a background pixel, the API function will return FALSE. In this case, there is no surface to be rendered at that pixel. An example of a simple depth effect shader, that adds a constant color to a pixel based on its distance between the eye point and the yon view clipping plane is shown below:

```
fz_rgb_float_td     fz_shdr_cbak_fgnd_pixel()
{
        fz_rgb_float_td     col;
        double              dist,ratio,inv_ratio;


        fz_shdr_get_col(col);
        if( fz_shdr_get_dist_eye_world_pnt(dist) == TRUE )
        {

                ratio = dist / yon;
                if ( ratio > 1.0 ) ratio = 1.0;

                inv_ratio = 1.0 - ratio;

                col.r = col.r * inv_ratio + col.r * ratio;
                col.g = col.g * inv_ratio + col.g * ratio;
                col.b = col.b * inv_ratio + col.b * ratio;
        }

        return(col);
}
```

The post render function (optional)

```
long fz_shdr_cbak_colr_post_render();
```

This function is called once at the end of each rendering. It is expected to perform any tasks necessary when the shader is done rendering the image.

```
long fz_shdr_cbak_colr_post_render()
{
        /* CLEANUP CODE, IF ANY, GOES HERE */
        ...

        return(FZRT_NOERR);
}
```

Shader utiltity functions

There are a number of additional API function, which are intended to facilitate the implementation of a shader script. The most important of these apis are described in more detail below.

Repeating patterns

If a pattern is regular and repeats in a tile like fashion, such as bricks or checkers, the values of the texture coordinate need to be modulated. This can be done with the API call:

```
s = fz_shdr_saw_tooth(st.x,1.0);
t = fz_shdr_saw_tooth(st.y,1.0);
```

This guarantees, that the incoming values st.x and st.y, for example, oscillate between 0.0 and 1.0. The pattern algorithm then only needs to consider values in that range. In the Sine Wave shader, for example, the y component of the 2d texture coordinate is modified with `fz_shdr_saw_tooth`. This will yield one sine curve for each texture tile, instead of just one sine curve in the whole texture space. The saw tooth function can also be described through this simple algorithem:

```
if ( val_in < 0.0 ) val_out = -fmod(val_in,module);
else                val_out =  fmod(val_in,module);
```

Random Patterns

form•Z offers a number of utility functions, which compute a random pattern based on a single value, a 2d coordinate or a 3d coordinate. They are

```
fz_shdr_turbulance_1d
fz_shdr_turbulance_2d
fz_shdr_turbulance_3d
fz_shdr_noise_1d
fz_shdr_noise_2d
fz_shdr_noise_3d
```

The turbulance and noise functions are very similar. The turbulance functions take an additional integer parameter, which creates more detail if passed in with a higher value. The input to the noise and turbulance functions is usually a value of the texture space coordinate of the pixel to be rendered. The function returns a pseudo random number between 0.0 and 1.0. This number can be used to design a pattern. For example, the code below creates a random pattern of black dots on a white background:

```
fz_shdr_get_tspace_st(st);

val = fz_shdr_noise_2d(st,FZ_SHDR_TURB_TYPE_BETTER,0);

if ( val < 0.5 )  col = black_color;
else              col = white_color;
```

It is up to the creativity of the shader developer to use noise and turbulance functions to break up regular patterns and to create unique pattern designs. In **form•Z** these functions are used in a number of shaders. For example, the Textured Brick shader uses noise functions to mix two brick colors and also to break up the straight line of the mortar edges. The Textured Marble color shader uses turbulance functions to mix the marble colors.

Smooth transitions

It is often desirable to create a soft transition between two colors in a pattern. In **form•Z** shaders, this softening of contrast is called fuzz and offered in many shaders. Not only can it be used to create different variations of the shader pattern, but it also help to avoid aliasing artifacts. A API utility function is available to compute smooth transitions:

```
val_out = fz_shdr_smooth_step(min,max,val_in);
```

If the val parameter is less than min `fz_shdr_smooth_step` will return 0.0. If the val parameter is greater than max `fz_shdr_smooth_step` will return 1.0. If the val parameter is between min and max, `fz_shdr_smooth_step` will return a value between 0.0 and 1.0. However, the value is not a linear interpolation, When plotted as a function graph, the curve resembles a leaning S, connecting y = 0.0 and y = 1.0 in a smooth fashion. This function can be used to create fuzz along edges of sharp contrast in a pattern.

For example consider a simple pattern of horizontal stripes:

```
fz_shdr_get_tspace_st(st);
st.y = fz_shdr_saw_tooth(st.y,1.0);
if ( st.y < 0.5 )       col = black;
else                    col = white;
```

This will create a crisp border between the black and white color. To create a fuzzy border, `fz_shdr_smooth_step` can be used:

```
fz_shdr_get_tspace_st(st);
st.y = fz_shdr_saw_tooth(st.y,1.0);
val = fz_shdr_smooth_step(0.4,0.6,st.y);
col = val * white + (1.0 - val) * black;
```

If `st.y` is less than 0.4 `fz_shdr_smooth_step` returns 0.0 and the color computation yields :

```
col = 0.0 * white + (1.0 - 0.0) * black;
```

which is all black. If `st.y` is greater than 0.6 `fz_shdr_smooth_step` returns 1.0 and the color computation yields :

```
col = 1.0 * white + (1.0 - 1.0) * black;
```

which is all white. In the zone where `st.y` is between 0.4 and 0.6 black and white are mixed. More black is used as `st.y` approaches 0.4 and more white is used as it approaches 0.6.This creates a smooth color transition from black to white.

Naturally, the smooth step function is not limited to the context of blending colors. It is just as useful to create smooth transitions between opaque and transparent areas in a transparency shader and between high and low areas in a bump shader.

Another method to create smooth transitions is the API

```
fz_shdr_spline_color(val,ncolors,colors,color_out);
```

It computes a smoothly blended color from a list of individual colors. The first argument is a parametric value that must be in the range of 0.0 to 1.0. For example, if there are four colors, and the val argument is below 0.25, the first color is returned. If val is around 0.25, a mixture between the first and second color is returned. If it is between 0.25 and 0.5 the second color is returned, etc. This function can be combined with a turbulence function to create a pattern of random colored spots.

```
fz_shdr_get_tspace_st(st);
val = fz_shdr_turbulance_2d(st,3,FZ_SHDR_TURB_TYPE_BETTER,0);
fz_shdr_spline_color(val,5,colors_in,color_out);
```

3.7.4 Tool Scripts

Tool scripts are extensions that complement the **form•Z** tool set and behave consistently with the **form•Z** tools. They appear in the **form•Z** interface in the icon tool palettes just like a **form•Z** tool. Tools can either be ***operators*** or ***modifiers***. An operator creates or edits the **form•Z** project data (objects, lights, etc.) through graphic manipulation in the **form•Z** project window. A modifier is a tool that controls a setting that affects a group of operators. For example, the self/copy modifier tools affect how the transformation operator tools function. Modifiers are never implemented as a single tool but rather a set of tools that have a number of modifiers representing different options and a set of operators that are sensitive to the selected modifier.

The user selects a tool from a tool icon menu or via a key shortcut to make it the active tool. A click (or multiple clicks) in the project window or input in the prompt palette is used to execute the tool. Tools are dependent on a project window and are expected to function on the provided project window. Tools are unavailable when there is no open project window.

Tools my have user controlled options associated with them. These options appear in the tool options palette when the tool is active . The options can also be accessed in a dialog that is invoked by double clicking on the tool's icon or by ***right***-clicking on the tool's icon. The dialog can also be invoked by pressing ***option*** (Macintosh) or ***ctrl+shift*** (Windows) while clicking on the tool's icon.

Tools are very flexible and can do a variety of things. Object ***creation***, ***editing*** and ***derivation*** operations are common uses of tools. In an object creation tool, input from the user in the form of clicks and/or prompt entry is used to construct an object. To create an interactive tool, a base object should be constructed as early in the tool as possible and then refined as additional input is acquired.

An editing operation modifies existing objects. A derivative operation uses existing objects as a starting point to create new objects. Both of these operations need to execute pick operations to select the objects (or other topological levels) to operate on. The tools should support the prepick and postpick model that is standard in **form•Z**.

The graphic image of the icon is supplied by the script. If one is not provided, a default script icon is supplied by **form•Z**. The script can also specify where in the tool palette the icon for the tool is positioned. If a position is not provided, then the tool is placed at the bottom of the tool palette. The icons for tool scripts appear at the bottom of the Tool Set in the Icons Customization dialog. It can be customized as with any **form•Z** tool. All tools appear in the Key Shortcuts Manager dialog so that they may have key shortcuts assigned for them.

The Scripts directory in the **form•Z** application folder contains a sample script tool : star_tool.fsl. It creates star shaped objects with interactive or preset user input.

Tool script type

Tool scripts are defined by tagging the script in its header with the `script_type` keyword and the proper identifier as follows :

```
script_type FZ_TOOL_EXTS_TYPE
```

Tool call back function set.

Tool scripts are implemented by defining a set of callback functions. There are a total of twenty-four callback functions. Note that some of these functions are optional and some are mutually exclusive hence a script would never implement all of these functions. Each of these functions is described in the following sections. As with all other script types, the tool script may implement the `fz_script_cbak_info` callback function, which defines basic information about the script. This is discussed in more detail in section 3.3.

The tool initialization function (optional)

```
long fz_tool_cbak_init();
```

This function is called by **form•Z** once when the script is successfully loaded and registered. The initialization function is where the script should initialize any data that may be needed by the other callback functions. This function is called by **form•Z** once when the plugin is successfully loaded and registered. The initialization function is where the plugin should initialize any data that may be needed by the other functions in the function set. If the tool is an editing operation which creates new objects from selected objects, the staus of objects options for the tool needs to be initialized by calling `fz_sys_cmnd_set_status_of_objt` in the tool's initialization funtion.

```
long fz_tool_cbak_init()
{
	long		err = FZRT_NOERR;

	/* Do initialization here */

	return(err);
}
```

The tool finalization function (optional)

```
long fz_tool_cbak_finit();
```

This function is called by **form•Z** once when the script is unloaded when **form•Z** is quitting. This is the complementary function to the initialization function. This function should be used to perform any cleanup operations.

```
long fz_tool_cbak_finit()
{
	long		err = FZRT_NOERR;

	/* Perform cleanup here */

	return(err);
}
```

The tool info function (required)

```
long fz_tool_cbak_info(
	mod fz_tool_kind_enum	kind,
	mod fz_proj_level_enum	level
	);
```

This function is called by **form•Z** once when the script is successfully loaded to determine the kind and level of the tool that is implemented by the script. The `kind` parameter indicates if the tool is an operator (`FZ_TOOL_KIND_OPERATOR`) or a modifier (`FZ_TOOL_KIND_MODIFIER`). **form•Z** uses the value in this parameter to determine how the icons are handled when they are selected by the user.

The `level` parameter indicates the context of the tool. `FZ_PROJ_LEVEL_MODEL` indicates that the tool operates on the projects modeling content (objects for example). `FZ_PROJ_LEVEL_MODEL_WIND` indicates that the tool operates on modeling window specific content (views for example) of modeling windows. `FZ_PROJ_LEVEL_DRAFT` indicates that the tool operates on the projects drafting content (elements for example). `FZ_PROJ_LEVEL_DRAFT_WIND` indicates that the tool operates on drafting window specific content (views for example) of drafting windows. **form•Z** uses the value in this parameter to determine which tool palette to add the icon for the tool script.

```
long fz_tool_cbak_info(
       mod fz_tool_kind_enum      kind,
       mod fz_proj_level_enum     level
       )
{
       long         err = FZRT_NOERR;

       /* set kind and level for the tool */
       kind  = FZ_TOOL_KIND_OPERATOR;
       level = FZ_PROJ_LEVEL_MODEL;

       return(err);
}
```

The tool name function (recommended)

```
long fz_tool_cbak_name(
       mod fz_string_td    name,
       long                max_len
       );
```

This function is called by **form•Z** to get the name of the tool. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the tool name string is stored in a .fzr file so that it is localizable. This function is recommended for all tool scripts. If this function is not provided , the name of the script file is used.

```
long fz_tool_cbak_name(
       mod fz_string_td    name,
       long                max_len
       )
{
       long         err = FZRT_NOERR;

       /* Get the title string "My Tool" from the script's resource file */
       err = fzrt_fzr_get_string(my_rfzr_refid, 1, 1, name);

       return(err);
}
```

The tool help function (optional)

```
long fz_tool_cbak_help(
       mod fz_string_td    help,
       long                max_len
       );
```

This function is called by **form•Z** to display a help string that describes the detail of what the tool does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a string which can handle up to `max_len` characters. It is recommended that the tool name is stored in a .fzr file so that it is localizable . The display area for help is limited so **form•Z** currently will ask for no more than 256 bytes (characters).

```
long fz_tool_cbak_help(
	mod fz_string_td	help,
	long			max_len
	)

{
	long		err = FZRT_NOERR;

	/* Get the help string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, help);

	return(err);
}
```

The tool available function (optional)

```
long fz_tool_cbak_avail(
	long			windex,
	mod long		rv
	);
```

This function is called by **form•Z** at various times to see if the tool is available. This is useful if the tool is dependent on certain conditions and it is desirable to restrict its use when the conditions are not currently satisfied. If the tool is not available, then it is shown as inactive (dimmed) in the **form•Z** tool palette. Key shortcuts are also disabled for the tool when it is not available. If this function is not provided then the tool is always available.

Availability is determined by the value that is returned by the `rv` parameter. A value of 1 indicates that the tool is available, a value of 0 indicates that the tool is unavailable.

```
long fz_tool_cbak_avail(
	long			windex,
	mod long		rv
	)
{
	long		err = FZRT_NOERR;

	/* return 1 for available, 0 for not available */
	rv = 1;

	return(err);
}
```

The tool active function (required for modifiers, not used for operators)

```
long fz_tool_cbak_active(
	long			windex,
	mod long		rv
	);
```

This function is called by **form•Z** at various times to see if the modifier tool is active. This is used by **form•Z** to draw the icon in the selected state. The value that is returned by the `rv` parameter determines if the tool is active or not. A value of 1 indicates that the tool is active, a value of 0 indicates that the tool is inactive.

```
long fz_tool_cbak_active(
```

```
        long                windex,
        mod long            rv
        )
{
        long          err = FZRT_NOERR;

        /* return 1 for active, 0 for not active */
        if(my_modifier_state == 2) rv = 1;
        else                       rv = 0;

        return(err);
}
```

The tool select function (optional)

```
long fz_tool_cbak_select(
        long                windex
        );
```

This function is called by **form•Z** when the tool is selected from the tool icon palette or when a key shortcut for the tool is invoked.

For operator tools, the select function is where any tool specific preparation occurs for the execution of the tool (which is triggered by a click in the project window). The select function should set the prompt string (in the prompts palette) for the tool. The select function is also called after the execution of the tool to prepare it for the next execution.

The following example shows the select function for an operator tool that draws a line. It starts by asking for the origin point for an object in the prompts palette. Note the prompt string is shown here for readability. It should be stored in a .fzr resource file and loaded with `fzrt_get_string` to support localization.

```
long fz_tool_cbak_select(
        long                windex
        )
{
        fz_string_td        prompt_str;
        long                err = FZRT_NOERR;

        /* Get the prompt string "First point:" from the script's resource file
*/
        if((err = fzrt_fzr_get_string(my_rfzr_refid, 1, 3, prompt_str)) ==
FZRT_NOERR)
        {
                err = fz_fuim_prompt_line(
                        prompt_str,                 /* prompt string */
                        FZ_FUIM_PROMPT_LINE_NEXT,  /* place it on the next line */
                        FZ_FUIM_PROMPT_EDIT_XYZ);  /* set the edit mode of prompt */
        }
        return(err);
}
```

The following example shows the select function for a tool that starts by asking the user to select an object. Note that the prompt handles prepick and postpick by checking the state of the pick buffer.

```
long fz_tool_cbak_select(
        long                windex
        )
{
```

```
	fz_string_td		prompt_str;
	fzrt_boolean		pre_pick;
	long			i,npick;
	fz_model_pick_enum	pkind;
	long			err = FZRT_NOERR;

	/* Get the number of picked entities */
	fz_model_pick_get_count(windex,npick);

	/* loop through picked entities */
	for(i = 0; i < npick; i++)
	{
		/* get one picked entity */
		fz_model_pick_get_data(windex,i,pkind,NULL,NULL,NULL);

		/* check if it was picked at the object level */
		if ( pkind == FZ_MODEL_PICK_OBJT )
		{	pre_pick = TRUE;
			break;
		}
	}

	/* check if it was picked at the object level */
	if(pre_pick)
	{	/* Get the string "Click to frame selected objects" */
		err = fzrt_fzr_get_string(my_rfzr_refid, 1, 4, prompt_str);
	}
	else
	{	/* Get the string "Select object to frame" */
		err = fzrt_fzr_get_string(my_rfzr_refid, 1, 5, prompt_str);
	}

	err = fz_fuim_prompt_line(
		prompt_str,			/* prompt string */
		FZ_FUIM_PROMPT_LINE_NEXT,	/* place it on the next line */
		FZ_FUIM_PROMPT_EDIT_NONE);	/* set the edit mode of prompt to
	none */


	return(err);
}
```

For modifier tools, the select function should change the state of the modifier to the desired value for the selected icon. The modifier is usually a global variable in the script that can be accessed by the tools that use it.

```
long fz_tool_cbak_select(
	long			windex
	)
{
	long		err = FZRT_NOERR;

	/* Set modifier state for the tool */
	my_modifier_state = 2;

	return(err);
}
```

The tool click function (required for operators, not used for modifiers)

```
long fz_tool_cbak_click (
	long				windex,
```

```
	fzrt_point                        where,
	fz_xyz_td                         where_3d,
	fz_map_plane_td                   map_plane,
	fz_fuim_click_enum                clicks,
	long                              click_count,
	mod fzrt_boolean                  click_handled,
	mod fz_fuim_click_wait_enum       click_wait,
	mod fzrt_boolean                  done
	);
```

This function is called by **form•Z** for operators when the tool is the active tool and a click occurs in the active project window. This function is called by **form•Z** for each click in the project window until TRUE is returned in the `done` parameter (or from the `fz_tool_cbak_prompt` function) or the user cancels the operation.

The `windex` parameter is the active window. The `where` parameter indicates in 2 dimensional screen space where the mouse was clicked. The `where_3d` parameter indicates the 3 dimensional location in world space where the mouse was clicked. This is a point on the active reference plane provided in the `map_plane` parameter. The `clicks` parameter indicates if the click is a single, double or triple click. The `click_count` parameter is the number of clicks since the start of the tool. This value starts at 1 for the first click and increases with each click of the mouse.

The `click_handled` parameter should be set to TRUE if the click function handled the click and it should be set to FALSE if the function did not handle the click. The default value is TRUE. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the click function again. The default is FZ_FUIM_CLICK_WAIT_NOT. The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

The following example shows the click function for a tool that draws a line. The first click creates a new object with a single segment (edge) with identical start and end points at the click point. The second click fixes the end point at the click point. This is done in this manner to accommodate the track function (see following section). If a track function is not provided then the object does not need to be created until the final click. In this situation, the click points could be accumulated into a buffer and then used to create the object. Note that this is not an ideal interface for the user as they will get no interactive feedback during the operation. If performance is a concern because of the complexity of the operation, then a proxy should be used so that the user gets some feedback during the tools execution.

```
fz_objt_ptr      line_obj;
fz_xyz_td        line_points[3];

long fz_tool_cbak_click (
   long                           windex,
   fzrt_point                     where,
   fz_xyz_td                      where_3d,
   fz_map_plane_td                map_plane,
   fz_fuim_click_enum             clicks,
   long                           click_count,
   mod fzrt_boolean               click_handled,
   mod fz_fuim_click_wait_enum    click_wait,
   mod fzrt_boolean               done
   )
{
   long          err = FZRT_NOERR;
   fz_string_td        prompt_str;
   long          pindx[2];
```

```
    if(click_count == 1)                        /* handle first click */
    {
        /* make new object */
        if((err = fz_objt_cnstr_objt_new(windex,line_obj)) == FZRT_NOERR )
        {
            /* construct line object */
            line_points[0] = where_3d;
            line_points[1] = where_3d;
            fz_objt_fact_add_pnts(windex,line_obj,line_points,2);

            pindx[0] = 0;
            pindx[1] = 1;
            fz_objt_fact_create_wire_face(windex,line_obj,pindx,2,NULL);

            /* add object to the project */
            err = fz_objt_add_objt_to_project(windex,line_obj);

            if ( err != FZRT_NOERR )
            {   fz_objt_edit_delete_objt(windex,line_obj);
            }
            else
            {
                /* Get the string "Second point:" */
                err = fzrt_fzr_get_string(my_rfzr_refid, 1, 6, prompt_str);

                /* set prompt for next point */
                fz_fuim_prompt_line(prompt_str,
                                    FZ_FUIM_PROMPT_LINE_NEXT,
                                    FZ_FUIM_PROMPT_EDIT_XYZ);
            }
        }
    }
    else if(click_count == 2)                   /* handle second click */
    {
        /* reset object and construct with new second point */
        fz_objt_fact_reset(windex, line_obj);
        line_points[1] = where_3d;
        fz_objt_fact_add_pnts(windex,line_obj,line_points,2);

        pindx[0] = 0;
        pindx[1] = 1;
        fz_objt_fact_create_wire_face(windex,line_obj,pindx,2,NULL);

        done = TRUE;                /* tool complete */
    }

    return(err);
}
```

If the operation requires the picking (selection) of objects (or other topological levels), then this should be handled following the **form•Z** prepick and postpick standard. That is for each click the pick buffer is inspected to see if the requirements have been satisfied for the operation (prepick). If it is not satisfied, the function fz_model_pick is called to handle the click as a postpick and then the pick buffer is re-inspected. If the pick requirements have been satisfied with the prepick or postpick then the operation completes. The prompts palette should also be updated in the click function to reflect the desired user actions using the `fz_fuim_prompt_line` function.

```
long fz_tool_cbak_click (
        long                            windex,
        fzrt_point                      where,
        fz_xyz_td                       where_3d,
        fz_map_plane_td                 map_plane,
```

```
	fz_fuim_click_enum		clicks,
	long				click_count,
	mod fzrt_boolean		click_handled,
	mod fz_fuim_click_wait_enum	click_wait,
	mod fzrt_boolean		done
	)
{
	fz_objt_ptr	pick_obj1,pick_obj2;
	long		npick;
	fz_model_pick_enum pkind1,pkind2;
	long		err = FZRT_NOERR;

	done = FALSE;

	/* Get the number of picked entities */
	fz_model_pick_get_count(windex,npick);
	if(npick < 2)
	{
		/* use the click to pick an object */
		fz_model_pick(windex,where,FZ_MODEL_PICK_OBJT);
		fz_model_pick_get_count(windex,npick);
	}

	/* check if enough picked to execute operation */
	if(npick >= 2)
	{
		/* get first two objects from pick buffer */
		fz_model_pick_get_data(windex,0,pkind1,NULL,pick_obj1,NULL);
		fz_model_pick_get_data(windex,1,pkind2,NULL,pick_obj2,NULL);
		if(pkind1 == FZ_MODEL_PICK_OBJT && pkind2 == FZ_MODEL_PICK_OBJT)
		{
			/** operate on objects here **/
		}

		done = TRUE;
	}

	return(err);
}
```

If the tool is an editing operation which creates new objects from selected objects, the status of objects functionality should be implemented. This can be done easily with two api calls: `fz_objt_edit_handle_status_of_opnd` and `fz_objt_edit_handle_new_objt_volms`. These two functions correspond directly to the options in the Status Of Objects palette. Note that the tool also needs to initialize its status of objects option in the `fz_tool_cbak_init` callback function by calling `fz_syst_cmnd_set_status_of_objt` with the appropriate arguments.

The tool prompt function (required for operators, not used for modifiers)

```
long fz_tool_cbak_prompt (
	long				windex,
	fz_xyz_td			prompt_value,
	fz_string_td			prompt_string,
	fz_map_plane_td			map_plane,
	long				click_count,
	mod fzrt_boolean		prompt_handled,
	mod fz_fuim_click_wait_enum	click_wait,
	mod fzrt_boolean		done
	);
```

This function is called by **form•Z** when the tool is the active tool and the user makes input in an editable prompt string in the prompts palette. This function is very similar to the click function and each input of data in the prompts palette is treated by **form•Z** the same as a click. This function is called by **form•Z** each time the user enters data in the prompts palette and then presses the enter or return keys. Like the click function, this function is called until TRUE is returned in the `done` parameter (or TRUE is returned in the `done` parameter from the click function) or the user cancels the operation.

The windex parameter is the active window. The `prompt_value` and `prompt_string` parameters are the users input from the prompts palette. An editable prompt is created by calls to the `fz_fuim_prompt_line` function in the select function, click function, undo function, redo function or previous click handling in the prompt function. Editable input is specified by the last parameter to the `fz_fuim_prompt_line` function. This parameter instructs the prompts palette as to what type of input is desired (if any). The following table shows the available options.

Name	Description
FZ_FUIM_PROMPT_EDIT_NONE	No editable text in prompt string
FZ_FUIM_PROMPT_EDIT_XY	Standard 2D world Cartesian coordinate
FZ_FUIM_PROMPT_EDIT_XYZ	Standard 3D world Cartesian coordinate
FZ_FUIM_PROMPT_EDIT_ANGLE	Angular dimension
FZ_FUIM_PROMPT_EDIT_LINEAR_X	Linear dimension
FZ_FUIM_PROMPT_EDIT_LINEAR_XY	Linear 2D
FZ_FUIM_PROMPT_EDIT_LINEAR_XYZ	Linear 3D
FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_X	Linear dimension, displayed in decimal format.
FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XY	Linear 2D, displayed in decimal format.
FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XYZ	Linear 3D, displayed in decimal format.
FZ_FUIM_PROMPT_EDIT_STRING	string

Note that the `FZ_FUIM_PROMPT_EDIT_STRING` does not return a value for the `prompt_value` parameter. Instead the raw string is returned in the `prompt_string` parameter. The `prompt_value` parameter is interpreted based on the type of the prompt edit shown in the above table. If the prompt edit is `FZ_FUIM_PROMPT_EDIT_ANGLE, FZ_FUIM_PROMPT_EDIT_LINEAR_X, or FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_X`, then the value is found in the first field (x). If the prompt edit is `FZ_FUIM_PROMPT_EDIT_XY, FZ_FUIM_PROMPT_EDIT_LINEAR_XY, or FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XY`, then the values are found in the first two fields (x and y). If the prompt edit is `FZ_FUIM_PROMPT_EDIT_XYZ, FZ_FUIM_PROMPT_EDIT_LINEAR_XYZ, or FZ_FUIM_PROMPT_EDIT_LINEAR_DECIMAL_XYZ`, then the values are found all three fields (x, y and z).

The `map_plane` parameter is the active reference plane. The `click_count` parameter is the number of clicks (or prompts) since the start of the tool. This value starts at 1 for the first click (or prompt) and increases with each click (or prompt).

The `prompt_handled` parameter should be set to TRUE if the prompt function handled the prompt and it should be set to FALSE if the function did not handle the prompt. The default value is TRUE. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the next click function. The default is FZ_FUIM_CLICK_WAIT_NOT. The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

The following example shows the prompt function for a tool that draws a line. The prompt function is very similar to the click function in the previous line tool example. In the prompt function the coordinate location comes from the `prompt_value` parameter rather than the click point.

```
fz_objt_ptr         line_obj;
fz_xyz_td           line_points[3];

long fz_tool_cbak_prompt(
   long                          windex,
   fz_xyz_td                            prompt_value,
   fz_string_td                  prompt_string,
   fz_map_plane_td               map_plane,
   long                          click_count,
   mod fzrt_boolean              prompt_handled,
   mod fz_fuim_click_wait_enum   click_wait,
   mod fzrt_boolean              done
   )
{
   long         err;
   fz_string_td prompt_str;
   long         pindx[2];

   if(click_count == 1)                        /* handle first click */
   {
      /* make new object */
      if((err = fz_objt_cnstr_objt_new(windex,line_obj)) == FZRT_NOERR )
      {
         /* construct line object */
         line_points[0] = prompt_value;
         line_points[1] = prompt_value;
         fz_objt_fact_add_pnts(windex,line_obj,line_points,2);

         pindx[0] = 0;
         pindx[1] = 1;
         fz_objt_fact_create_wire_face(windex,line_obj,pindx,2,NULL);

         /* add object to the project */
         err = fz_objt_add_objt_to_project(windex,line_obj);

         if ( err != FZRT_NOERR )
         {  fz_objt_edit_delete_objt(windex,line_obj);
         }
         else
         {
            /* Get the string "Second point:" */
            err = fzrt_fzr_get_string(my_rfzr_refid, 1, 6, prompt_str);

            /* set prompt for next point */
            fz_fuim_prompt_line(prompt_str,
                                FZ_FUIM_PROMPT_LINE_NEXT,
                                FZ_FUIM_PROMPT_EDIT_XYZ);
         }
      }
   }
   else if(click_count == 2)            /* handle second click */
   {
      /* reset object and construct with new second point */
      fz_objt_fact_reset(windex, line_obj);
      line_points[1] = prompt_value;
      fz_objt_fact_add_pnts(windex,line_obj,line_points,2);

      pindx[0] = 0;
      pindx[1] = 1;
      fz_objt_fact_create_wire_face(windex,line_obj,pindx,2,NULL);
```

```
        done = TRUE;        /* tool complete */
    }
}
```

The tool track function (optional, not used for modifiers)

```
long fz_tool_cbak_track(
        long                        windex,
        fzrt_point                  where,
        fz_xyz_td                   where_3d,
        fz_map_plane_td             map_plane,
        long                        click_count
        );
```

This function is called by **form•Z** when the tool is the active tool and the mouse is moved in the active project window after the first click. This function is used to update any interactive input as the mouse moves in the window. In general this function performs the same action as the next click would allowing the input to appear interactive

The `windex` parameter is the active window. The `where` parameter indicates in 2 dimensional screen space where the cursor is located. The `where_3d` parameter indicates the 3 dimensional location in world space where the cursor is located. This is a point on the active reference plane provided in the `map_plane` parameter. The `click_count` parameter is the number of clicks since the start of the tool (first click).

The following example shows the track function for a tool that draws a line. This complements the previous line tool example for the click and prompt functions. In this function the location of the second point is updated to the current cursor location.

```
fz_objt_ptr                 line_obj;
fz_xyz_td                   line_points[3];

long fz_tool_cbak_track(
        long                        windex,
        fzrt_point                  where,
        fz_xyz_td                   where_3d,
        fz_map_plane_td             map_plane,
        long                        click_count
        )
{
        long                err = FZRT_NOERR;
        long                pindx[2];

        if(click_count == 1)
        {
                /* reset object and construct with new second point */
                fz_objt_fact_reset(windex, line_obj);
                line_points[1] = where_3d;
                fz_objt_fact_add_pnts(windex,line_obj,line_points,2);

                pindx[0] = 0;
                pindx[1] = 1;
                fz_objt_fact_create_wire_face(windex,line_obj,pindx,2,NULL);
        }
}
```

The tool cancel function (optional)

```
long fz_tool_cbak_cancel (
        long                windex,
```

```
	long			click_count
	);
```

This function is called by **form•Z** when a tool is interrupted. A tool can be canceled by the user using the key cancel key shortcut or by **form•Z** if a **form•Z** operation id executed that cancels the current operation (selecting another tool for example). This function is used to cleanup any data that was generated during the execution of the tool.

The `windex` parameter is the active window. The `click_count` parameter is the number of clicks since the start of the tool (first click).

The following example complements the previous line tool example for the click, prompt and track functions. In this function, the object that was created in the prior functions is deleted.

```
fz_objt_ptr			line_obj;
fz_xyz_td			line_points[3];

long fz_tool_cbak_cancel (
	long			windex,
	long			click_count
	)
{
	long		err = FZRT_NOERR;

	/* delete object crated at first click */
	if(click_count  >= 1)fz_objt_edit_delete_objt(windex,line_obj);

	return(err);
}
```

The tool undo function (optional)

```
long fz_tool_cbak_undo (
	long				windex,
	long				click_count,
	mod fz_fuim_click_wait_enum	click_wait,
	mod fzrt_boolean		done
	);
```

This function is called by **form•Z** when the user selects the undo menu item from the Edit menu during the execution of the tool. This function is used to back the input up to the state of the previous click. If this function is not provided, the undo command does not perform undos during the tool.

The `windex` parameter is the active window. The `click_count` parameter is the number of clicks which will be one less than the last call to the click or prompt functions. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the click function again.

The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

```
long fz_tool_cbak_undo (
	long				windex,
	long				click_count,
	mod fz_fuim_click_wait_enum	click_wait,
	mod fzrt_boolean		done
	)
{
```

```
	long		err = FZRT_NOERR;

	/** return to previous click state here ***/

	return(err);
}
```

The tool redo function (optional)

```
long fz_tool_cbak_redo (
	long				windex,
	long				click_count,
	mod fz_fuim_click_wait_enum	click_wait
	mod fzrt_boolean		done
	);
```

This function is called by **form•Z** when the user selects the redo menu item from the Edit menu during the execution of the tool. This function is used to move the input up to the state of the previously undone click. If this function is not provided, the redo command does not perform redos during the tool. This function is only called immediately after a call to the undo function. Once a click or prompt entry occurs, the redo is reset.

The `windex` parameter is the active window. The `click_count` parameter is the number of clicks that will be one more that the last call to the undo function. The `click_wait` parameter tells **form•Z** to wait until a specific type of click happens before calling the click function again.

The `done` parameter determines the completion of the tool. A value of TRUE indicates that the tool is done, a value of FALSE indicates that the tool expects more clicks. The default is FALSE.

```
long fz_tool_cbak_redo (
	long				windex,
	long				click_count,
	mod fz_fuim_click_wait_enum	click_wait,
	mod fzrt_boolean		done
	)
{
	long		err = FZRT_NOERR;

	/** return to previously undone click state here ***/

	return(err);
}
```

The tool icon menu function (Optional, mutually exclusive with icon menu adjacent function)

```
long fz_tool_cbak_icon_menu (
	fzrt_UUID_td			icon_menu_uuid,
	mod fzrt_UUID_td		group_uuid,
	mod fz_fuim_icon_group_enum	group_pos,
	mod long			group_row,
	mod long			group_col
	);
```

This function is called by **form•Z** to add the tool to the Tool icon menu. The presence of this function places the tool in the Tool set of tools. If no other parameters are set then the tool will get added to a group of icons at the bottom (end) of the icon menu. Note that this only adds the position to the tool menu. The function `fz_tool_cbak_icon_file` must be provided to add custom graphics for the icon. If one of these is not provided, **form•Z** uses a generic icon graphic.

The group_uuid parameter is assigned to all tools that should be grouped together. That is, all fz_tool_cbak_icon_menu implemented functions that return the same group_uuid parameter are placed together in the system icon menu in the same group (pop-out tool menu). This group is added to the bottom (end) of the menu. The placement of the item in the group is controlled by the group_pos parameter. A value of FZ_FUIM_ICON_GROUP_START places the item at the start of the group and a value of FZ_FUIM_ICON_GROUP_END places it at the end of the group. Note that these may not always yield constant results because plugin and script load order can vary hence multiple uses of FZ_FUIM_ICON_GROUP_END my note build the menu in the expected order. When FZ_FUIM_ICON_GROUP_CUSTOM is selected, then the group_row and group_col parameters specify the position of the item in the tool menu group.

```
#define MY_GRUP_ID
"\x5d\xe6\x85\x41\x6b\xaa\x4f\xb4\xa5\x6a\xf5\x0e\x65\x36\xfb\xd0"

long fz_tool_cbak_icon_menu (
	fzrt_UUID_td				icon_menu_uuid,
	mod fzrt_UUID_td			group_uuid,
	mod fz_fuim_icon_group_enum	group_pos,
	mod long				group_row,
	mod long				group_col
	)
{
	long		err = FZRT_NOERR;

	fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
	group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
	group_row = 1;
	group_col = 1;

	return(err);
}
```

The function fuim_cmds_new_icon_group can be called to better control the group containing the set of tools. This adds the ability to name the group and insert the pop-out menu group in the existing menu groups. The icon pop-out menu can be created in each fz_tool_cbak_icon_menu so that if the user has disabled one of the scripts, the icon menu will still be formed properly. **form·Z** ignores attempts to create a menu when the uuid already exists that would occur if all the scripts are enabled. The following is an example of a pop-out menu.

```
long fz_tool_cbak_icon_menu (
	fzrt_UUID_td				icon_menu_uuid,
	mod fzrt_UUID_td			group_uuid,
	mod fz_fuim_icon_group_enum	group_pos,
	mod long				group_row,
	mod long				group_col
	)
{
	long		err = FZRT_NOERR;

	err = fz_fuim_exts_icon_group(
			"My  Group", MY_GRUP_ID, icon_menu_uuid,
			FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE,
			FZRT_UUID_NULL, FZ_FUIM_POS_BEFORE);

	if(err = FZRT_NOERR)
```

```
	{	fzrt_UUID_copy(MY_GRUP_ID, group_uuid);
		group_pos = FZ_FUIM_ICON_GROUP_CUSTOM;
		group_row = 1;
		group_col = 1;
	}
	return(err);
}
```

The tool icon menu adjacent function (Optional, mutually exclusive with icon menu function)

```
long fz_tool_cbak_icon_menu_adjacent(
	fzrt_UUID_td			icon_menu_uuid,
	mod fzrt_UUID_td		adjacent_uuid,
	mod fz_fuim_icon_adjacent_enum	where
	);
```

This function is called by **form•Z** to add the tool to the system icon menu. It serves the same purpose as the `fz_cmnd_cbak_proj_icon_menu` function, however it specifies the location of the icon item quite differently. The location is identified by referencing another tool in the icon menu. The `adjacent_uuid` parameter is the UUID of the tool to which the icon should be added adjacent. The `where` parameter specifies to which side of the adjacent icon the icon should be added. The available options are `FZ_FUIM_ICON_ADJACENT_TOP`, `FZ_FUIM_ICON_ADJACENT_BOTTOM`, `FZ_FUIM_ICON_ADJACENT_LEFT`, `FZ_FUIM_ICON_ADJACENT_RIGHT`. The default action is specified by `FZ_FUIM_ICON_ADJACENT_DEFAULT` which currently is the same as `FZ_FUIM_ICON_ADJACENT_RIGHT`. New pop-out groups can not be created with this function. The following example ads the icon to the right of the **form•Z** primitive spheroid tool.

```
long fz_tool_cbak_icon_menu_adjacent(
	fzrt_UUID_td			icon_menu_uuid,
	mod fzrt_UUID_td		adjacent_uuid,
	mod fz_fuim_icon_adjacent_enum	where
	)
{
	long		err = FZRT_NOERR;

	/* copy UUID of adjacent tool */
	fzrt_UUID_copy(FZ_CMND_MODEL_PRIM_SPHR, adjacent_uuid);
	where = FZ_FUIM_ICON_ADJACENT_RIGHT;

	return(err);
}
```

The tool icon file function (Optional, mutually exclusive with icon resource function)

```
long fz_tool_cbak_icon_file (
	fz_fuim_icon_enum		which,
	fzrt_floc_ptr			floc,
	mod long			hpos,
	mod long			vpos,
	fzrt_floc_ptr			floc_mask,
	mod long			hpos_mask,
	mod long			vpos_mask
	);
```

This function is called by **form•Z** to get an icon for the tool from an image file. The icon image can be in any of the **form•Z** supported image file formats or format for which an image file translator is installed. The TIFF format is the recommended format as the TIFF translator is commonly available. **form•Z** will request an icon when the tool is displayed in a tool menu using `fz_tool_cbak_icon_menu` or `fz_tool_cbak_icon_menu_adjacent`.

form•Z supports 3 styles of icon display. Recall that these are selectable by the user from the Icon Style menu in the Customize Tools dialog. The first two options (White and Gray) are generated from a black and white source graphic with different treatments at drawing time. The third option is generated from a color source graphic. The first two options are older icon styles that are provided for backward compatibility. The color icons became the default with v 4.0. Note that if an icon of one type or the other (or both) is not provided, then **form•Z** uses a generic icon graphic.

The `which` parameter indicates the type of source graphic icon that is needed by **form•Z**. For each type of icon source (black and white and color), there are two possible sizes. The full size icon is the size that is used in the main tool palettes and tear off tool palettes. The black and white source full size is 30 x 30 pixels and indicated by `FZ_FUIM_ICON_MONOC`. The color source is 32 x 32 pixels and indicated by `FZ_FUIM_ICON_COLOR`. The alternate size is the smaller size used for window icons that are drawn in the lower margin of the window. The alternate size for both black and white and color sources is 20 x 16 pixels and indicated by `FZ_FUIM_ICON_MONOC_ALT` and `FZ_FUIM_ICON_COLOR_ALT` respectively.

The `floc` parameter should be filled with the file name and location of the file that contains the icon graphic. The `hpos` and `vpos` parameters should be set to the left and top pixel location of icon data in the file respectively. It is recommended that the icon file be in the same directory as the script file. This makes it simple to find the file. The location of the script file can be acquired using the `fz_script_file_get_floc` API function.

The `floc_mask` parameter should be filled with the file name and location of the file that contains the icon mask (this can be the same file as the `floc` parameter). The icon mask defines the transparent areas of the icon. The `hpos_mask` and `vpos_mask` parameters should be set to the left and top pixel location of icon mask data in the file respectively. If a mask is not provided than the entire background of the icon will be drawn.

A single file can be used for multiple icons across a variety of tools by creating a grid of icons in the file and specifying the location for each icon in the corresponding provided function.

```
long fz_tool_cbak_icon_file (
	fz_fuim_icon_enum			which,
	fzrt_floc_ptr				floc,
	mod long				hpos,
	mod long				vpos,
	fzrt_floc_ptr				floc_mask,
	mod long				hpos_mask,
	mod long				vpos_mask
	)
{
	long	err = FZRT_NOERR;

	switch(which)
	{
		case FZ_FUIM_ICON_MONOC :
			err = fz_script_file_get_floc(floc);
			if(err == FZRT_NOERR)
			{	err = fzrt_file_floc_set_name(floc,"my_icon_bw.tif");
				hpos = 0;
				vpos = 0;
			}
```

```
			break;
			case FZ_FUIM_ICON_COLOR :
				err = fz_script_file_get_floc(floc);
				if(err == FZRT_NOERR)
				{	err = fzrt_file_floc_set_name(floc,"
			my_icon_col.tif");
					hpos = 0;
					vpos = 0;
				}
			break;
		}
		return(err);
}
```

The tool preferences IO function (optional)

```
long fz_tool_cbak_pref_io (
		fz_iost_ptr				iost,
		fz_iost_dir_td_enum				dir,
		mod long				version,
		long					size
		);
```

form•Z calls this function to read and write any tool specific data to a **form•Z** preference file. This function is called when reading and writing user specified preference files (Save Preferences button in the Preferences dialog). It is also called by **form•Z** when reading and writing the session to session preference file maintained by **form•Z**. The file IO is performed using the IO streams (iost) interface. This interface provides functions for reading and writing data from a file (stream) and handles all cross platform endian issues. The `iost` parameter is the pointer to the preference file and should be used in all IO Stream function calls. The IO Stream functions available to scripts are fully documented in the **form•Z** API reference.

The `dir` parameter indicates if the file is being written with a value of `FZ_IOST_WRITE` or read with a value of `FZ_IOST_READ`. The `version` parameter should return the version of the data that is written when writing a file. When reading a file, the `version` parameter contains the version of the data that was written to the file (and hence being read). The `size` parameter is only valid when `dir == FZ_IOST_READ` (read). This is the size of the data that was written in the file.

It is the responsibility of the script to maintain version changes of the script data. In the following example, in its first release, a tools data consisted of four long integer values, a total of 16 bytes. When written, the version reported back to **form•Z** was 0. In a subsequent release, a fifth long integer is added to increase the size to 20 bytes. When writing this new data, the version reported to **form•Z** needs to be increased. When reading a file with the old version of the tool preference, **form•Z** will pass in the version number of the attribute when it was written, in this case 0. This indicates to the script, that only four integers, 16 bytes, need to be read and the fifth integer should be set to a default value.

```
long fz_tool_cbak_pref_io (
	fz_iost_ptr			iost,
	fz_iost_dir_td_enum			dir,
	mod long			version,
	long				size
	)
{
	long	err = FZRT_NOERR;

	if ( dir == FZ_IOST_WRITE ) version = 1;
```

```
    err = fz_iost_one_long(iost,my_tool_value1);
    if(err == FZRT_NOERR)
    {   err = fz_iost_one_long(iost,my_tool_value2);
        if(err == FZRT_NOERR)
        {   err = fz_iost_one_long(iost,my_tool_value3);
            if(err == FZRT_NOERR)
            {   err = fz_iost_one_long(iost,my_tool_value4);

                if(version >= 1)
                {   err = fz_iost_one_long(iost,my_tool_value5);
                }
            }
        }
    }

    return(err);
}
```

The tool options name function (Optional)

```
long fz_tool_cbak_opts_name(
        mod fz_string_td    name,
        long                max_len
        );
```

This function is called by **form•Z** to get the name of the tools options. The name is shown in various places in the **form•Z** interface including the key shortcuts manager dialog. It is recommended that the tool name is stored in a .fzr file so that it is localizable

```
long fz_tool_cbak_opts_name(
        mod fz_string_td    name,
        long                max_len
        )
{
        long          err = FZRT_NOERR;

        /* Get the title str “My Tool Options” from the script’s resource file */
        err = fzrt_fzr_get_string(my_rfzr_refid, 1, 11, name);

        return(err);
}
```

The tool options uuid function (optional)

```
long fz_tool_cbak_opts_uuid (
        mod fzrt_UUID_td    uuid
        );
```

This function is called by **form•Z** to get the uuid of the tools options. This unique id is used by **form•Z** to distinguish the tool options from other tool options . This function is recommended for all tool scripts. If a UUID is not provided, one will be generated internally by **form•Z**. in this situation the UUID will not be the same each time **form•Z** is run and hence persistent information will not be retained. This any user customization like key shortcuts.

```
#define MY_TOOL_OPTS_ID
        "\xc1\x29\xc9\x71\x87\x16\x43\x19\xb9\xa5\x96\xe4\x1d\xe1\x7e\xb9"

long fz_tool_cbak_opts_uuid (
        mod fzrt_UUID_td    uuid
```

```
	)
{
	long		err = FZRT_NOERR;

	/* copy constant UUID to into the uuid parameter */
	fzrt_UUID_copy(MY_TOOL_OPTS_ID, uuid);

	return(err);
}
```

The tool options help function (optional)

```
long fz_tool_cbak_opts_help(
	mod fz_string_td	help,
	long			max_len
	);
```

This function is called by **form•Z** to display a help string that describes the detail of what the tool does. This string is shown in the key shortcut manager dialog and the help dialogs. The `help` parameter is a pointer to a string which can handle up to `max_len` characters. It is recommended that the tool help string is stored in a .fzr file so that it is localizable . The display area for help is limited so **form•Z** currently will ask for no more than 256 characters.

```
long fz_tool_cbak_opts_help(
	mod fz_string_td	help,
	long			max_len
	)

{
	long		err = FZRT_NOERR;

	/* Get the help string from the script's resource file */
	err = fzrt_fzr_get_string(my_rfzr_refid, 1, 2, help);

	return(err);
}
```

The tool options interface template function (optional)

```
long fz_tool_cbak_opts_iface_tmpl(
	fz_fuim_tmpl_ptr		tmpl_ptr
	);
```

This function is called by **form•Z** when the interface for the tool options is needed. This template is displayed inside the tool options palette when the tool is active and in a dialog when the user invokes the dialog from the icon. The **form•Z** interface template functions should be called to construct the interface of the palette in this function. Please see section XXX for more details on the fuim template functions that are available for scripts. As scripts are more limited in scope than plugins, the range of fuim functions is smaller and only certain dialog interface items can be constructed by a palette script.

The following sample is a template which creates a number of different interface items.

```
#define MY_TOOL_RSRC_ID 1
#define MY_TOOL_TOOL_OPTS_NAME 1
#define MY_TOOL_BASE_TYPE 2
#define MY_TOOL_DYNAMIC 3
#define MY_TOOL_PRESET 4
#define MY_TOOL_RADIUS 5
```

```
#define MY_TOOL_RATIO 6

long my_tool_base_type;
fzrt_boolean my_tool_tool_opts_fixed;
double my_tool_radius;
double my_tool_ratio;

long fz_tool_cbak_opts_iface_tmpl(
   fz_fuim_tmpl_ptr    tmpl_ptr
   )
{
   long         i,g1;
   fz_string_td menu_items[],name;
   long         err;

   fzrt_fzr_get_string(my_rfzr_refid,MY_TOOL_RSRC_ID,
      MY_TOOL_TOOL_OPTS_NAME,name);

   if((err = fz_fuim_script_tmpl_init(tmpl_ptr,name,0,
            MY_TOOL_TMPL_UUID,0)) == FZRT_NOERR )
   {
      for(i = 0; i < 8; i++)
      {  fzrt_fzr_get_menu_string(my_rfzr_refid,1,i+2,menu_items[i]);
      }

      fzrt_fzr_get_string(my_rfzr_refid,MY_TOOL_RSRC_ID,
               MY_TOOL_BASE_TYPE,name);
      g1 = fz_fuim_script_new_menu(tmpl_ptr,FZ_FUIM_ROOT,FZ_FUIM_FLAG_NONE,
            FALSE,name,menu_items,8);
      fz_fuim_script_item_range_long(tmpl_ptr,g1,my_tool_base_type,0,7,
         FZ_FUIM_FORMAT_INT_DEFAULT,FZ_FUIM_RANGE_NONE);

      fzrt_fzr_get_string(my_rfzr_refid,MY_TOOL_RSRC_ID,
               MY_TOOL_DYNAMIC,name);
      g1 = fz_fuim_script_new_radio(tmpl_ptr,FZ_FUIM_ROOT,
               FZ_FUIM_FLAG_NONE,name);
      fz_fuim_script_item_unary_bool(tmpl_ptr, g1, my_tool_tool_opts_fixed, 0);

      fzrt_fzr_get_string(my_rfzr_refid,MY_TOOL_RSRC_ID,
               MY_TOOL_PRESET,name);
      g1 = fz_fuim_script_new_radio(tmpl_ptr,FZ_FUIM_ROOT,
               FZ_FUIM_FLAG_NONE,name);
      fz_fuim_script_item_unary_bool(tmpl_ptr, g1, my_tool_tool_opts_fixed, 1);

      fzrt_fzr_get_string(my_rfzr_refid,MY_TOOL_RSRC_ID,
               MY_TOOL_RADIUS,name);
      g1 = fz_fuim_script_new_text_edit(tmpl_ptr,FZ_FUIM_ROOT,
               FZ_FUIM_FLAG_NONE,name);
      fz_fuim_script_item_range_double(tmpl_ptr,g1,my_tool_radius,0.0,0.0,
         FZ_FUIM_FORMAT_FLOAT_DISTANCE,FZ_FUIM_RANGE_MIN);

      fzrt_fzr_get_string(my_rfzr_refid,MY_TOOL_RSRC_ID,
               MY_TOOL_RATIO,name);
      fz_fuim_script_new_slider_edit_pcent_double(tmpl_ptr,FZ_FUIM_ROOT,
               name,my_tool_ratio,
               0.0,1.0,0.0,100.0,
               FZ_FUIM_RANGE_MIN | FZ_FUIM_RANGE_MIN_INCL |
               FZ_FUIM_RANGE_MAX | FZ_FUIM_RANGE_MAX_INCL,NULL);

   }
   return(err);
}
```

3.7.5 Utility Scripts

Utility scripts are designed to execute a task which is either less frequently used or an item in the **form•Z** interface is not desired. Utility scripts are best used on tasks that are linear in nature (like batch processing). Utility scripts are not loaded by **form•Z** at startup. This allows **form•Z** to start up faster and use less memory. Utility scripts are not listed in the Extensions Manager dialog and they do not need to be located in the Extensions Manager's search paths.

The user invokes a utility script by selecting the Run Utility… item from the Extensions menu. The user is then prompted with a standard file open dialog to select the scripts executable file (.fsb) or script source file (.fsl) to run. If a source file is selected, it is first compiled to create the script executable. If any compile errors occur, the utility is not executed.

When the utility script is invoked, **form•Z** loads the utility script, calls the utility main execution function to execute the script and then the script is unloaded. The script can call **form•Z** API functions (including interface) in the main execution function to perform its task. While a utility is executing no other tasks can take place in **form•Z** (except network rendering communication). Control remains within the utility until the script has completed its task. To provide the best user experience is recommended that you provide the ability to cancel the operation and provide a progress bar for time-consuming tasks.

There are two variants to the utility scripts, system and project. System utilities are not dependent on a project window. Project utilities are dependent on a project window and are expected to function on the provided project window. A script that renders all of the **form•Z** projects in a directory is an example of a system utility.

3.7.5.1 System Utility

System utilities are defined by tagging the script in its header with the `script_type` keyword and the proper identifier as follows:

```
script_type FZ_UTIL_SYST_EXTS_TYPE
```

The user invokes a system utility script by selecting the Run Utility… item from the Extensions menu. A system utility can also be invoked from another plugin or script by calling the API function `fz_syst_script_exec_util`, or `fz_syst_plugin_exec_util`. The desired utility is specified by its location and file name.

System utility call back functions.

System utility scripts are implemented by defining one callback function. This is the function that is invoked by **form•Z** when the utility script is selected by the user.

The main execution function (required)

```
long fz_util_cbak_syst_main();
```

This is the main function for a System utility. When the script is invoked, this function is called to perform the work of the script. All execution for the script is done inside this function (or local script functions called from this function). When execution flow exits this function, the script is unloaded.

```
long fz_util_cbak_syst_main( )
{
	long		err = FZRT_NOERR;

	/* Do utility work, call API functions etc. */

	return(err);
}
```

3.7.5.2 Project Utility

Project utilities are defined by tagging the script in its header with the `script_type` keyword and the proper identifier as follows:

```
script_type FZ_UTIL_PROJ_EXTS_TYPE
```

The user invokes a project utility script by selecting the Run Utility… item from the Extensions menu. A project utility can also be invoked from another plugin or script by calling the API function `fz_proj_script_exec_util`, or `fz_proj_plugin_exec_util`. The desired utility is specified by its location and file name.

Project utility call back functions.

Project utility scripts are implemented by defining one callback function. This is the function that is invoked by **form•Z** when the utility script is selected by the user.

The main execution function (required)

```
long  fz_util_cbak_proj_main(
	long   windex
	);
```

This is the main function for a project utility. When the script is invoked, this function is called to perform the work of the script. All execution for the script is done inside this function (or local script functions called from this function). When execution flow exits this function, the script is unloaded.

```
long  fz_util_cbak_proj_main(
	long   windex
	)
{
	long		err = FZRT_NOERR;

	/* Do utility work with windex, call API functions etc. */

	return(err);
}
```

3.7.6 Object types

In **form•Z**, there is a large number of object types, also called controlled objects. They are, for example, extrusions, enclosures, cubes, cones, cylinders, spheres, tori, sweeps, stairs etc. A controlled object stores its generation parameters in a data block that is maintained with the object. The parameters can be displayed in a dialog editing environment, which can be invoked form the Query dialog. The parameters of some controlled objects can also be edited graphically through the Edit Controls tool. It is possible to create custom object types in a script

Object type script type

Object type scripts are defined by tagging the script in its header with the script_type keyword and the proper identifier as follows :

```
script_type FZ_OTYP_EXTS_TYPE
```

Object type call back function set.

Object type scripts are implemented by defining a set of callback functions. Some of these functions are optional, while others are required. Each of these functions is described in the following sections. As with all other script types, the object type script must implement the fz_script_cbak_info callback function, which defines basic information about the script. This is discussed in more detail in section 3.3.

The name function (required)

```
long fz_otyp_cbak_name (
       long          windex,
       fz_objt_ptr   objt,
       fzrt_ptr      parm,
       fz_string_td  str,
       long          max_str
       );
```

The name function defines a name for the object type. This name will show up in the **form•Z** interface, whenever object types are listed. The name function must assign a string to the function's name argument. The length of the string assigned cannot exceed max_len characters. An example of a name function is shown below.

```
long fz_otyp_cbak_name(
       long                 windex,
       fz_objt_ptr          obj,
       fzrt_ptr             parm,
       mod fz_string_td     str,
       long                 max_len
       )
{
       long err = FZRT_NOERR;

       str = "Star - SDK Script Sample";

       return(err);
}
```

The objt and parm parameters may be passed as NULL. In this case a name for all objects of this type should be returned. If objt and parm are passed in, a particular object of this type exists, and the type name may be further specified based on the parameters of the object. For example, the sweep object type in **form•Z** works this way. When its name function is called with NULL, it returns "Sweep". However, if it is called with a particular object as the parameter, the returned name contains which type of sweep it is, for example, "Axial Sweep", or "Two Source Sweep". Other object types do not make such a distinction and always return the same name, such as spheres, nurbz or symbols. It is recommended that the object type name is stored in a .fzr resource file and retrieved from it, when assigned to the name argument, so that it can be localized for different languages. In the example above, this step is omitted for the purpose of simplicity.

The info function (required)

```
long fz_otyp_cbak_info (
       mod long            flags
       );
```

form•Z needs to know some basic information about the object type, for example, whether the object type is always smooth, always facetted or both. This information is defined in the flags argument. This argument should be set with the bit encoded flags defined in the enum `fz_otyp_flags_enum`. Setting a bit in the flags argument of the function enables the functionality described by the bit. Setting a bit can be done with the `FZ_SETBIT` utility function. In case of the star object, it is defined to always generate facetted model type objects and also chooses to let **form•Z** handle the reversing of the object topology. The info function for the star object type is shown below.

```
long fz_otyp_cbak_info (
       mod long            flags
       )
{
       flags = 0;

       FZ_SETBIT(flags,FZ_OTYP_ALWAYS_FACET);
       FZ_SETBIT(flags,FZ_OTYP_HANDLE_RVRS);

       return(FZRT_NOERR);
}
```

A complete description of all object type flags follows:

`FZ_OTYP_NON_UNI_SCALE`
Certain parametric data cannot be scaled non uniformly. For example, local coordinate system with its own x, y and z axes would be distorted and even skewed with a non uniform scaling. In such a case, this bit should not be set. If a non uniform scale is applied to the object, the control parameters are automatically dropped by **form•Z**. Other parametric data can be scaled non uniformly. This is the case, for example, with nurbZ curves, which are defined by a set of control points. Scaling the control points also scales the evaluated shape of the curve. In this case, the bit should be set. The object can then be scaled non uniformly without loosing the parameters data.

`FZ_OTYP_NO_RENDER`
When this bit is set, the object will not be rendered in high end rendering modes, such as RenderZone. They will only be rendered in the interactive rendering modes. If the bit is not set, the object will always be rendered. This flag is expected to be used less frequently. It may be applied to object types, which are temporary in nature.

`FZ_OTYP_NO_SYS_FLIP`

When this bit is set, the object cannot be transformed so that a coordinate system changes from left hand to right hand without dropping the object to a plain object. Such a transformation occurs, for example, when mirroring about a plane or when scaling with one of the scale factors being negative and the other ones positive. If this bit is not set, such transformations are allowed and the object controls are not dropped.

`FZ_OTYP_ALWAYS_SMOOTH`
When this bit is set, the object is always a smooth object. In other words, its model type is always smooth. It is not possible to have both, `FZ_OTYP_ALWAYS_SMOOTH` and `FZ_OTYP_ALWAYS_FACET` set. However if none are set, the object may be smooth or facetted.

`FZ_OTYP_ALWAYS_FACET`
When this bit is set, the object is always a facetted object. In other words, it never has a smooth object representation. It is not possible to have both, `FZ_OTYP_ALWAYS_SMOOTH` and `FZ_OTYP_ALWAYS_FACET` set. However if none are set, the object may be smooth or facetted.

`FZ_OTYP_HANDLE_RVRS`
When this bit is set, the parametric representation of the object cannot be reversed in direction. In this case, **form•Z** will reverse the object facets after a reverse operation occurred. If this bit is not set, it is the responsibility of the object type to reverse its parametric data. This is usually done in the fz_otyp_cbak_rvrs callback function.

`FZ_OTYP_EXPL_PER_PART`
When this bit is set, the explode operation may yield multiple volumes for this object. When this bit is not set, the object is always represented by only one volume. In the Convert Options dialog, the Per Part check box will be added if this bit is set.

`FZ_OTYP_NESTED_CURVE_CNTRL`
When this bit is set, the object type is assumed to define an open or closed curve, which lends itself as the source for a number of other derivative objects, such as sweep, helix or revolved objects.

The parameter count function (recommended)

```
long fz_otyp_cbak_parm_count(
       mod long     count
       );
```

The parameter count function tells **form•Z**, through how many parameters the object type is defined. This number may not only include the parameters exposed to the user in the dialog interface, but also hidden parameters that may be necessary to store additional information.

```
long fz_otyp_cbak_parm_count(
       mod long     count
       )
{
       count = 6;
       return(FZRT_NOERR);
}
```

The parameter info function (recommended)

```
long fz_otyp_cbak_parm_get_info2 (
       long                              parm_indx,
       mod fzrt_UUID_td                  parm_uuid,
       mod fz_string_td                  parm_name,
       mod fz_type_enum                  parm_type,
```

```
      mod fz_fuim_format_int_enum       parm_format_int,
      mod fz_fuim_format_float_enum     parm_format_float,
      mod fz_fuim_item_type_enum        parm_fuim_item,
      mod long                          parm_range,
      fz_type_ptr                       parm_range_min,
      fz_type_ptr                       parm_range_max,
      mod long                          flags
      );
```

The parameter info function returns a number of informational values about a particular parameter. **form•Z** may invoke this function, for example, to automatically save a parameter's value to file. **form•Z** typically calls this function by looping over the number of parameters returned by the parameter count function (`fz_otyp_cbak_parm_count`). The only input argument to the info function is `parm_indx`. This is the nth parameter of the object relative to the parameter count. All other function arguments are output arguments. Each parameter needs to have a unique id. This id is returned by the `parm_uuid` argument. The name of the parameter, as it appears in a dialog is returned by `parm_name`. The data type of the parameter is defined by `parm_type`. The interface format for integer and floating point parameters are returned by `parm_format_int` and `parm_format_float`. The choice of dialog interface control by which the parameter is shown in a dialog is defined by `parm_fuim_item`. Whether or not the parameter value has lower and upper range limits is returned by `parm_range`. The min and max ranges are set in `parm_range_min` and `parm_range_max`. The flags argument defines additional attributes of the parameter. They are bit encoded. The allowable bits for the flags argument are :

`FZ_OTYP_PARM_NO_ANIM_BIT`
When this bit is set, **form•Z** cannot animate the parameter.

`FZ_OTYP_PARM_READ_ONLY_BIT`
When this bit is set, the parameter cannot be changed through the `fz_otyp_cbak_parm_set` function.

`FZ_OTYP_PARM_ANIM_LEVEL1_BIT`
When this bit is set, the parameter is considered a good parameter for animation. The parameter usually represents a fluid state. That is, a small change in the parameter causes a small change in the object. This makes it meaningful for animation. It is therefore added to the object's track list, by default, when keyframing the object. An example for such a parameter would be the radius of a sphere.

`FZ_OTYP_PARM_ANIM_LEVEL2_BIT`
When this bit is set, the parameter is considered a secondary parameter for animation. Usually, the parameter represents a state, that is not fluid. That is, a change in the parameter causes the object to take on a significantly different shape. While such a parameter can be animated, it is not added to the object's track list, by default, when keyframing the object. An example for such a parameter would be the type of a spherical object (tetrahedron, hexahedron, octahedron ...).

`FZ_OTYP_PARM_HIDDEN_BIT`
When this bit is set, the parameter is considered hidden, when the dialog interface is build. This may be the case, for example, when a parameter is used for storage of data only, but not for modification by the user.

The parameter info function for the star object type is shown below.

```
long fz_otyp_cbak_parm_get_info2 (
      long                              parm_indx,
      mod fzrt_UUID_td                  parm_uuid,
```

```
        mod fz_string_td                  parm_name,
        mod fz_type_enum                  parm_type,
        mod fz_fuim_format_int_enum       parm_format_int,
        mod fz_fuim_format_float_enum     parm_format_float,
        mod fz_otype_fuim_item_enum       parm_fuim_item,
        mod long                          parm_range,
        fz_type_ptr                       parm_range_min,
        fz_type_ptr                       parm_range_max,
        mod long                          flags
        )
{
        long   lval;
        double fval;

        switch ( parm_indx )
        {
                case 0 :
                        parm_uuid                 = STAR_PARM_TYPE_ID;
                        parm_name                 = "Base Type";
                        parm_type                 = FZ_TYPE_LONG;
                        parm_format_int           = FZ_FUIM_FORMAT_INT_DEFAULT;
                        parm_fuim_item            = FZ_FUIM_ITEM_MENU;
                        lval = 0;
                        fz_type_set_long(lval,parm_range_min);
                        lval = 7;
                        fz_type_set_long(lval,parm_range_max);
                        flags = 0;
                break;

                case 1 :
                        parm_uuid                 = STAR_PARM_RADIUS_ID;
                        parm_name                 = "Radius";
                        parm_type                 = FZ_TYPE_DOUBLE;
                        parm_format_float         = FZ_FUIM_FORMAT_FLOAT_DISTANCE;
                        parm_fuim_item            = FZ_FUIM_ITEM_TEXT;
                        parm_range                = FZ_FUIM_RANGE_MIN;
                        fval = 0.0;
                        fz_type_set_double(fval,parm_range_min);
                        flags = 0;
                break;

                case 2 :
                        parm_uuid                 = STAR_PARM_RATIO_ID;
                        parm_name                 = "Ray Ratio";
                        parm_type                 = FZ_TYPE_DOUBLE;
                        parm_format_float         = FZ_FUIM_FORMAT_FLOAT_PERCENT;
                        parm_fuim_item            = FZ_FUIM_ITEM_SLIDER_TEXT;
                        parm_range                = FZ_FUIM_RANGE_MIN |
                                                  FZ_FUIM_RANGE_MIN_INCL |
                                                  FZ_FUIM_RANGE_MAX |
                                                  FZ_FUIM_RANGE_MAX_INCL;
                        fval = 0.0;
                        fz_type_set_double(fval,parm_range_min);
                        fval = 1.0;
                        fz_type_set_double(fval,parm_range_max);
                        flags = 0;
                break;

                case 3:
                        parm_uuid                 = STAR_PARM_ORG_ID;
                        parm_name                 = "Origin";
                        parm_type                 = FZ_TYPE_XYZ;
                        flags = 0;
                        FZ_SETBIT(flags,FZ_OTYP_PARM_HIDDEN_BIT);
                break;
```

```
		case 4:
			parm_uuid			= STAR_PARM_XAXIS_ID;
			parm_name			= "X Axis";
			parm_type			= FZ_TYPE_XYZ;
			flags = 0;
			FZ_SETBIT(flags,FZ_OTYP_PARM_HIDDEN_BIT);
		break;

		case 5:
			parm_uuid			= STAR_PARM_YAXIS_ID;
			parm_name			= "Y Axis";
			parm_type			= FZ_TYPE_XYZ;
			flags = 0;
			FZ_SETBIT(flags,FZ_OTYP_PARM_HIDDEN_BIT);
		break;

	}

	return(FZRT_NOERR);
}
```

The parameter get state name function (recommended)

```
long fz_otyp_cbak_parm_get_state_str (
	fzrt_UUID_td		parm_uuid,
	long			indx,
	mod fz_string_td	str
	);
```

This function should be implemented, if an integer or boolean parameter is displayed as a menu item in a dialog. Given the parameter's uuid, this function returns the nth string associated with the nth state of that parameter. This function may also be used if the parameter is shown through a set of radio buttons. The get state name function is mainly used when **form•Z** automatically builds a dialog interface and by the animation track editor interface.

```
long fz_otyp_cbak_parm_get_state_str (
	fzrt_UUID_td		parm_uuid,
	long			indx,
	mod fz_string_td	str
	)
{
	if ( parm_uuid == STAR_PARM_TYPE_ID )
	{
		switch ( indx )
		{
			case 0 : str = "Tetrahedron";		break;
			case 1 : str = "Hexahedron";		break;
			case 2 : str = "Octahedron";		break;
			case 3 : str = "Dodecahedron";	break;
			case 4 : str = "Icosahedron";		break;
			case 5 : str = "Soccer Ball";		break;
			case 6 : str = "Geodesic Level 1";	break;
			case 7 : str = "Geodesic Level 2";	break;
		}
	}

	return(FZRT_NOERR);
}
```

The init function (recommended)

```
long fz_otyp_cbak_init (
       long                       windex,
       fz_objt_ptr                obj,
       fzrt_ptr                   parm
       );
```

form•Z calls this function to initialize the parameters of the object with default values. The storage for the parameters has already been allocated by **form•Z** and is passed in to this function as the parm parameter. The object to which the parameters belong and the project in which the object resides are passed in as well. The init function for the star object type is shown below.

```
long fz_otyp_cbak_init (
       long                windex,
       fz_objt_ptr         obj,
       fzrt_ptr            parm
       )
{
       long                lval;
       double              dval;
       fz_xyz_td           xyz;

       lval = 0;           fz_objt_edit_parm_set(windex,obj,STAR_PARM_TYPE_ID,lval);
       dval = 12.0;        fz_objt_edit_parm_set(windex,obj,STAR_PARM_RADIUS_ID,dval);
       dval = 0.5;         fz_objt_edit_parm_set(windex,obj,STAR_PARM_RATIO_ID,dval);

       xyz = {0,0,0};      fz_objt_edit_parm_set(windex,obj,STAR_PARM_ORG_ID,xyz);
       xyz = {1,0,0};      fz_objt_edit_parm_set(windex,obj,STAR_PARM_XAXIS_ID,xyz);
       xyz = {0,1,0};      fz_objt_edit_parm_set(windex,obj,STAR_PARM_YAXIS_ID,xyz);

       return(FZRT_NOERR);
}
```

The regeneration function (required)

```
long fz_otyp_cbak_regen (
       long                       windex,
       fz_objt_ptr                obj,
       fzrt_ptr                   parm
       );
```

The regeneration function is called when **form•Z** needs to recreate the shape of the object based on the current settings of the object's parameters. This may be necessary, for example, after the display resolution attribute of the object was edited, or a parameter of the object was altered through the edit dialog, invoked from the Query dialog. This function constitutes the real essence of the object type, as it defines the steps necessary to create the final form of the object, executed by calling various **form•Z** API functions. There are a number of ways to create the object's shape. One would be to construct one face at a time, using the API `fz_objt_fact_create_face`. This process is illustrated in the regenerate function of the star object type shown below.

```
long fz_otyp_cbak_regen (
       long                windex,
       fz_objt_ptr         obj,
       fzrt_ptr            parm
       )
{
       long          rv = FZRT_NOERR;
       fz_xyz_td     rxyz,rot,pnt[],vec,star_origin,xaxis,yaxis;
       double        radius,star_radius,star_rad_ratio;
       long          i,n,ncord,nsegt,ncurv,nface,ncord2,nsegt2,ncurv2,nface2;
```

```
long            sindx,shead,snext,pindx[3],lval;
fz_map_plane_td                     local_mplane;
fz_objt_ptr                         temp_obj;
fz_objt_spid_type_enum              spid_type;
fz_objt_spid_cnstr_opts_ptr         spid_opts;
fzrt_boolean                        bval;
long                                star_base_type;

if(parm != NULL)
{

        star_otyp_get_mplane(windex,obj,local_mplane);
        fz_objt_fact_reset(windex, obj);

        fz_objt_edit_parm_get(windex,obj,STAR_PARM_TYPE_ID,star_base_type);
        fz_objt_edit_parm_get(windex,obj,STAR_PARM_RADIUS_ID,star_radius);
        fz_objt_edit_parm_get(windex,obj,STAR_PARM_RATIO_ID,star_rad_ratio);
        fz_objt_edit_parm_get(windex,obj,STAR_PARM_ORG_ID,star_origin);
        fz_objt_edit_parm_get(windex,obj,STAR_PARM_XAXIS_ID,xaxis);
        fz_objt_edit_parm_get(windex,obj,STAR_PARM_YAXIS_ID,yaxis);

        fz_math_3d_vec_rotation_xyz(xaxis,yaxis,rot);

        radius = star_radius * (START_RATIO_MIN +
                star_rad_ratio * (START_RATIO_MAX - START_RATIO_MIN));
        rxyz.x = radius;
        rxyz.y = radius;
        rxyz.z = radius;
        spid_opts = NULL;

        switch ( star_base_type )
        {
                case 0 : spid_type = FZ_OBJT_SPID_TYPE_TETRA;         break;
                case 1 : spid_type = FZ_OBJT_SPID_TYPE_HEXA;          break;
                case 2 : spid_type = FZ_OBJT_SPID_TYPE_OCTA;          break;
                case 3 : spid_type = FZ_OBJT_SPID_TYPE_DODECA;        break;
                case 4 : spid_type = FZ_OBJT_SPID_TYPE_ICOSA;         break;
                case 5 : spid_type = FZ_OBJT_SPID_TYPE_SOCCER;        break;
                case 6 :
                case 7 :
                        spid_type = FZ_OBJT_SPID_TYPE_GEO;
                        fz_objt_cnstr_spid_opts_init(windex,spid_opts);
                        if ( star_base_type == 6 ) lval = 2;
                        else                       lval = 4;


                        fz_objt_cnstr_spid_opts_set(windex,spid_opts,
                                FZ_OBJT_SPID_PARM_GEO_NUM_SUBDIV,lval);
                        bval = TRUE;
                        fz_objt_cnstr_spid_opts_set(windex,spid_opts,
                                FZ_OBJT_SPID_PARM_GEO_BY_LEVEL,bval);
                break;

        }

        if((rv = fz_objt_cnstr_spid(windex,rxyz,spid_type,
                        star_origin,rot,spid_opts,temp_obj)) == FZRT_NOERR)
        {
                fz_objt_get_face_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,nface);
                fz_objt_get_curv_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,ncurv);
                fz_objt_get_segt_count(windex,temp_obj,
                        FZ_OBJT_MODEL_TYPE_FACT,nsegt);
                fz_objt_get_point_count(windex,temp_obj,
```

```
        FZ_OBJT_MODEL_TYPE_FACT,ncord);

ncord2 = ncord + nface;
ncurv2 = 0;
nface2 = 0;
nsegt2 = 0;
for(i = 0; i < ncurv; i++)
{
        fz_objt_curv_get_segt_count(windex,temp_obj,i,
                FZ_OBJT_MODEL_TYPE_FACT,n);
        ncurv2 +=  n;
        nface2 +=  n;
        nsegt2 +=  n * 3;
}

if((rv = fz_objt_fact_allocate(windex,obj,
                nface2,ncurv2,nsegt2,ncord2)) == FZRT_NOERR )
{
        /* COPY SPHEROID POINTS */
        for(i = 0; i < ncord; i++)
        {       fz_objt_point_get_xyz(windex,temp_obj,i,
                        FZ_OBJT_MODEL_TYPE_FACT,pnt[i]);
        }
        fz_objt_fact_add_pnts(windex,obj,pnt,ncord);

        /* CREATE STAR TIP POINTS */
        radius = star_radius - radius;
        for(i = 0; i < nface; i++)
        {
                fz_objt_alys_get_face_cog(windex,temp_obj,i,
                        FZ_OBJT_MODEL_TYPE_FACT,pnt[i]);
                fz_math_3d_create_unit_vec(star_origin,pnt[i],vec);
                pnt[i] += vec * radius;
        }
        fz_objt_fact_add_pnts(windex,obj,pnt,nface);

        /* CREATE FACES */
        for(i = 0; i < ncurv; i++)
        {
                fz_objt_curv_get_segt_count(windex,temp_obj,i,
                        FZ_OBJT_MODEL_TYPE_FACT,n);

                fz_objt_curv_get_sindx(windex,temp_obj,i,
                        FZ_OBJT_MODEL_TYPE_FACT,shead);
                sindx = shead;
                do
                {
                        fz_objt_segt_get_next(windex,temp_obj,sindx,
                                FZ_OBJT_MODEL_TYPE_FACT,snext);

                        fz_objt_segt_get_start_pindx(windex,temp_obj,
                                sindx,FZ_OBJT_MODEL_TYPE_FACT,pindx[0]);
                        fz_objt_segt_get_end_pindx(windex,temp_obj,
                                sindx,FZ_OBJT_MODEL_TYPE_FACT,pindx[1]);
                        pindx[2] = ncord + i;

                        fz_objt_fact_create_face(windex,obj,
                                pindx,3,NULL);

                } while ((sindx = snext) != shead );
        }

        /* LINK FACES */
        fz_objt_fact_link_faces(windex,obj);
}
```

```
                    if (spid_opts) fz_objt_cnstr_spid_opts_finit(windex,spid_opts);

                    fz_objt_edit_delete_objt(windex, temp_obj);
            }
    }

    return(rv);
}
```

Another method to create the object's shape would be to use a sequence of higher level API construction functions. These will create temporary objects, which can be combined using editing API function to yield the final object. The temporary objects used along the way need to be deleted and the content of the final object copied into the object passed into the regeneration function. For example, the star object could be constructed by creating a number of pyramids (the star's rays), transforming them to attach to the faces of a spheroid object and then using the boolean union tool to join the all together into the final shape. The intermediate objects all need to be deleted. In this case, the direct creation process clearly is the better approach.

The finit function (optional)

```
long fz_otyp_cbak_finit (
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm
        );
```

form•Z calls the finit function whenever an object of the given type is deleted. The function is expected to take whatever action is necessary, when an object of this type ceases to exist. Note that it is not necessary to delete the basic storage for the object's parameters, which is passed in this function as the parm argument. In case of the star object, the finit function is not necessary as no special steps are necessary when the object is deleted.

The transform function (optional)

```
long fz_otyp_cbak_tform (
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm,
        fz_mat4x4_td        tform
        );
```

form•Z calls the transform function whenever an object is transformed (moved, rotated, scaled and/or mirrored). When an object contains positional geometric properties, such as an origin or 3d points they need to be transformed as well. Points can be transformed with the math API function `fz_math_4x4_multiply_mat_xyz`. If an object contains a linear dimension, such as a radius, only the scale factor of the matrix need to be applied. This scale factor can be extracted with the math API `fz_math_4x4_mat_to_trl_scl_rot`. The transform function for the star object type is listed below.

```
long fz_otyp_cbak_tform (
        long                windex,
        fz_objt_ptr         objt,
        fzrt_ptr            parm,
        fz_mat4x4_td        tform
        )
{

        fz_xyz_td           org,xaxis,yaxis,xaxis_pt,yaxis_pt,scl;
```

```
        double              radius;

        fz_math_4x4_mat_to_trl_scl_rot(tform,NULL,scl,NULL);

        fz_objt_edit_parm_get(windex,objt,STAR_PARM_ORG_ID,org);
        fz_objt_edit_parm_get(windex,objt,STAR_PARM_XAXIS_ID,xaxis);
        fz_objt_edit_parm_get(windex,objt,STAR_PARM_YAXIS_ID,yaxis);
        fz_objt_edit_parm_get(windex,objt,STAR_PARM_RADIUS_ID,radius);

        xaxis_pt = xaxis + org;
        yaxis_pt = yaxis + org;

        fz_math_4x4_multiply_mat_xyz(tform, org);
        fz_math_4x4_multiply_mat_xyz(tform, xaxis_pt);
        fz_math_4x4_multiply_mat_xyz(tform, yaxis_pt);


        xaxis  = xaxis_pt - org;
        yaxis  = yaxis_pt - org;
        radius *= scl.x;

        fz_objt_edit_parm_set(windex,objt,STAR_PARM_RADIUS_ID,radius);
        fz_objt_edit_parm_set(windex,objt,STAR_PARM_ORG_ID,org);
        fz_objt_edit_parm_set(windex,objt,STAR_PARM_XAXIS_ID,xaxis);
        fz_objt_edit_parm_set(windex,objt,STAR_PARM_YAXIS_ID,yaxis);


        return(FZRT_NOERR);
}
```

The geometry function (optional)

```
long fz_otyp_cbak_geom(
        long                        windex,
        fz_objt_ptr                 obj,
        fzrt_ptr                    parm,
        mod     fz_map_plane_td     plane,
        mod     fz_xyz_td           center,
        mod     fz_xyz_mm_td        bbox
        );
```

form•Z calls the geometry function to retrieve basic geometric information about the object. It should be implemented if the object has its own, local coordinate system. For example, a sphere has its own x, y and z axis, which describe the location and orientation of the sphere in 3d space. The plane parameter returns the origin and rotation of the object's coordinate system in world space. This information is used, for example, to draw the object axes in wireframe. The center parameter returns the object's origin in the coordinate space of the object. Usually the center would be set to {0.0, 0.0, 0.0}, but may have different values, depending on the nature of the object. The bbox parameter returns the extent of the object along its x, y and z axis. If this function is not implemented by the plugin, the information is calculated from the facetted data of the object. For example, the center is computed as the average of all coordinate points of the object. The geometry function for the star object type is shown below.

```
long fz_otyp_cbak_geom(
        long                        windex,
        fz_objt_ptr                 obj,
        fzrt_ptr                    parm,
        mod     fz_map_plane_td     plane,
        mod     fz_xyz_td           center,
        mod     fz_xyz_mm_td        bbox
        )
{
```

```
	double		radius;
	fz_xyz_td	org,xaxis,yaxis,xaxis_pt,yaxis_pt;
	long		err = FZRT_NOERR;

	if(parm != NULL)
	{
		fz_objt_edit_parm_get(windex,obj,STAR_PARM_ORG_ID,org);
		fz_objt_edit_parm_get(windex,obj,STAR_PARM_XAXIS_ID,xaxis);
		fz_objt_edit_parm_get(windex,obj,STAR_PARM_YAXIS_ID,yaxis);

		xaxis_pt = xaxis + org;
		yaxis_pt = yaxis + org;

		fz_math_3d_map_plane_from_pts(xaxis_pt, org, yaxis_pt, plane);

		center.x = 0.0;
		center.y = 0.0;
		center.z = 0.0;

		fz_objt_edit_parm_get(windex,obj,STAR_PARM_RADIUS_ID,radius);
		bbox.xmin = bbox.ymin = bbox.zmin = -radius;
		bbox.xmax = bbox.ymax = bbox.zmax =  radius;
	}

	return(err);
}
```

The cvsl function (optional)

```
fzrt_error_td fz_otyp_cbak_cvsl (
	long			windex,
	fz_objt_ptr		obj,
	fzrt_ptr		parm,
	mod fz_xyz_td		cog,
	mod double		volume,
	mod double		surf_area,
	mod double		length,
	mod long		result
	);
```

The cvsl function is called by **form•Z** to retrieve the center of gravity, volume, surface area and length (abbreviated cvsl) of an object. This function should be implemented, when the object type can provide more accurate values, than those computed from the facetted or smooth topology and geometry of the object. Since not all of these properties can be calculated for an object, the result parameter returned to **form•Z** tells which properties were computed by the function, by setting certain bits to on.
bit 0: center of gravity was calculated
bit 1: volume was calculated
bit 2: surface area was calculated
bit 3: perimeter length was calculated
For example, the perimeter length can only be calculated for curve like objects but not for solids. Therefore, for solids, bit #3 should not be set.

The key points function (optional)

```
long fz_otyp_cbak_get_key_pnts(
	long				windex,
	fz_objt_ptr			obj,
	fzrt_ptr			parm,
```

```
	mod long			knt,
	long				pnt_indx,
	mod fz_xyz_td			pnt
	);
```

form•Z calls the key points function to get important points from the object, which may not be part of the object's actual geometry. For example, the key points of an arc are its center, its start and end point. This function is called in two modes. If `pnt_indx` is passed as `-1` the function only needs to determine how many key points there are and pass that value back in the knt parameter. If `pnt_indx` is passed as 0 or greater, it identifies which key point to set in the `pnt` function argument. The key points function for the star object type is shown below.

```
long fz_otyp_cbak_get_key_pnts(
	long				windex,
	fz_objt_ptr			obj,
	fzrt_ptr			parm,
	mod long			knt,
	long				pnt_indx,
	mod fz_xyz_td			pnt
	)
{
	long				err = FZRT_NOERR;
	fz_map_plane_td			local_mplane;

	knt = 1;

	if( pnt_indx == 0 )
	{
		star_otyp_get_mplane(windex,obj,local_mplane);

		pnt.x = 0.0;
		pnt.y = 0.0;
		pnt.z = 0.0;

		fz_math_3d_map_plane_to_world(local_mplane, pnt);
	}

	return(err);
}
```

3.8 Developing and debugging scripts

3.8.1 Editing Scripts

form•Z Script environment

The Script Editor is an environment that allows you to edit and compile **form•Z** Scripts. The Script environment is available whenever a Script Window is selected, or by clicking on the **New Script** menu item or the **Open** item, both in the **File** menu. This environment is useful for a quick edit-compile-test cycle without the need to leave the **form•Z** Application to edit the script file. The Script Editor also exists as a standalone application. That is, it only offers the script editing environment. No modeling or drafting windows can be opened in the Script Editor application. With the exception of the utility scripts, it is necessary to restart **form•Z** to see the changes made to a script after it was compiled. With the Script Editor as a standalone application, it is possible to run the Editor application all the time to edit and compile the script, while starting and quitting **form•Z** to test the script.

The Script window is a basic text editor. Typing text into this window will enter the text at the current insertion point indicated by a flashing vertical bar. The insert point can be changed by clicking once in the contents of the script window. The insert point is changed to the spot that is clicked. The insertion point may also be changed by using the navigational arrow keys. Pressing the left or right arrow keys will move the insertion point one character left or right respectively. Pressing the up or down arrow keys will move the insertion point up or down one line respectively.

You may select text by clicking and dragging the mouse over the text you wish to select, or you can double-click to select a word or triple-click to select a line of text. Typing text with a range of text currently selected will replace the selected text with the newly typed text.

All of the text in the script window is styled with the same font and font size, which can be set in the Script Preferences described below. It is recommended that a fixed width font be used so that all lines of text can be lined up to indicated nesting levels in the script language.

The menu bar of the Script Editor has 6 menus: **File**, **Edit**, **Window**, **Search**, **Script**, and **Help**. The following sections will describe the menu items that are available in the standalone application. The menus for the Script Editor environment in the **form•Z** Application will contain additional menu items, that do not directly affect Script Editor windows. These menu items are described in the **form•Z** Application User's Manual.

3.8.1.1 The File Menu

New Script

The New Script item opens a new **form•Z** Script window. The Script Window is used to create and edit your **form•Z** script. The new window becomes the active window and appears above all other windows. The contents in a new Script window are not stored on disk until it is saved to a file using Save, or Save As... discussed below.

Open...

This item will open an existing file. If the file was already open, it will bring the previously opened window to the top of all other windows. The file may be a **form•Z** Script or any other **form•Z** native file format. Other **form•Z** native file formats include **form•Z** Project files, **form•Z** Libraries and **form•Z** Imager Sets. These are discussed elsewhere in the manual.

Close

This command will close the active script window. If changes have been made since the last save command, an alert will be displayed prompting you to save change or close without saving. After the window is closed the previously active window becomes the active window.

Save

This command will save the currently active Script window. The Script windows is saved as a text file with the extension .fsl. On Macintosh it will additionally have the Finder type of 'TEXT'. If the window has not been saved before, this command will display the Save As... dialog. The default name of a file the first time it is saved is ScriptN.fsl where N is a decimal number indicating which New Script window was created.

Save As...

This command will save the contents of the currently active Script window to a new file. This command is usually executed the first time a window is saved or your want the contents saved to a different file. It will display a standard Save As dialog. By default the file is saved in the users Documents folder. After the file has been saved to a new file, the title of the script window now reflects the new filename, and all subsequent Save commands will save the file to the new file.

Save A Copy As...

This command will save the contents of the currently active Script window to a new file. It resembles the Save As... command in that it displays the standard Save As dialog, but it does not change the title of the window, and all subsequent Save commands will still save to the original file, if it has been saved previously. If the file has not been saved previously, then the title remains ScriptN and the first Save command will invoke the Save As... dialog.

Revert To Saved

This command will change the contents of the window to the state it was at the time of the last save. It will display an alert that requests confirmation before the command is completed. The Undo and Redo states are reset. This command cannot be undone.

Page Setup...

This command will display the system's standard printer Page Setup dialog. This dialog contains settings for the current printer's page attributes, such as orientation, scale, and page size.

Print...

This item will print the contents of the current Script window. It will display the standard Print dialog. From this dialog you may print all pages or a selected page range. Lines of text are not wrapped at the page edge. Lines of text that are too long to fit on the page will be truncated. Setting the Paper Size, scale, or orientation will affect how much of the line is printed to the age. The text is printed using the current font family and font size selected in the Script Preferences.

Quit/Exit

This command will terminate the application. All open Script windows are closed. If there are any unsaved changes in a window, an alert is displayed asking to save or discard changes before closing the window, or to cancel the quit command. After all windows have been closed, the application terminates.

3.8.1.2 The Edit Menu

Undo
Redo

These commands will revert or reapply changes made since the file was opened with the Open... command or created with the New Script command. The script environment supports multiple undo and redo commands and undo across **Save...** command.

The Undo command will revert the contents of the window to the state prior to the last action. Actions that can be undone are typing text, Cut and Paste Commands, Shift Right and Shift Left commands, and Replace commands.

When the Undo command is selected after typing text, all text that was typed since the last non-typing action, changing the position of the insertion point or selecting text is removed. If the prior state included selected text, the selected text is recovered and re-selected.

The Redo command will reapply the previous Undo action. It is available after the Undo command has been executed and before another action is taken. After another action has been taken after executing the Undo command, the previous Undo commands cannot be redone.

Cut

This command will delete the currently selected text and place it on the system clipboard. The text will replace any previous text that was placed in the system clipboard with the Cut or Copy commands. This includes text that was placed on the system clipboard by another application. Once the text has been placed on the system clipboard, it can be pasted into another location in the script window, into a different script window, or another application that supports pasting text from the system clipboard. If no text is selected, this command is disabled.

Copy

This command will place the currently selected text on the system clipboard. The text will replace any previous text that was placed in the system clipboard with the Cut or Copy commands. This includes text that was placed on the system clipboard by another application. Once the text has been placed on the system clipboard, it can be pasted into another location in the script window, into a different script window, or another application that supports pasting text from the system clipboard. If no text is selected, this command is disabled.

Paste

This command will Paste the text from the system clipboard into the current script window. Pasting text from the system clipboard will insert the text from the system clipboard at the current insertion point or selection range. If text is selected the selected text is deleted and replaced with the text from the clipboard. This command is disabled if there is no text currently on the system clipboard.

Balance

This command will find balanced pairs of enclosure characters. This command is useful for finding the corresponding open or close character to a function or if-then block.

Enclosure characters are defined to be parentheses '(' and ')', braces '{' and '}', and brackets '[' and ']'. The balanced pairs and all text between the balanced pairs are selected. A balanced pair of enclosure characters are defined to be an opening and closing character of the same type with zero or more balanced matching pairs between the them.

The Balance command will select the smallest balanced pair from the current insertion point or the beginning of a selection range. If no balanced pair can be found, the system will beep and leave the current selection or insertion point unchanged.

This is useful for finding mismatched braces, or for indenting blocks of code to improve readability.

Shift Right
Shift Left

These commands will indent or outdent a selection of lines. This command is useful for indenting a block of text inside a function or if statement to denote the level of block enclosures.

The Shift Right command will selected the current line or lines of text and insert a tab character at the beginning of each line. If the current selection range is not an entire line the selection range is extended to the beginning of the line that contains the selection range start and the end of the line that contain the end of the selection range.

The Shift Left command command will remove the beginning tab character or spaces at the beginning of each line. As with the Shift Right command the selection range is extended to include the entire lines. If there is no tab character at the beginning of the line, but there are spaces at the beginning of the line, the spaces will be removed. The number of spaces removed is defined by the current Tab Spaces setting in the Script Preferences dialog. If there are no tab characters or spaces at the beginning of the line, the line is left unchanged.

Key Shortcuts

This command will display the Key Shortcuts dialog. You can assign short cut keys for all commands in the Script Editor from this dialog. To add and change a key shortcut, select the command from the list on the left. The current key shortcut for the command is shown in the

Shortcut window on the right, with a brief description of the command above that. To add a new key shortcut, click the Add button, to edit an existing key shortcut click the Edit button. The corresponding Add Shortcut or Edit Short dialog will be displayed. The key shortcuts for the Script Editor environment in the **form•Z** Application may not conflict with key shortcuts set for the **form•Z** Project environment.

For a more detailed discussion on editing key shortcuts see section 3.2.5 in the **form•Z** User's Manual.

Preferences

Script Preferences

The script preference category applies to settings when a Script window is the currently active window.

Output Directory:

This defines where the compiled script file is located. The Compiled script file has the same base filename as the source script file but has a .fsb extension.

With Source File:

The compiled script file is placed in the same folder as the source file.

Custom:

Defines a specific folder that will contain all compiled script files.

The specified custom folder is displayed below the Custom radio button. By default, the current folder is defined as the custom directory for compile script files.

To change the folder click the Choose button. This will display a standard Choose folder dialog to locate the folder to use as the custom output directory, and click OK.

Tab Spaces:

This defines how many spaces a tab character will take up in the script window. This also affects how many spaces are removed from the beginning of the line during the Shift Left command.

Font Size:

This defines the size of the font to use in the script window.

Font:

This defines the font family to use in the script window. All text displayed in the script window or printed is drawn with this font.

Syntax Coloring:

This will enable the coloring of keywords, function names, comments and constants in the script window. This is useful for quickly determining what is a reserved keyword or function name, and also for finding code that has been inadvertently commented out.

The text in the editor window are colored as follows:

Keywords	-	blue
Function names	-	cyan
Constants	-	gold
Comments	-	red
Quoted strings	-	light grey
Everything else	-	black

3.8.1.3 The Window Menu

Close

This command will close the currently active window. This is the same selecting the Close command from the File menu in the script environment.

Close All

This items works as in the main **form•Z** Menu

Windows

Following the Close All command on the menu is a list of all currently opened windows. To change the currently active window, select it from the list. This will cause the selected window to move to the top of all other windows, and make it the currently active window. If the window selected is not a script window, the script environment will be replace with the environment for the selected window.

3.8.1.4 The Search Menu

The items in this menu control the search and replace functionality of the script environment.

Find...

This item will display the Find... dialog. From this dialog you can set the search string, and the replacement string. To set the search string, click in the edit field next to Find and type the string to search for. To set the replacement string, click in the edit field next to Replace and type the replacement string.

Match whole word:

This will enable whole word matching. When match whole word is enabled, Find..., Find Next, and Find Previous will find the search string only if the found string contains non-alpha and non-numeric characters before and after it.

Case Sensitive:

This will enable case sensitive searching. When case sensitive is enabled, Find..., Find Next, and Find Previous will find the search string only if the found string matches exactly in upper and lower case with the search string.

Wrap At End of File:

This will cause the Find action to continue the search from the top of the file if a match is not found between the current insertion point and the end of the file. The Find Previous command will continue it search backward from the end of the file if a match is not found between the current point and the top of the file.

Click OK in the Find... dialog to close the dialog and find the next occurrence of the search tring from the current insertion point. If the search string is found, the found string is selected. If the search string is not found, the system will beep.

Find Next

This item will find the next occurrence of the search string from the current insertion point or the end of the current selected text. It is enabled only if the search string has been set with either Find... command or the Set Search String command.

Find Previous

This item will find the previous occurrence of the search string from the current insertion point or the beginning of the current selected text. It is enabled only if the search string has been set with either Find... command or the Set Search String command.

Set Search String

This item will set the current search string with the currently selected text, but no search is executed. The match whole word and case sensitive settings are left unchanged. This item is only available when text is selected.

Replace

This command will replace the currently selected text with the replacement string only if the selected text matches the search string.

Replace and Find Next

This command will replace the currently selected text with the replacement string only if the selected text matches the search string and then find the next occurrence of the search string.

Replace All

This command will search for the search string and replace it with the replacement string over the entire file. The current match word and case sensitive settings are used in the search and replace operation.

3.8.1.5 The Script Menu

Check Syntax

This command will check the syntax of the current script file.

Compile

This command will compile the current script file and if there are no errors it will generate the compiled script file in the directory specified by the Script Preferences Output Directory setting.

The Help Menu

form•Z Web Site...

This command will launch the operating system's default web browser and opens the Home Page of the auto•des•sys, Inc. web site (http://www.formz.com), where a variety of information about our products as well as support material can be found.

form•Z Web Support...

This command works as for the previous item, except that it takes ou to the Technical Support page of our web site (http://www.formz.com/support/index.html).

email Tech Support...

This command opens your operating system's default email application and sets a blank email addressed to the **form•Z** Technical Support department (support@formZ.com).

3.8.2 Creating script files

3.8.2.1 New Script...

New Script...

Choosing this command will bring up the **New Script** dialog, shown in Figure 3.8.2.1.1, which presents many options for creating new scripts.

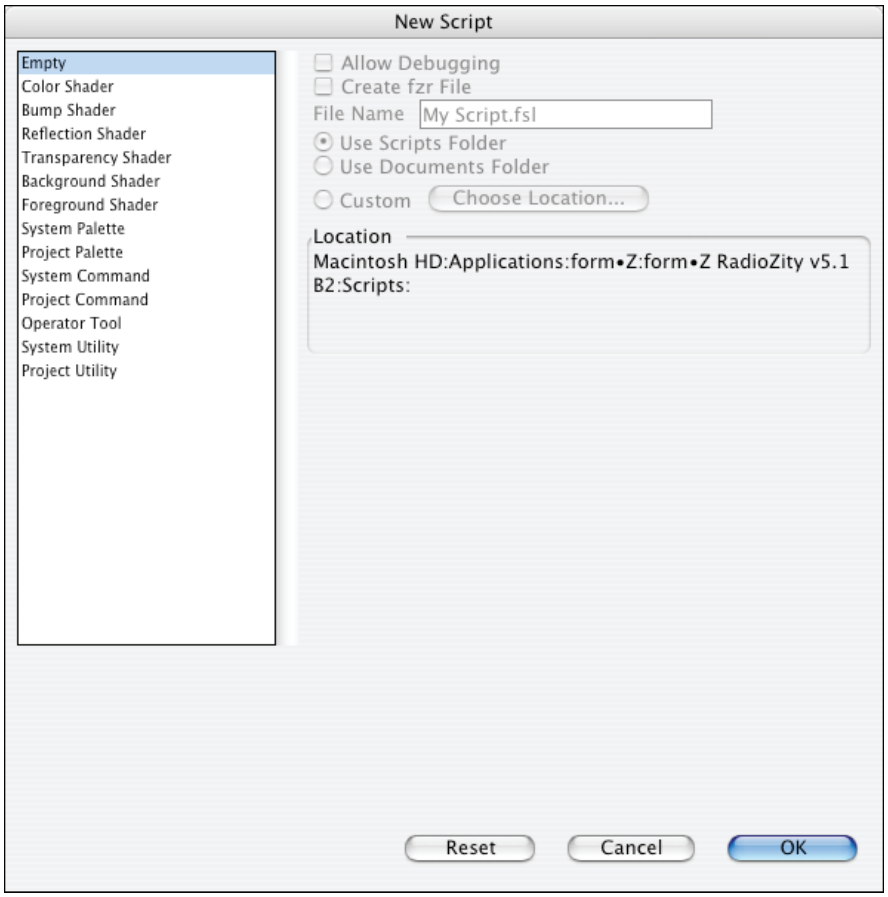


Figure 3.8.2.1.1: The **New Script** dialog.

With the **New Script...** command, a blank text edit window can be opened in which the script commands can be typed, edited and compiled. In order to facilitate the creation of a script extension in form•Z, the **New Script...** command also offers the option to automatically create the source code that is the same for each type of script. This allows the user to focus on writing the actual functionality of the script rather than needing to retype the required structure of the script. In other words, this option automatically creates the source code for a given script type, based on some options that the user selects. The script is complete, containing all the required functions and compiles without errors. The user simply has to fill in the code which constitutes the specific functionality of the script. Places in the script where the user is expected to fill in his/her own code are explicitly marked.

3.8.2.2 Common Script Options

There are several options that are common to all scripts, except empty scripts.

Allow Debugging: This checkbox will add a line to the header of the script that indicates that debugging of the script is allowed. If this checkbox isn't checked, debugging is not set up in the script.

Create fzr File: This checkbox indicates whether an fzr file should be created alongside the fsl file. An fzr file is a text file that contains all the string resources for a script. If this option is checked, any strings in the script that can be stored in the fzr file will be. This option is highly recommended for any script that may be run by users in other languages as an fzr file helps to facilitate localization of scripts. In the header section of the fsl file will be defines for each of the strings stored in the fzr file.
The fzr file that is created will be named using the base name of the fsl file with a language indicator and an fzr extension. For example, if the script was named "My Script.fsl" and the computer language is set to English, the fzr file would be named "My Script.ENU.fzr". The fzr file name can be changed after creation so long as the name is also changed where used in the fsl file. If this option is selected, the fzr file is opened for editing along with the fsl file.
The basic layout of an fzr file is shown below.

```
FZRF, 40, CHAR=MAC,
STR#, 1,
"My Operator Tool",
"My Operator Tool Help String",
FZND,
```

An fzr file uses a comma-separated value (CSV) file format, meaning that every entry in the file is separated by a comma. New lines between each string are not required, but make reading easier. The first line is the header. The *FZRF* indicates this is an fzr file. The 40 is a required entry. The *CHAR=MAC* indicates the character set of the fzr file, in this case Mac OS. If the fzr file is created on a Windows machine, it will read *CHAR=WIN*. The next line *STR#*,*1,* indicates that all the following entries are strings. This section is terminated by the entry *FZND*. The *1* in the line is the index for this group of strings. There can be multiple groups of strings to group like items together, but the **New Script** command will only create one group.

```
#define TOOL_STR_ID 1
#define TOOL_STR_NAME 1
#define TOOL_STR_HELP 2
```

Within the fsl file, defines are added to the top of the file for accessing the strings in the fzr file. The first define indicates the group the string belongs to, in this case it will always be 1.

```
#define TOOL_STR_ID 1
```

Following this define is a series of defines that act an an index to the correct line in the group. Strings are indexed in the the fzr file starting from 1.
In order to retrieve a string in an fzr file, the following is used:

```
fzrt_fzr_get_string(_tool_rsrc_ref, TOOL_STR_ID, TOOL_STR_NAME, name);
```

This function will get the correct string out of the fzr file. These are set up automatically in the fsl file. The *_tool_rsrc_ref* is a global variable the points to the fzr file. It is set up with a call to *fzrt_fzr_open*, which is placed in the proper place in the fsl file by the **New Script** command. The next two entries are the string group and string entry respectively. The final entry is a variable to hold the string.

File Name: This edit field indicates the file name of the fsl file. An fsl file will always have the extension ".fsl". That means that entering a name without an extension, or an extension other than fsl, will cause form•Z to add the extension to the name.

Following the **File Name** option are three mutually exclusive options used to indicate where the fsl file is to be saved.
Use Scripts Folder: This option will save the fsl file in the default scripts folder, which is a folder named "Scripts" in the same folder as the application.
Use Documents Folder: This option will save the fsl file to the documents folder of your computer.
Custom: This option allows the user to specify a location other than the two options above. Clicking the **Choose Location...** button will bring up a dialog where the user can choose the location for the fsl file.
It is recommended to use the Scripts folder because by default form•Z searches this folder for scripts at startup. The path where scripts are currently set to be saved to is displayed below the **Custom** radio button in the **Location** group.

In addition to the settings above, there are some items in scripts that are set to default values. All default values can be edited after the fsl file has been created or left as they are. There are certain default values that are set to values that are recommended to be edited after the file has been created, though these values can be left as-is. If the default value is a string, it will be set to **"***"**. The help string and vendor name are two such examples. If fzr files are used, be sure to check it for any default strings.

In addition to setting up default code, there will also be comments throughout the code to indicate areas that should be edited. Comments in the code are bracketed by "/* */". If the comment begins with "<<<<", it indicates an area where code generally should be added or modified. All functions have a header comment area where at least a description of the function and whether the function is optional or required is stated.

Clicking the **OK** button will create the script and open the fsl file for all scripts except for empty scripts. Empty scripts will not create a file on the disk but will instead open an empty editable text window. All the other script types will have the fsl file created in the location specified and opened into a text window. If the **Create fzr File** option is selected, the fzr file will be created in the same location as the fsl file and opened into a text window. Once the fsl file is open, it can be edited, saved, or compiled. If the generated fsl file is compiled, prior to editing, it will compile without errors. Note that when an fzr file is created and opened it is not a compilable file type and will give errors if compilation is attempted.

3.8.2.3 Empty Scripts

The simplest option available is to create an empty script. An empty script isn't a type of script but is instead a way to create an empty editable text window. There are no additional options available when creating an empty script. As opposed to all the other script types, this script type does not create a file on the disk, but instead opens an empty editable text window.

3.8.2.4 RenderZone Shader Scripts

When a pixel in an image is rendered, the shaders needed to compute the final pixel color are executed in a specific order. This order is referred to as the shader pipeline. The sequence is: color, bump, reflection, transparency, background, foreground. Each of the shader type scripts are described below.

Color Shader Scripts
This is the first step of the shader pipeline. The color shader of the material assigned to the surface on which the pixel lies is executed. This defines the unshaded pixel color. The options to create a color shader script are shown in Figure 3.8.2.4.1.

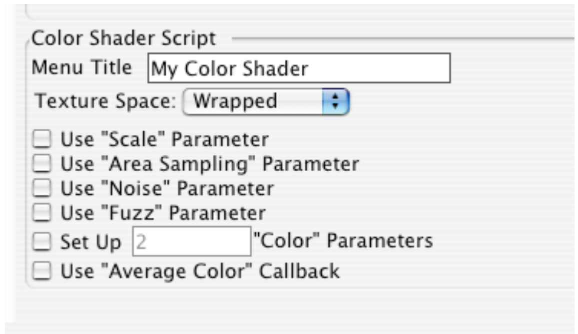


Figure: 3.8.2.4.1: The **Color Shader Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Color** menu of the **Surface Style Parameters** dialog.

Texture Space: This option indicates the type of texture space.
If **wrapped** is chosen, then the texture space is 2D, and the line *fz_shdr_set_wrapped(TRUE);* is added to the *fz_shdr_cbak_colr_set_parameters* callback function.
If **solid** is chosen, then the texture space is 3D, and the line *fz_shdr_set_solid(TRUE);* is added to the *fz_shdr_cbak_colr_set_parameters* callback function.
If **unknown** is chosen, neither 2D nor 3D is set up, and no lines are added to the *fz_shdr_cbak_colr_set_parameters* callback function.
Use "Scale" Parameter: This option is mainly useful for shaders that create a pattern. Setting this option will add the **Scale** field in the shader options dialog. By default, the scale is set to 100%. This option adds the line *fz_shdr_set_scale_parm(1.0);* to the *fz_shdr_cbak_colr_set_parameters* callback function.
Use "Area Sampling" Parameter: This option is mainly used for shaders that have patterns. Selecting this option will turn on area sampling, which is helpful in reducing moire artifacts in the pattern. Setting this option will add the **Area Sampling** check box in the shader options dialog. This option adds the line *fz_shdr_set_area_sample_parm(TRUE);* to the *fz_shdr_cbak_colr_set_parameters* callback function.
Use "Noise" Parameter: This option is used to add the standard noise parameters to a shader. Setting this option will add the **Noise** menu and **# of Impulses** field to the shader option dialog. The default type of noise is "better" with number of impulses set to 3. This option adds the line *fz_shdr_set_noise_parm(FZ_SHDR_TURB_TYPE_BETTER, 3);* to the *fz_shdr_cbak_colr_set_parameters* callback function. This option also adds the lines *fz_shdr_get_noise_type(_ntype);* and *fz_shdr_get_noise_impulses(_nimpulse);* to the *fz_shdr_cbak_colr_pre_render* callback function, which store the impulse and noise type in global variables for use in the *fz_shdr_cbak_colr_pixel* callback function.
Use "Fuzz" Parameter: This option is used to add fuzz to a shader. Setting this option will add the **Fuzz** slider and edit field to the shader option dialog. This option adds the line *fz_shdr_set_sld_flt_parm("fuzz", 0.0, 1, 1, PARAM_ID_FUZZ);* to the *fz_shdr_cbak_colr_set_parameters* callback function.
Set Up x "Color" Parameters: This option is used to add a number of colors to the shader. Setting this option will add a number of color selections in the shader options dialog. The default for each color is white. This option adds the lines

colr = {1.0, 1.0, 1.0};
fz_shdr_set_col_parm("color 1", colr, PARAM_ID_COLR_1); for each of the number of color parameters chosen to the *fz_shdr_cbak_colr_set_parameters* callback function.

Use "Average Color" Callback: This option will create the average color callback function in the script. The default action for this callback is to average all of the colors set with the **Set Up x "Color" Parameters** option. If this option is not selected, form•Z will substitute a 50% gray.

Bump Shader Scripts
This is the second step of the shader pipeline. The bump shader of the material assigned to the surface on which the pixel lies is executed. This defines a new normal direction at the pixel, which is important for the reflection calculation that comes next. The options for creating a bump shader script are shown in Figure 3.8.2.4.2.

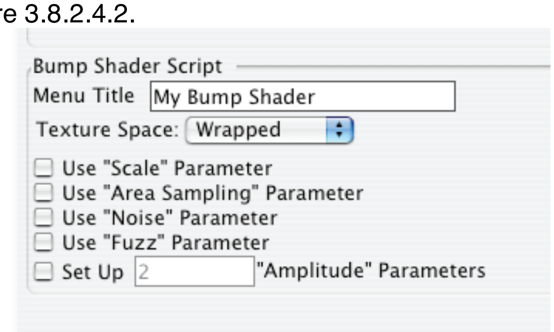


***Figure 3.8.2.4.2*:** The **Bump Shader Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Bump** menu of the **Surface Style Parameters** dialog.

Texture Space: This option indicates the type of texture space.
If **wrapped** is chosen, then the texture space is 2D, and the line *fz_shdr_set_wrapped(TRUE);* is added to the *fz_shdr_cbak_bump_set_parameters* callback function.
If **solid** is chosen, then the texture space is 3D, and the line *fz_shdr_set_solid(TRUE);* is added to the *fz_shdr_cbak_bump_set_parameters* callback function.
If **unknown** is chosen, neither 2D nor 3D is set up, and no lines are added to the *fz_shdr_cbak_bump_set_parameters* callback function.
Use "Scale" Parameter: This option is mainly useful for shaders that create a pattern. Setting this option will add the **Scale** field in the shader options dialog. By default, the scale is set to 100%. This option adds the line *fz_shdr_set_scale_parm(1.0);* to the *fz_shdr_cbak_bump_set_parameters* callback function.
Use "Area Sampling" Parameter: This option is mainly used for shaders that have patterns. Selecting this option will turn on area sampling, which is helpful in reducing moire artifacts in the pattern. Setting this option will add the **Area Sampling** check box in the shader options dialog. This option adds the line *fz_shdr_set_area_sample_parm(TRUE);* to the *fz_shdr_cbak_bump_set_parameters* callback function.
Use "Noise" Parameter: This option is used to add the standard noise parameters to a shader. Setting this option will add the **Noise** menu and **# of Impulses** field to the shader option dialog. The default type of noise is "better" with the number of impulses set to 3. This option adds the line *fz_shdr_set_noise_parm(FZ_SHDR_TURB_TYPE_BETTER, 3);* to the *fz_shdr_cbak_bump_set_parameters* callback function. This option also adds the lines *fz_shdr_get_noise_type(_ntype);* and *fz_shdr_get_noise_impulses(_nimpulse);* to the *fz_shdr_cbak_bump_pre_render* callback function, which store the impulse and noise type in

global variables for use in the *fz_shdr_cbak_bump_pixel* callback function.

Use "Fuzz" Parameter: This option is used to add fuzz to a shader. Setting this option will add the **Fuzz** slider and edit field to the shader option dialog. This option adds the line *fz_shdr_set_sld_flt_parm("fuzz", 0.0, 1, 1, PARAM_ID_FUZZ);* to the *fz_shdr_cbak_bump_set_parameters* callback function.

Set Up x "Amplitude" Parameters: This option is used to add a number of amplitudes to the shader. Setting this option will add a number of amplitude selections in the shader options dialog. The default amplitude is 10% with an inclusive range of 0 - 100%. This option adds the line *fz_shdr_set_sld_flt_parm("amplitude 1", 0.0, 1, 1, PARAM_ID_AMPL_1);* for each of the number of amplitude parameters chosen to the *fz_shdr_cbak_bump_set_parameters* callback function.

Reflection Shader Scripts

This is the third step of the shader pipeline. The reflection shader of the material assigned to the surface on which the pixel lies is executed. The unshaded pixel color, generated by the color shader is augmented with shading information from all lights in the scene. If a bump shader other than None was used, the altered surface normal direction will be used to create bump patterns from the shading calculation. The shaded color is returned by the reflection shader. The options for creating a reflection shader script are shown in Figure 3.8.2.4.3.

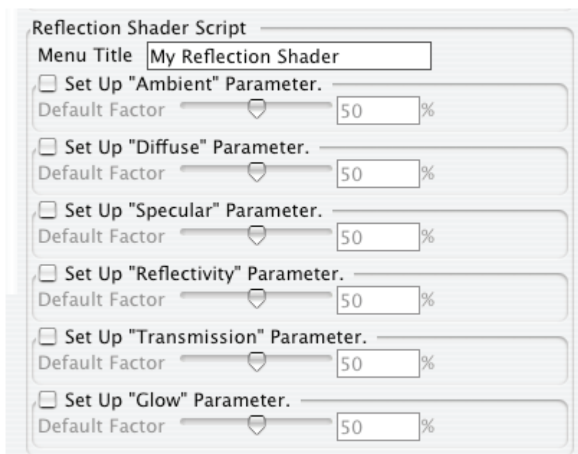


Figure 3.8.2.4.3: The **Reflection Shader Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Reflection** menu of the **Surface Style Parameters** dialog.

Set Up "Ambient" Parameter: This option is used to set the standard ambient reflection parameter. Setting this option will add the **Ambient Reflection** group in the shader options dialog. This option adds the line *fz_shdr_set_ambient_parm(x);* , where x is the value entered in the edit field, to the *fz_shdr_cbak_refl_set_parameters* callback function.

Set Up "Diffuse" Parameter: This option is used to set the standard diffuse reflection parameter. Setting this option will add the **Diffuse Reflection** group in the shader options dialog. This option adds the line *fz_shdr_set_diffuse_parm(x);* , where x is the value entered in the edit field, to the *fz_shdr_cbak_refl_set_parameters* callback function.

Set Up "Specular" Parameter: This option is used to set the standard specular reflection parameter. Setting this option will add the **Specular Reflection** group in the shader options dialog. This option adds the line *fz_shdr_set_specular_parm(x, 0.01);* , where x is the value entered in the edit field, to the *fz_shdr_cbak_refl_set_parameters* callback function.

Set Up "Reflectivity" Parameter: This option is used to set the standard reflectivity reflection parameter. Setting this option will add the **Reflectivity Reflection** group in the shader options dialog. This option adds the line *fz_shdr_set_mirror_parm(x); ,* where x is the value entered in the edit field, to the *fz_shdr_cbak_refl_set_parameters* callback function.

Set Up "Transmission" Parameter: This option is used to set the standard transmission reflection parameter. Setting this option will add the **Transmission Reflection** group in the shader options dialog. This option adds the line *fz_shdr_set_transmission_parm(x, 1.0); ,* where x is the value entered in the edit field, to the *fz_shdr_cbak_refl_set_parameters* callback function.

Set Up "Glow" Parameter: This option is used to set the standard glow reflection parameter. Setting this option will add the **Glow Reflection** group in the shader options dialog. This option adds the line *fz_shdr_set_glow_parm(x); ,* where x is the value entered in the edit field, to the *fz_shdr_cbak_refl_set_parameters* callback function.

Transparency Shader Scripts
This is the fourth step of the shader pipeline. The transparency shader of the material assigned to the surface on which the pixel lies is executed. The transparency of the pixel is returned by the shader and retained by form•Z. The options for creating a transparency shader are shown in Figure 3.8.2.4.4.

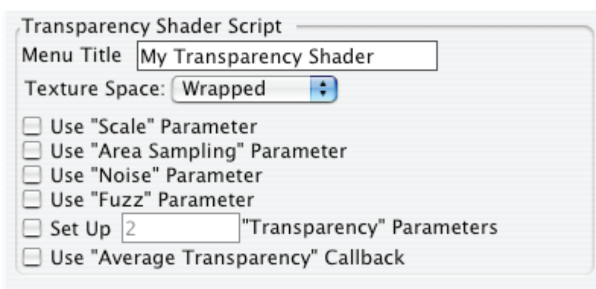


***Figure 3.8.2.4.4*: The **Transparency Shader Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Transparency** menu of the **Surface Style Parameters** dialog.

Texture Space: This option indicates the type of texture space.
If **wrapped** is chosen, then the texture space is 2D, and the line *fz_shdr_set_wrapped(TRUE);* is added to the *fz_shdr_cbak_trns_set_parameters* callback function.
If **solid** is chosen, then the texture space is 3D, and the line *fz_shdr_set_solid(TRUE);* is added to the *fz_shdr_cbak_trns_set_parameters* callback function.
If **unknown** is chosen, neither 2D nor 3D is set up, and no lines are added to the *fz_shdr_cbak_trns_set_parameters* callback function.

Use "Scale" Parameter: This option is mainly useful for shaders that create a pattern. Setting this option will add the **Scale** field in the shader options dialog. By default, the scale is set to 100%. This option adds the line *fz_shdr_set_scale_parm(1.0);* to the *fz_shdr_cbak_trns_set_parameters* callback function.

Use "Area Sampling" Parameter: This option is mainly used for shaders that have patterns. Selecting this option will turn on area sampling, which is helpful in reducing moire artifacts in the pattern. Setting this option will add the **Area Sampling** check box in the shader options dialog. This option adds the line *fz_shdr_set_area_sample_parm(TRUE);* to the *fz_shdr_cbak_trns_set_parameters* callback function.

Use "Noise" Parameter: This option is used to add the standard noise parameters to a shader. Setting this option will add the **Noise** menu and **# of Impulses** field to the shader option dialog. The default type of noise is "better" with number of impulses set to 3. This option adds the line *fz_shdr_set_noise_parm(FZ_SHDR_TURB_TYPE_BETTER, 3);* to the *fz_shdr_cbak_trns_set_parameters* callback function. This option also adds the lines *fz_shdr_get_noise_type(_ntype);* and *fz_shdr_get_noise_impulses(_nimpulse);* to the *fz_shdr_cbak_trns_pre_render* callback function, which store the impulse and noise type in global variables for use in the *fz_shdr_cbak_trns_pixel* callback function.

Use "Fuzz" Parameter: This option is used to add fuzz to a shader. Setting this option will add the **Fuzz** slider and edit field to the shader option dialog. This option adds the line *fz_shdr_set_sld_flt_parm("fuzz", 0.0, 1, 1, PARAM_ID_FUZZ);* to the *fz_shdr_cbak_trns_set_parameters* callback function.

Set Up x "Transparency" Parameters: This option is used to add a number of transparencies to the shader. Setting this option will add a number of transparency selections in the shader options dialog. The default transparency for each option is 100% with an inclusive range of 0 - 100%. This option adds the line *fz_shdr_set_sld_flt_parm("transparency 1", 1.0, 1, 1, PARAM_ID_TRNS_1);* for each of the number of transparency parameters chosen to the *fz_shdr_cbak_trns_set_parameters* callback function.

Use "Average Transparency" Callback: This option will create the average color callback function in the script. The default action for this callback is to average all of the transparencies set with the **Set Up x "Transparency" Parameters** option. If this option is not selected, the average transparency for a shader is set to 0% (fully opaque).

Background Shader Scripts

This is the fifth step of the shader pipeline. If the transparency value from step 4 is more than 0.0 (i.e. there is some level of transparency) the background shader is executed. The color from the background shader and the shaded color from step 3 are mixed using the transparency value and returned by the shader. The options for creating a background shader are shown in Figure 3.8.2.4.5.

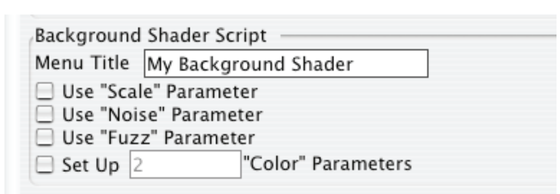


Figure 3.8.2.4.5**:** The **Background Shader Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Background** menu of the **RenderZone Options** dialog in the **Scene** tab and **Background** group.

Use "Scale" Parameter: This option is mainly useful for shaders that create a pattern. Setting this option will add the **Scale** field in the shader options dialog. By default, the scale is set to

100%. This option adds the line *fz_shdr_set_scale_parm(1.0);* to the *fz_shdr_cbak_bgnd_set_parameters* callback function.

Use "Noise" Parameter: This option is used to add the standard noise parameters to a shader. Setting this option will add the **Noise** menu and **# of Impulses** field to the shader option dialog. The default type of noise is "better" with number of impulses set to 3. This option adds the line *fz_shdr_set_noise_parm(FZ_SHDR_TURB_TYPE_BETTER, 3);* to the *fz_shdr_cbak_bgnd_set_parameters* callback function. This option also adds the lines *fz_shdr_get_noise_type(_ntype);* and *fz_shdr_get_noise_impulses(_nimpulse);* to the *fz_shdr_cbak_bgnd_pre_render* callback function, which store the impulse and noise type in global variables for use in the *fz_shdr_cbak_bgnd_pixel* callback function.

Use "Fuzz" Parameter: This option is used to add fuzz to a shader. Setting this option will add the **Fuzz** slider and edit field to the shader option dialog. This option adds the line *fz_shdr_set_sld_flt_parm("fuzz", 0.0, 1, 1, PARAM_ID_FUZZ);* to the *fz_shdr_cbak_bgnd_set_parameters* callback function.

Use "Average Color" Callback: This option will create the average color callback function is the script. The default action for this callback is to average all of the colors set with the **Set Up x "Color" Parameters** option. If this option is not selected, form•Z will substitute a 50% gray. This option adds the lines
colr = {1.0, 1.0, 1.0};
fz_shdr_set_col_parm("color 1", colr, PARAM_ID_COLR_1);
to the *fz_shdr_cbak_bgnd_set_parameters* callback function.

Foreground Shader Scripts
This is the sixth and final step of the shader pipeline. The foreground shader is also known as the **Depth Effect** shader. The depth effect shader is executed. It uses the color from step 5. A new color is calculated using the depth information of the current pixel. This color is returned and becomes the final pixel color in the image. The options for creating a background shader are shown in Figure 3.8.2.4.6.

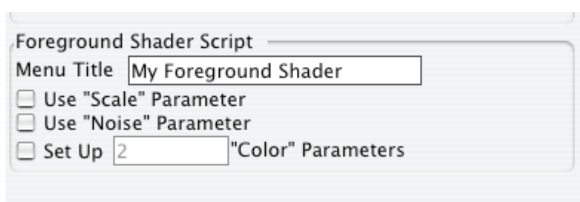


***Figure 3.8.2.4.6*: The Foreground Shader Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Shader** menu of the **RenderZone Options** dialog in the **Scene** tab and **Environment** group.

Use "Scale" Parameter: This option is mainly useful for shaders that create a pattern. Setting this option will add the **Scale** field in the shader options dialog. By default, the scale is set to 100%. This option adds the line *fz_shdr_set_scale_parm(1.0);* to the *fz_shdr_cbak_fgnd_set_parameters* callback function.

Use "Noise" Parameter: This option is used to add the standard noise parameters to a shader. Setting this option will add the **Noise** menu and **# of Impulses** field to the shader option dialog. The default type of noise is "better" with number of impulses set to 3. This option adds the line *fz_shdr_set_noise_parm(FZ_SHDR_TURB_TYPE_BETTER, 3);* to the *fz_shdr_cbak_fgnd_set_parameters* callback function. This option also adds the lines *fz_shdr_get_noise_type(_ntype);* and *fz_shdr_get_noise_impulses(_nimpulse);* to the *fz_shdr_cbak_fgnd_pre_render* callback function, which store the impulse and noise type in global variables for use in the *fz_shdr_cbak_fgnd_pixel* callback function.

3.8.2.5 Palette Scripts

A palette is a floating window that contains an interface for a feature or set of related features. The interface is composed of a variety of interface elements (buttons, radio buttons, check boxes, etc.) provided by the form•Z interface manager (fuim). Palette scripts are extensions that complement the form•Z palettes and behave consistently with the form•Z palettes.
Palettes are available in system and project levels. System palettes are global in nature and do not require a project window index while project palettes require a project or window index and are expected to operate on project information for a provided project,

The **New Script..** command will create an empty palette, with an indicator as to where to add interface code. The options for creating a system or project palette script are shown in figure 3.8.2.5.1.

System Palette Scripts

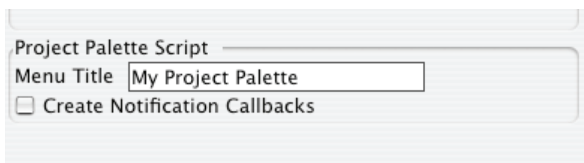


***Figure 3.8.2.5.1*:** The **Project Palette Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Palettes** menu.

Create Notification Callbacks: This checkbox indicates whether to add all the notification callbacks to the fsl file. See section 3.1.8.8 for more details.

Project Palette Scripts

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Palettes** menu.
Create Notification Callbacks: This checkbox indicates whether to add all the notification callbacks to the fsl file. See section 3.1.8.8 for more details.

3.8.2.6 Command Scripts

A command in form•Z is an action that is invoked from a menu item or a key shortcut. Command scripts are extensions that complement the form•Z commands and behave consistently with the form•Z commands. Command scripts are available in system and project levels. A system command is global in nature and does not require a project window index. These are typically utility actions for which it is desirable to have access to the utility in the form•Z interface. A project command requires a project or window index and are expected to operate on the project information for a provided project. Project commands are unavailable when there is no open project window. The options for creating a system or project command script are shown in Figure 3.8.2.6.1.

System Command Scripts

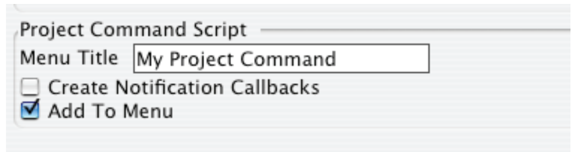


Figure 3.8.2.6.1**:** The **Project Command Script** options in the **New Script** dialog.

Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Extensions** menu.
Create Notification Callbacks: This checkbox indicates whether to add all the notification callbacks to the fsl file. See section 3.1.8.8 for more details.
Add To Menu: This option indicates whether the command should appear in the extensions menu or not. This option is selected by default.

Project Command Scripts
Menu Title: This option indicates the name of the script. The name entered here will be used for the name of the script and will appear in the **Extensions** menu.
Create Notification Callbacks: This checkbox indicates whether to add all the notification callbacks to the fsl file. See section 3.1.8.8 for more details.
Add To Menu: This option indicates whether the command should appear in the extensions menu or not. This option is selected by default.

3.8.2.7 Tool Scripts
Tool scripts are extensions that complement the form•Z tool set and behave consistently with the form•Z tools. They appear in the form•Z interface in the icon tool palettes just like a form•Z tool. Tools can either be operators or modifiers. An operator creates or edits the form•Z project data (objects, lights, etc.) through graphic manipulation in the form•Z project window. A modifier is a tool that controls a setting that affects a group of operators. The **New Script...** command only handles operator type tools. This is because modifier type tools are less frequently used and require more user decision than a simple dialog can present. Please see the SDK documentation for further information on modifier tools. The options for creating an operator tool script are shown in Figure 3.8.2.7.1.

Operator Tool Scripts

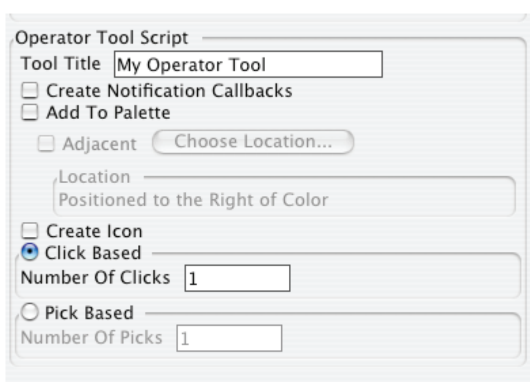


Figure 3.8.2.7.1: The **Bump Shader Script** options in the **New Script** dialog.

Tool Title: This option indicates the name of the script. The name entered here will be used for the name of the script and the tool title.

Create Notification Callbacks: This checkbox indicates whether to add all the notification callbacks to the fsl file. See section 3.1.8.8 for more details.

Add To Palette: This option indicates whether the tool should appear in the tool palette or not. If this option is not selected, the tool will only be accessible via a key shortcut (which is not set up by default).

Adjacent: This option indicates how the tool's icon will be placed in the tool palette. If this option is enabled, the position of the icon will be determined by the location chosen in the **Tool Adjacency** dialog, brought up by clicking the **Choose Location...** button. If the option is disabled, the tool's icon will be placed in a group at the bottom of the tool palette where all other extensions with option disabled are placed.

The **Tool Adjacency** dialog (Figure 3.8.2.7.2) is opened by clicking the **Choose Location...** button next to the **Adjacent** check box. It presents a list of all the tools in the tool palette. Selecting a tool will cause the new tool to be positioned adjacent to it. The **Adjacent to which side** option indicates which of the edges of the tool selected in the list the new tool will be positioned.

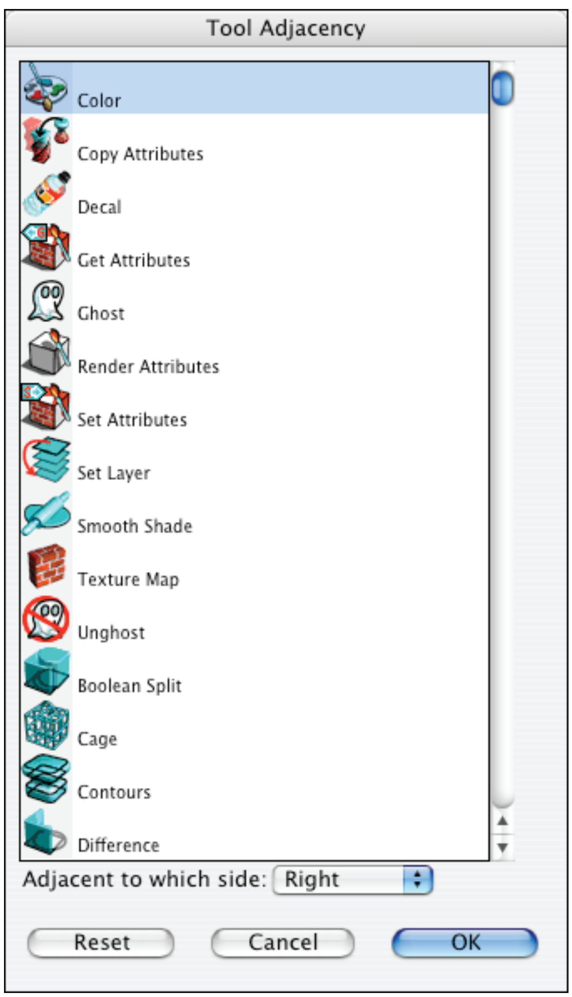


***Figure 3.8.2.7.2*:** The **Tool Adjacency** dialog.

Create Icon: This option sets up the script to load an icon file out of a TIFF file. It also creates two TIFF files that are placeholders for the script. They can be used as-is, but it is recommended that they be edited in a graphics program (like **Photoshop**) to reflect the nature of the script. The two TIFF files that are created will be named based on the base name of the fsl file with a color indicator and a TIF extension. For example, if your script was named "My Script.fsl", the black and white TIFF file would be named "My Script_bw.TIF" and the color TIFF file would be named "My Script_color.TIF". The TIFF file names can be changed after creation so long as the name is also changed where used in the fsl file. If this option is not selected and the tool is set to be in the tool palette, form•Z will assign a default icon to the tool.

The **Click Based** and **Pick Based** options are mutually exclusive and indicate how mouse clicks are handled by the tool.
Number Of Clicks: This option indicates how many clicks the tool expects to do its job.
Number Of Picks: This option indicates how many picked objects the tool expects to do its job.

A click based tool is mainly used for creating new objects. The mouse clicks should be interpreted by the script as input used to create the object. By default, clicks are handled as XYZ points, though the script can be later edited to change this. A click based tool will also add several additional callbacks.

Prompt callback: This function handles text input from the prompt window. By default, the script is set up to interpret input as XYZ points, though the script can be later edited to change this. This function should most likely work the same as the click function, except handling text input instead of mouse clicks.

Track callback: This function is used to update any interactive input as the mouse moves in the window.

Cancel callback: This function is called by form•Z when a tool is interrupted. A tool can be canceled by the user using the key cancel key shortcut or by form•Z if a form•Z operation ID executed that cancels the current operation (selecting another tool for example). This function is used to cleanup any data that was generated during the execution of the tool.
Undo callback: This function is called by form•Z when the user selects the undo menu item from the Edit menu during the execution of the tool. This function is used to back the input up to the state of the previous click.
Redo callback: This function is called by form•Z when the user selects the redo menu item from the Edit menu during the execution of the tool. This function is used to move the input up to the state of the previously undone click.

A pick based tool is mainly used for editing and derivation.
The pick based tool created will handle both pre picking and post picking. The picks are interpreted as object picks, but the script can be later edited to pick by other topological levels.

3.8.2.8 Utility Scripts

Utility scripts are designed to execute a task which is either less frequently used or an item in the form•Z interface is not desired. Utility scripts are best used on tasks that are linear in nature (like batch processing). Utility scripts are not loaded by form•Z at startup. This allows form•Z to start up faster and use less memory. Utility scripts are not listed in the Extensions Manager dialog and they do not need to be located in the Extensions Manager's search paths. There are two variants to the utility scripts, system and project. System utilities are not dependent on a project window.

Project utilities are dependent on a project window and are expected to function on the provided project window.

System Utility Scripts
System utility scripts are simple scripts that are run from the extension menu. These differ from project utility scripts in that they operate independently of any particular project window. There are no options to choose in the system utility script. All the functionality of the script takes place in the fz_util_cbak_syst_main callback function.

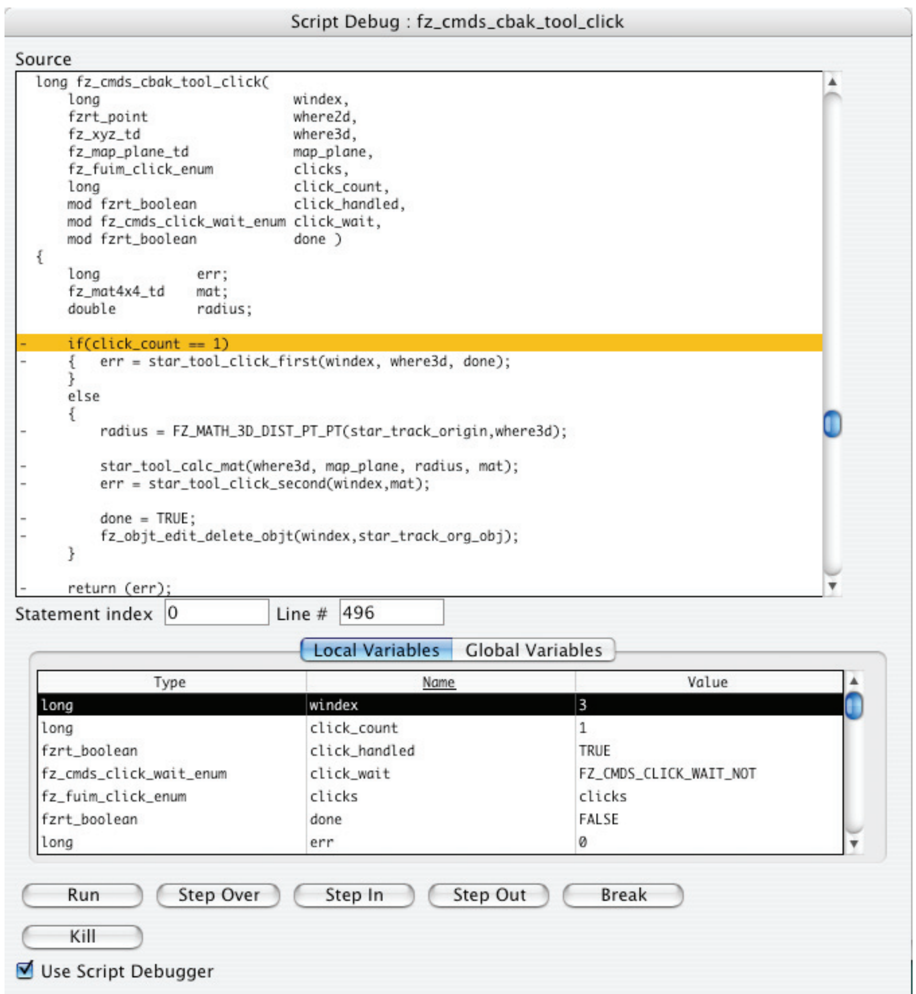
Project Utility Scripts
System utility scripts are simple scripts that are run from the extension menu. These differ from system utility scripts in that they operate on the active project window. There are no options to choose in the system utility script. All the functionality of the script takes place in the fz_util_cbak_proj_main callback function.

3.8.3 Debugging Scripts

After writing and successfully compiling a script, **form•Z** will load the script the next time **form•Z** is started and enable the functionality defined by the script. For example, when a color shader script is located in on of the directories, searched by **form•Z** at startup for scripts and plugins, the color shader is automatically added to the Color menu in the Surface Style Parameters dialog. It is often necessary to check that the statements in the script function perform the correct task. This process is called debugging. **form•Z** offers an environment, where the source code of the script is displayed in a window, and the developer can step through the script code, one statement at a time. The same environment also displays a list of all variables in a function and their current values. To invoke this debugging environment two steps need to be taken. First, the script source needs to include the header identifier:

```
script_debug TRUE
```

at the top of the script source file. This tells **form•Z**, that this particular script is meant for debugging. Second, the Use Script Debugger item in the Extensions menu needs to be selected. This enables the debugging mode in **form•Z**. As soon as the **form•Z** script debugging mode is enabled and a function in a script which is labels with the `script_debug` identifier is about to be executed, the Script Debug dialog is invoked, as shown below.



Executing a single statement

At this point, the execution of the script is suspended at the first statement of the function, which is highlighted. The developer may execute the highlighted statement by pressing the Step Over button at the bottom of the dialog. After the statement's execution, the next statement is highlighted. Pressing the Step Over button again will go to the next statement in the function, etc. When the last statement in the function is reached, usually a `return` statement, and executed, the Script Debug dialog is closed and control is returned to **form•Z**.

Setting break points

At the left border of the display of each statement a dash is shown. When clicking on it, it is changed to a star. This symbolizes a break point. When pressing the Run button at the bottom of the dialog, all statements up to the breakpoint are executed without stopping. This allows the developer to quickly move to a specific location in the source code, without having to press the Step Over button repeatedly. Clicking on the star will return to a dash and the breakpoint is removed. If Run is pressed and there are no break points set, the entire script will execute without stopping again, and the dialog will disappear.

Stepping into a function

If the current statement is a function call, and the Step Over button is pressed, the function and all its statements are executed. If the Step In button is pressed and the function is a script function (as opposed to a **form•Z** API function), the next statement highlighted will be the first statement in that function. The display window will scroll, so that this statement will appear in the middle of the window. The statements in the stepped in function can be executed in the same manner by pressing the Step Over button. When the last statement is reached and executed, the display jumps back out to the place where the Step In button was pressed and the next statement after the function call is highlighted.

Stepping out of a function

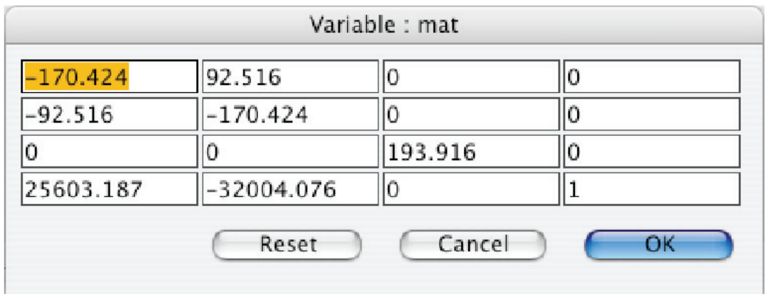
After pressing the Step In button to step through the statements of a function call, the developer may exit the function in one step by pressing the Step Out button. This will execute all the remaining statements in the function and stop at the next statement after the function call. This is equivalent to pressing the Step Over button until the last statement in the stepped in function is reached. Note, that if Run would be pressed inside a stepped in function, all the remaining steps in the script, including the ones outside the stepped in function would be executed. If no break points would be set, the script would continue to execute and the Script Debug dialog would be closed.

Variable display

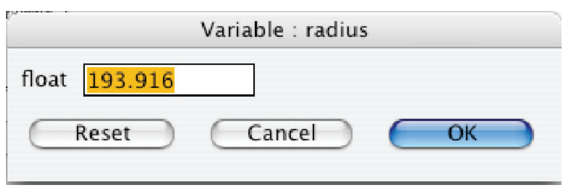
Below the source window is a list window which displays all the variables in the current function and all the global variables. Recall, that a global variable is defined in a script outside of a function's body and can be accessed by all functions in a script. To display the function or global variables, the respective tab at the top of the list needs to be selected. The variable list consists of three columns. The left most column shows the variable's type. The center column contains the variable's name and the right column displays the current value of the variable. As the developer steps through the script's statements, the values of the variables will be updated. For example, if a statement increases the value of an integer variable by 10, the value display will reflect that change after the statement was executed with the Step Over button.

Changing the value of a variable

The value of some variable types cannot be displayed directly in the list window. For example, a matrix contains too many individual members (16 for a 4 by 4 matrix). Array variables also don't show their content in the list window directly. When double clicking on an entry in the variable list window, a dialog is invoked which displays the content of that variable and also allows the developer to edit the variable's value. Depending on what kind of variable it is, the dialog takes on a different layout. For example, for an integer variable, the dialog contains a single text edit field. For a 4 by 4 matrix the 16 members are displayed in 16 text edit fields laid out in a 4 by 4 grid.

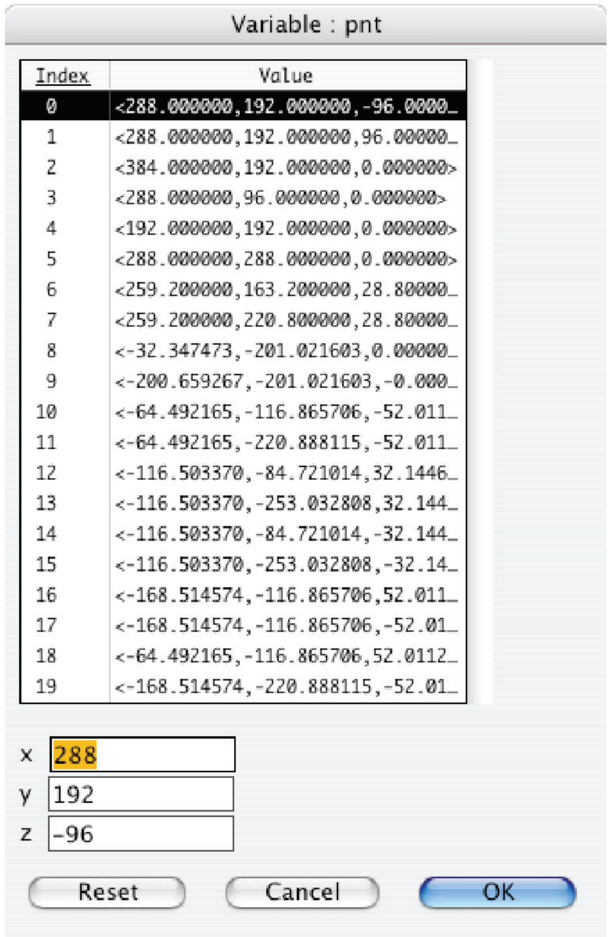


The variable editor dialog for a 4x4 matrix variable



The variable editor dialog for a floating point variable

For array variables, the dialog consists of a scrolled list which contains each array member and an edit section below, which displays the content of the currently selected array member in the list above.



The variable editor dialog for an array of xyz points

If the content of a variable is changed in that dialog and OK is pressed, the variable's value is updated in the variable list of the debugging dialog. This allows the developer to alter the execution of a script by manually changing variables. This may be useful for example, to force the script to execute certain statements, which otherwise would only execute rarely occurring conditions.

Debugging scripts with multiple callback functions

When debugging a utility script, form•Z has to invoke only one callback function, the main utility function. The Script Debug dialog is invoked before the first line of this main function is executed (see above). Other script types have more than one callback function. In case of a tool script, there may be quite a few. When the script file is set to debug mode with the

```
script_debug TRUE
```

statement at the top of the source file, the Script Debug dialog is invoked each time any of the callback functions of the tool script is invoked. Quite possibly, the developer is not interested to debug all functions, but only a few, maybe even only one particular one. In this case, the continuous presence of the Script Debug dialog is quite annoying. In order to debug just one, or a few selected functions, the `script_debug` statement at the top of the source file should be changed to :

```
script_debug 2
```

Now, by default the Script Debug dialog will not be invoked for any function. In order to debug a specific function, the keyword `debug` needs to precede the return type in the function header. For example :

```
debug long fz_tool_cbak_select(long windex)
{
	…

	return(FZRT_NOERR);
}
```

Now only those functions, which are specifically tagged for debugging cause the Script Debug dialog to pop up, when the function is invoked.

To summarize :

```
script_debug TRUE
```

causes the Script Debug dialog to be invoked for all callback functions, whereas

```
script_debug 2
```

causes the Script Debug dialog to be invoked only for those callback functions, that have the `debug` keyword in their function header. In either case, the Use Script Debugger item in the Extensions menu must be selected to activate debug mode.