MuPDF Explored

Robin Watts

March 31, 2017

Preface

This is the beginnings of a book on MuPDF. It is far from complete, but offers useful information as far as it goes. We offer it with the latest release of MuPDF (currently 1.11) in the hopes it will be useful.

Any feedback, corrections or suggestions are welcome. Please visit `bugs.ghostscript.com` and open a bug with "MuPDF" as the product, and "Documentation" as the component.

We will endeavour to make new versions available from the Documentation section of mupdf.com as they appear.

Contents

Chapter 1

Introduction

1.1 What is MuPDF?

MuPDF is a portable C library for opening, manipulating and rendering documents in a variety of formats, including PDF, XPS, SVG, e-pub, and many common image formats.

This core C library provides an API (known as the MuPDF API) that allows a wide range of actions to be performed on those documents. The exact actions available depend on the format of the document, but always includes rendering of those files.

As well as this library, the MuPDF distribution includes various tools built on top of this API. These tools include simple viewers, tools to manipulate documents, to add, remove or resize pages, and to extract resources and other information from the documents. These tools are deliberately kept as 'thin' as possible. The heavy lifting is all performed by the core library, so as to be as reusable as possible.

Finally, the MuPDF distribution includes bindings to reflect the MuPDF C API into other languages, such as Java and Javascript.

1.2 License

MuPDF is released under two licenses.

Firstly, it is available under the GNU Afferro General Purpose License (henceforth the GNU AGPL). This is a complex license worthy of careful study and more words than we have space for here. Some key points, however are:

- You are free to use MuPDF within a piece of software written entirely for your own use with no problems. The moment you pass that software to any other person, or make it available to any other person as part of a "Software as a service" installation, you **must** abide by the following terms.
- If you link MuPDF into your own software, then the entirety of that software must be licensed under the GNU AGPL.
- If you use MuPDF as part of a "Software as a service" installation, then you must license the entirety of that installation under the GNU AGPL.
- Releasing a piece of software under the GNU AGPL requires you to be prepared to give full source code to any user that receives a copy of the software. No charge (other than nominal media costs) may be made for this.
- You must ensure that all end users of that system have the ability to update the software with an updated version of MuPDF. This includes embedded systems.
- Using MuPDF under the GNU AGPL, you receive no warranty and no support.

There are other terms too, and we strongly recommend that you read the license in full and understand your obligations under it before developing code based upon MuPDF.

If you find that you can abide by all the terms of the GNU AGPL, you can use MuPDF in your own projects without any license fee.

These terms, however, are generally stringent enough that they are inappropriate for people producing commercial products - giving the source code to a commercial product away is generally unacceptable, and the 'relinking' requirements of the GNU AGPL are too cumbersome for embedded users.

It is for this reason that Artifex (the developers of MuPDF) offer commercial licenses. Contact sales@artifex.com for a quote tailored to your exact needs.

The Artifex commercial license removes all the onerous terms of the GNU AGPL, including the need to license your entire app, to give away source, and to ensure relinking capabilities.

If you find yourself unable to accept and comply with the terms of the GNU AGPL, and unwilling to obtain a Commercial license from Artifex, you cannot legally use MuPDF in any software that you distribute.

1.3 Dependencies

The core MuPDF library makes use of various software libraries.

Freetype Renderer for various font types.

Harfbuzz OpenType Font shaper built upon freetype, required for e-pub files.

JBig2dec Image decoder for JBIG2 images.

JpegLib Image decoder for JPEG images.

MuJS Javascript engine used for PDF files.

OpenJPEG Image decoder for JPEG2000 images.

ZLib Compression library.

In addition, the MuPDF library can optionally make use of:

OpenSSL Encryption library, required for Digital Signatures support.

Finally, the MuPDF viewer for Linux and Windows can optionally make use of:

Curl An http fetcher used for displaying files as they download.

These libraries are packaged with MuPDF, either in the distribution archives or as git submodules. From time to time, these libraries may include bug fixes that have not been accepted back into the upstream repositories. We therefore strongly recommend using the versions of the libraries that we ship, rather than any other versions you may find on your system.

Chapter 2

Quick Start

2.1 How to open a document and render some pages

For a simple example of how to open a document and render some pages, see `docs/example.c`.

Chapter 3

The Context

3.1 Overview

The core MuPDF library is designed for simplicity, portability, and ease of integration. For all these reasons, it has no global variables, has no thread library dependencies, and has a well defined exception system to handle runtime errors. Nonetheless, in order to be as useful as possible, clearly the library must have some state and needs to be able to take advantage of multi-threaded environments.

The solution to these seemingly conflicting requirements is the Context (`fz_context`).

Every caller to MuPDF should create a Context at the start of its use of the library, and destroy it at the end. This Context (or one 'cloned' from it) will then be passed in to every MuPDF API call.

Global State At its simplest, the Context contains global settings for the library. For instance, the levels of Antialiasing used by the text and line art rendering routines are set in the Context, as is the default style sheet for Epub or FB2 files. In addition, the library stores its own private information there too.

Error handling All error handling within MuPDF is done using the `fz_try`/ `fz_catch` constructs; see chapter 4 Error handling for more details.

These constructs can be nested, so rely on an exception stack maintained within the context. As such it is vitally important that no two threads use the same context at the same time. See section 3.4 Multi-threading for more information.

Allocation When embedding MuPDF into a system it is often desirable to control the allocators used. A set of allocator functions can be provided to the Context at creation time, and all allocations will be performed using these. See chapter 5 Reference Counting, Memory Management and The Store for more information.

The Store MuPDF uses a memory cache to aid performance, and to avoid repeated decoding of resources from the file. The store is maintained using the context, and shared between a context and its clones. See chapter 5 Reference Counting, Memory Management and The Store for more information.

Multi-threading MuPDF does not rely on threading itself, but it can be used in a multi-threaded environment to give significant performance improvements. Any thread library can be used with MuPDF. A set of locking/ unlocking functions must be passed to the context at creation time, and the library will use these to ensure it is thread safe. See section 3.4 Multi-threading for more information.

3.2 Creation

To create a context, use `fz_new_context`:

```
/*
   fz_new_context: Allocate context containing global state.

   The global state contains an exception stack, resource store,
   etc. Most functions in MuPDF take a context argument to be
   able to reference the global state. See fz_drop_context for
   freeing an allocated context.

   alloc: Supply a custom memory allocator through a set of
   function pointers. Set to NULL for the standard library
   allocator. The context will keep the allocator pointer, so the
   data it points to must not be modified or freed during the
   lifetime of the context.

   locks: Supply a set of locks and functions to lock/unlock
   them, intended for multi-threaded applications. Set to NULL
   when using MuPDF in a single-threaded applications. The
   context will keep the locks pointer, so the data it points to
   must not be modified or freed during the lifetime of the
   context.

   max_store: Maximum size in bytes of the resource store, before
   it will start evicting cached resources such as fonts and
   images. FZ_STORE_UNLIMITED can be used if a hard limit is not
   desired. Use FZ_STORE_DEFAULT to get a reasonable size.
   Does not throw exceptions, but may return NULL.
*/
fz_context *fz_new_context(const fz_alloc_context *alloc, const
    fz_locks_context *locks, unsigned int max_store);
```

For example, a simple, single threaded program using the standard allocator can just use:

```
fz_context *ctx = fz_new_context(NULL, NULL, FZ_STORE_UNLIMITED);
```

3.3 Custom Allocators

In some circumstances it can be desirable to force all allocations through a set of 'custom' allocators. These are defined as an `fz_alloc_context` structure whose address is passed in to `fz_new_context`. This structure must exist for the life time of the returned `fz_context` (and any clones).

```
typedef struct
{
   void *user;
   void *(*malloc)(void *, size_t);
   void *(*realloc)(void *, void *, size_t);
   void (*free)(void *, void *);
} fz_alloc_context;
```

The `malloc`, `realloc` and `free` function pointers have essentially the same semantics as the standard `malloc`, `realloc` and `free` standard functions, with the exceptions that they take an additional initial argument - that of the `user` value specified in the `fz_alloc_context`.

3.4 Multi-threading

MuPDF itself does not rely on a thread system, but it will make use of one if one is present. This is crucial to ensure that MuPDF can be called from multiple threads at once.

A typical example of this might be in a multi-core processor on a printer. We can interpret the PDF file to a display list, and then render 'bands' from that display list to send to the printer. By using multiple threads we can render multiple bands at once, thus vastly improving processing times.

In this example, although each thread will be rendering different things, they will probably share some information - for instance the same font is likely to be used in multiple bands. Rather than have every thread render all the glyphs that it needs from the font independently, it would be nice if they could collaborate and share results.

We therefore arrange that data structures such as the font cache can be shared between the different threads. This, however, brings dangers; what if two threads try to write to the same data structure at once?

To save this being a problem, we rely on the user providing some locking functions for us.

```
/*
   Locking functions

   MuPDF is kept deliberately free of any knowledge of particular
   threading systems. As such, in order for safe multi-threaded
   operation, we rely on callbacks to client provided functions.

   A client is expected to provide FZ_LOCK_MAX number of mutexes,
   and a function to lock/unlock each of them. These may be
   recursive mutexes, but do not have to be.

   If a client does not intend to use multiple threads, then it
   may pass NULL instead of a lock structure.

   In order to avoid deadlocks, we have one simple rule
   internally as to how we use locks: We can never take lock n
   when we already hold any lock i, where 0 <= i <= n. In order
   to verify this, we have some debugging code, that can be
   enabled by defining FITZ_DEBUG_LOCKING.
*/

typedef struct
{
   void *user;
   void (*lock)(void *user, int lock);
   void (*unlock)(void *user, int lock);
} fz_locks_context;

enum {
   ...
   FZ_LOCK_MAX
};
```

If MuPDF is to be used in a multi-threaded environment, then the user is expected to define `FZ_LOCK_MAX` locks (currently 4, though this may change in future), together with functions to lock and unlock them.

In pthreads, a lock might be implemented by `pthread_mutex_t`. In windows, either `Mutex` or a `CriticalSection` might be used (the latter being more lightweight).

These locks are not assumed to be recursive (though recursive locks will work just fine).

To avoid deadlocks, MuPDF guarantees never to take lock n if that thread already holds lock m (for n ¿ m).

There are 3 simple rules to follow when using MuPDF in a multi threaded environment:

1. **No simultaneous calls to MuPDF in different threads are allowed to use the same context.**

   Most of time it is simplest just to use a different context for every thread; just create a new context at the same time as you create the thread. See section 3.5 Cloning for more information.

2. **No simultaneous calls to MuPDF in different threads are allowed to use the same document.**

   Only one thread can be accessing an document at a time. Once display lists are created from that document, multiple threads can operate on them safely.

   The document can safely be used from several different threads as long as there are safeguards in place to prevent the usages being simultaneous.

3. **No simultaneous calls to MuPDF in different threads are allowed to use the same device.**

   Calling a device simultaneously from different threads will cause it to get confused and may crash. Calling a device from several different threads is perfectly acceptable as long as there are safeguards in place to prevent the calls being simultaneous.

3.5 Cloning

The context contains the exception stack for the `fz_try`/`fz_catch` constructs. As such trying to use the same context from multiple threads at the same time will lead to crashes.

The solution to this is to ‘clone’ the context. Each clone will share the same underlying store (and will inherit the same settings, such as allocators, locks etc), but will have its own exception handle. Other settings, such as anti-alias levels, will be inherited from the original at the time of cloning, but can be changed to be different if required.

For example, in a viewer application, we might want to have a background process that runs through the file generating page thumbnails. In order for this not to interfere with the foreground process, we would clone the context, and use the cloned context in the thumbnailing thread. We might choose to disable anti-aliasing for the thumbnailing thread to trade quality for speed.

Any images decoded for the thumbnailing thread would live on in the store though, and would hence be available should the viewers normal render operations need them.

To clone a context, use `fz_clone_context`:

```
/*
   fz_clone_context: Make a clone of an existing context.

   This function is meant to be used in multi-threaded
   applications where each thread requires its own context, yet
   parts of the global state, for example caching, is shared.

   ctx: Context obtained from fz_new_context to make a copy of.
   ctx must have had locks and lock/functions setup when created.
   The two contexts will share the memory allocator, resource
   store, locks and lock/unlock functions. They will each have
   their own exception stacks though.

   Does not throw exception, but may return NULL.
*/
fz_context *fz_clone_context(fz_context *ctx);
```

For example:

```
fz_context *worker_ctx = fz_clone_context(ctx);
```

In order for cloned contexts to work safely, they rely on being able to take locks around certain operations to make them atomic. Accordingly, `fz_clone_context` will return NULL (to indicate failure) if the base context did not have locking functions defined.

3.6 Destruction

Once you have finished with an `fz_context` (either your original one, or a ‘cloned’ one) you can destroy it using `fz_drop_context`.

```
/*
   fz_drop_context: Free a context and its global state.

   The context and all of its global state is freed, and any
   buffered warnings are flushed (see fz_flush_warnings). If NULL
   is passed in nothing will happen.
   Does not throw exceptions.
*/
void fz_drop_context(fz_context *ctx);
```

For example:

```
fz_drop_context(ctx);
```

3.7 Tuning

Some of MuPDF's functionality relies on heuristics to make decisions. Rather than hard code these decisions in the library code, the tuning context allows callers to override the defaults with their own 'tuned' versions.

Currently, we have just 2 calls defined here, both to do with image handling, but this may expand in future.

The first tuning function enables fine control over how much of an image MuPDF should decode if it only requires a subarea:

```
/*
   fz_tune_image_decode_fn: Given the width and height of an image,
   the subsample factor, and the subarea of the image actually
   required, the caller can decide whether to decode the whole image
   or just a subarea.

   arg: The caller supplied opaque argument.

   w, h: The width/height of the complete image.

   l2factor: The log2 factor for subsampling (i.e. image will be
   decoded to (w>>l2factor, h>>l2factor)).

   subarea: The actual subarea required for the current operation.
   The tuning function is allowed to increase this in size if required.
*/
typedef void (fz_tune_image_decode_fn)(void *arg, int w, int h, int
    l2factor, fz_irect *subarea);
```

Having defined a function of this type to implement the desired strategy, it can be set into the context using:

```
/*
   fz_tune_image_decode: Set the tuning function to use for
   image decode.
   image_decode: Function to use.

   arg: Opaque argument to be passed to tuning function.
*/
void fz_tune_image_decode(fz_context *ctx, fz_tune_image_decode_fn
    *image_decode, void *arg);
```

The second function allows fine control over the scaling used when images are scaled:

```
/*
   fz_tune_image_scale_fn: Given the source width and height of
   image, together with the actual required width and height,
   decide whether we should use mitchell scaling.

   arg: The caller supplied opaque argument.

   dst_w, dst_h: The actual width/height required on the target device.

   src_w, src_h: The source width/height of the image.

   Return 0 not to use the Mitchell scaler, 1 to use the Mitchell
       scaler. All
   other values reserved.
*/
typedef int (fz_tune_image_scale_fn)(void *arg, int dst_w, int dst_h,
    int src_w, int src_h);
```

Having defined a function of this type to implement the desired strategy, it can be set into the context using:

```
/*
   fz_tune_image_scale: Set the tuning function to use for
   image scaling.

   image_scale: Function to use.

   arg: Opaque argument to be passed to tuning function.
*/
void fz_tune_image_scale(fz_context *ctx, fz_tune_image_scale_fn
    *image_scale, void *arg);
```

3.8 Summary

The basic usage of Contexts is as follows:

1. Call `fz_new_context` to create a context. Pass in any custom allocators

required. If you wish to use MuPDF from multiple threads at the same time, you must also pass in locking functions. Set the store size appropriately.

2. Call `fz_clone_context` to clone the context as many times as you need; typically once for each 'worker' thread.
3. Perform the operations required using MuPDF within `fz_try`/`fz_catch` constructs.
4. Call `fz_drop_context` with each cloned context.
5. Call `fz_drop_context` with the original context.

Things to remember:

1. A `fz_context` can only be used in 1 thread at a time.
2. A `fz_document` can only be used in 1 thread at a time.
3. A `fz_device` can only be used in 1 thread at a time.
4. A `fz_context` shares the store with all the `fz_context`s cloned from it.

Chapter 4

Error handling

4.1 Overview

MuPDF handles all its errors using an exception system. This is superficially similar to C++ exceptions, but (as MuPDF is written in C) it is implemented using macros that wrap the `setjmp`/`longjmp` standard C functions.

It is probably best not to peek behind the curtain, and just to think of these constructs as being extensions to the language. Indeed, we have worked very hard to ensure that the complexities involved are minimised.

Unless otherwise specified, all MuPDF API functions can throw exceptions, and should therefore be called within an `fz_try`/`fz_always`/`fz_catch` construct.

The general anatomy of such a construct is as follows:

```
fz_try(ctx)
{
    /* Do stuff in here that might throw an exception.
     * NEVER return from here. 'break' can be used to
     * continue execution (either in the always block or
     * after the catch block). */
}
fz_always(ctx)
{
    /* Anything in here will always be executed, regardless
     * of whether the fz_try clause exited normally, or an
     * exception was thrown. */
}
fz_catch(ctx)
{
    /* This block will execute if (and only if) anything in
     * the fz_try block calls fz_throw. We should clean up
     * anything we need to. If we are in a nested fz_try/
     * fz/catch block, we can call fz_rethrow to propagate
     * the error to the enclosing catch. */
}
```

The `fz_always` block is completely optional. The following is perfectly valid:

```
fz_try(ctx)
{
    /* Do stuff here */
}
fz_catch(ctx)
{
    /* Clean up from errors here */
}
```

In an ideal world, that would be all there is to it. Unfortunately, there are 2 wrinkles.

The first one, relatively simple, is that you must not return from within a `fz_try` block. To do so will corrupt the exception stack and cause problems and crashes. To mitigate this, you can safely `break` out of the `fz_try`, and execution will pass into the `fz_always` block (if there is one, or continue after the `fz_catch` block if not).

The second one, is more convoluted. If you do not wish to understand the long and complex reasons behind this, skip forward to the summary and just follow the rules there.

As stated before `fz_try`/ `fz_catch` are implemented using `setjmp`/`longjmp`, and these can 'lose' changes to variables.

For example:

```
house_t *build_house(fz_context *ctx)
{
    walls_t *w = NULL;
    roof_t  *r = NULL;
    house_t *h = NULL;

    fz_try(ctx)
    {
        w = make_walls();
        r = make_roof();
        h = combine(w, r); /* Note, NOT: return combine(w,r); */
    }
    fz_always(ctx)
    {
        drop_walls(w);
        drop_roof(r);
    }
    fz_catch(ctx)
    {
        return NULL; /* Or fz_rethrow if we're nested */
    }
    return h;
}
```

In the above code (as well as throughout MuPDF), we follow the convention that destructors always accept NULL. This makes cleanup code much simpler.

Reading through this code, it is fairly obvious what will happen if everything works correctly. First we'll make some walls, `w`, and a roof, `r`. Then we combine the walls and the roof, to get our house, `h`. Next we tidy up the walls and the roof, and we return the completed house to our caller.

It's more interesting to consider what will happen if we have failures.

First let's consider what happens if the `make_walls` fails. This will `fz_throw` an exception, and control will jump immediately to the `fz_always`. This will drop `w` and `r` (both of which are still NULL). The `fz_catch` can then handle the error, either by returning NULL, to indicate failure, or perhaps by `fz_rethrow`ing the error to an enclosing `fz_try`/ `fz_catch` construct. No problems there.

So what happens when the failure occurs in `make_roof` fails? Let's run through the code again.

This time, `make_walls` succeeds, and `w` is set to this new value. Then `make_roof` fails, `fz_throw`ing an exception, and control will jump immediately to the `fz_always`. This will then try to drop `w` (now a valid value) and `r` (which is still NULL). The `fz_catch` can then handle the error, either by returning NULL, to indicate failure, or perhaps by `fz_rethrow`ing the error to an enclosing `fz_try`/ `fz_catch` construct. All sounds quite plausible.

Unfortunately, if you try it, on some systems you will find that you have a memory leak (or worse). When `drop_walls` is called, sometimes you will find that `w` has 'lost' its value.

This is due to an obscure part of the C specification that states that any changes to local variables made between a `setjmp` and a `longjmp` can be lost.

In `fz_try`/ `fz_catch` terms, this means that any local variables set within the `fz_try` block can be 'lost' when either `fz_always` or `fz_catch` are reached.

Fortunately, there is a fix for this, `fz_var`. By calling `fz_var(w);` we can 'protect' variable `w` from such unwanted behaviour.

It's not really necessary to know how this works, but for those interested, a quick explanation. The 'loss' of the value occurs because the compiler can postpone writing the value back into the storage location for the variable (or can choose to just hold it in a register). The call to `fz_var` passes the address of the variable out of scope; this forces the compiler not to hold it in a register. Further, the compiler has no way of knowing whether any functions it call might access that location, so it needs to make sure that the variable value is written back on every function call - such as longjmp. Hence the variable is magically protected.

Calls to `fz_var` are very low cost (but are not NOPs), so erring on the side of caution and calling `fz_var` on more than you need to will probably not hurt.

A corrected version of the above example is therefore:

```
house_t *build_house(fz_context *ctx)
{
    walls_t *w = NULL;
    roof_t  *r = NULL;
    house_t *h = NULL;

    fz_var(w);
    fz_var(r);

    fz_try(ctx)
    {
        w = make_walls();
        r = make_roof();
        h = combine(w, r); /* Note, NOT: return combine(w,r); */
    }
    fz_always(ctx)
    {
        drop_walls(w);
        drop_roof(r);
    }
    fz_catch(ctx)
    {
        return NULL; /* Or fz_rethrow if we're nested */
    }
    return h;
}
```

4.2 Throwing exceptions

Most client code need never worry about anything more than catching exceptions thrown by the core core. If you are implementing your own devices or extending the core of MuPDF, then you will need to know how to generate (and pass on) your own exceptions.

An exception is constructed and thrown from an integer code and a `printf` like string:

```
enum
```

```
{
    FZ_ERROR_NONE = 0,
    FZ_ERROR_GENERIC = 1,
    FZ_ERROR_SYNTAX = 2,
    FZ_ERROR_TRYLATER = 3,
    FZ_ERROR_ABORT = 4,
    FZ_ERROR_COUNT
};

void fz_throw(fz_context *ctx, int errcode, const char *, ...);
```

In almost all cases, you should be using `FZ_ERROR_GENERIC`, for example:

```
fz_throw(ctx, FZ_ERROR_GENERIC, "Failed to open file '%s'", filename);
```

4.3 Handling exceptions

Once you have caught an exception, most code will simply tidy up any loose resources (to prevent leaks), and rethrow the exception up to a higher layer handler.

At the top level of the program, clearly this is not an option. The catch clause needs to return the error using whatever process the calling program is using for error handling.

Details of the message from the caught error can be read using:

```
const char *fz_caught_message(fz_context *ctx);
```

The error will remain readable in this way until the next use of `fz_try`/`fz_catch` on that same context.

Some code may choose to swallow the error and retry the same code again in a different manner. To facilitate this, we can find out the type of error using:

```
int fz_caught(fz_context *ctx);
```

See section 4.2 Throwing exceptions for a list of the possible exception types.

To simplify the job of deciding whether to pass on exceptions of a given type, we have a convenience function that with rethrow just a particular type.:

```
void fz_rethrow_if(fz_context *ctx, int errcode);
```

4.4 Summary

The basic exception handling rules are as follows:

1. All MuPDF functions except those that explicitly state otherwise, throw exceptions on errors, and must therefore be called from within an `fz_try`/ `fz_catch` construct.
2. A `fz_try` block must be paired with an `fz_catch` block, and optionally an `fz_always` block can appear between them.
3. Never return from an `fz_try` block.
4. An `fz_try` block will terminate when control reaches the end of the block, or when `break` is called.
5. Any variable that is changed within an `fz_try` block may lose its value if an exception occurs, unless protected by `fz_var` call.
6. The contents of the `fz_always` block will always be executed (after the `fz_try` block and before the `fz_catch` block, if appropriate).
7. If an exception is thrown during the `fz_try` block, control will jump to the `fz_always` block (if there is one) and then continue to the `fz_catch` block.

Chapter 5

Reference Counting, Memory Management and The Store

5.1 Overview

While MuPDF is running, it holds various objects in memory, and passes them between its various components. For instance, MuPDF might read a path definition in in the PDF interpreter, and pass it first into the display list and then on into the renderer.

To avoid needless copying of data, a reference counting scheme is used. Each significant object has a reference count, so that when one area of the code retains a reference to something (perhaps the display list), the data need not be copied wholesale. In the above example, the PDF interpreter might hold one reference, and first the display list and then the renderer might take others. Some references are held just for a short length of time, but others can persist for a much longer period.

During the course of displaying files, MuPDF loads various resources into memory, such as fonts and images. By holding these resources in memory throughout the processing of the file we can avoid reloading them each time they are required.

As the document is rendered, more memory is needed to hold rendered versions of glyphs from the font, or decoded versions of images. By keeping these decoded versions around in memory, we can avoid the need to redecode them the next time we need the same glyph, or the same image.

Keeping all this data around can end up using a large amount of memory, which may be unfeasible for some systems. Equally, not keeping any of it around will result in a drastic performance drop.

The solution is to keep as much around as can conveniently fit in memory. MuPDF achieves this using a mechanism known as "The Store".

The Store is a mechanism for holding blocks of data likely to be reusable. Whenever MuPDF needs such a block of data, it checks the Store to see if the data is there already - if it is, it can be instantly reused. If not the code forms the data itself, and then puts it into the store.

The MuPDF allocation code is tied into the store, so that if an allocation ever fails, objects are evicted from the store, and the allocation retried. This 'scavenging' of memory means that we can safely keep lots of cached data around without ever worrying that it will cause us to run out of memory.

5.2 Reference Counting

As mentioned above, most MuPDF objects are reference counted. This means that on creation (typically with an `fz_new_...` call), they have a reference count of 1. Think of these object pointers as 'handles'.

If a 'copy' of the object is required, a new handle can be generated using the appropriate `fz_keep_...` call. This is a very low cost operation that just involves incrementing the reference count, so no physical copying of the data is involved. Accordingly it is vital that objects that have multiple handles do not have their contents altered.

Once a reference is finished with, it should be disposed of using the appropriate `fz_drop_...` call. This is true regardless of whether the handle was created by a `fz_new_...` or a `fz_keep_...` call. This drops the reference count by 1.

Once the reference count hits 0, the storage used by the object is freed.

As an implementation detail, certain objects within MuPDF are allocated statically and have a reference count of -1. These are unaffected by reference counting operations, and will never be freed. Nonetheless, these should be treated exactly as for normal objects and kept/dropped as usual.

5.2.1 Implementation

These keep and drop calls for simple objects are generally implemented by using one of a set of standard functions. There are a range of these, depending on the expected size of the reference counts, and all handle the locking required to ensure thread safety:

```
void *fz_keep_imp(fz_context *ctx, void *p, int *refs);
void *fz_keep_imp8(fz_context *ctx, void *p, int8_t *refs);
void *fz_keep_imp16(fz_context *ctx, void *p, int16_t *refs);
int fz_drop_imp(fz_context *ctx, void *p, int *refs);
int fz_drop_imp8(fz_context *ctx, void *p, int8_t *refs);
int fz_drop_imp16(fz_context *ctx, void *p, int16_t *refs);
```

As an example, an `fz_path` structure is defined as:

```
typedef struct {
  int8_t refs;
} fz_path;
```

and thus appropriate keep and drop functions can be defined simply:

```
fz_path *fz_keep_path(fz_context *ctx, fz_path *path)
{
    return fz_keep_imp8(ctx, &path->refs);
}

void fz_drop_path(fz_context *ctx, fz_path *path)
{
    if (!fz_drop_imp8(ctx, &path->refs))
        return;
    <code to free the contents of the path structure>
}
```

More complex variations of these functions are available to cope with ‘storable’ objects, and still more complex versions to cope with ‘key storable’ objects - these are explained in the following sections.

However they are implemented, these objects all look basically the same to most users - they can simply be ‘kept’ and ‘dropped’ as required.

5.3 Creating the Store

The Store is created as part of the `fz_new_context` call, (see the Context chapter) and is shared with any contexts obtained with `fz_clone_context`. The ‘store limit’ is specified as a byte size as part of this call. A special value of `FZ_STORE_UNLIMITED` (0) is used to indicate that no amount of memory is too much.

5.4 Using the store

5.4.1 Overview

Every "storable piece of information" in MuPDF is held in a data structure that begins with an `fz_storable` structure. Rather than repeatedly say "a storable piece of information", we shall henceforth just say "an `fz_storable`".

MuPDF uses reference counting for most of its data structures (see section 5.2 Reference Counting), and `fz_storable`s are no exception.

Whenever MuPDF needs to use a `fz_storable`, it first checks to see if there is one in the Store already. It does this by forming a unique 'key' and scanning the Store for an object of a given type, with that key. If the object exists within the Store, the fact that the object has been used is noted, a reference is taken, and returned to the caller.

If no reference is returned, the code creates its own version of the `fz_storable`. It calculates its size, and puts it into the Store, together with the same key as before. The Store takes a reference to the object, links it into its data structure, and updates its running total of the size of all the objects within it.

If placing a new object into the store would take it over the limit, it runs through and looks for the least recently used objects to evict to bring the limit down. In order for an object to be considered for eviction, their refcount must be 1. We know that the Store is holding 1 reference to the object - if anything else is, then removing it from the Store won't actually save us any memory.

Regardless of whether the Store can be reduced to a suitable size, the object is always placed into the store. This ensures that the Store's figure for "amount of memory used by `fz_storable`'s" remains correct (thus ensuring that should objects become evictable, the store size will fall correctly). It also does no harm, because clearly we have managed to allocate enough memory to form the `fz_storable` in the first place.

Regardless of whether a caller finds the object in the Store, or has to store it itself, it then proceeds identically. It uses the object for whatever purpose it needed it, and then calls the appropriate `fz_drop` function to lose its reference. The object will live on in the Store until it needs to be evicted to make room.

5.4.2 Handling keys

As discussed above, the Store is basically a set of key/value pairs. While the values are always `fz_storable`s, the keys can be of many different types, due to coming from many disparate parts of the system.

Accordingly, we need a mechanism to allow us to safely know what 'type' a given key is, and to compare 2 keys of identical type.

We solve this, by using an fz_store_type structure:

```
typedef struct fz_store_type_s
{
    int (*make_hash_key)(fz_context *ctx, fz_store_hash *, void *);
    void *(*keep_key)(fz_context *,void *);
    void (*drop_key)(fz_context *,void *);
    int (*cmp_key)(fz_context *ctx, void *, void *);
    void (*print)(fz_context *ctx, fz_output *out, void *);
    int (*needs_reap)(fz_context *ctx, void *);
} fz_store_type;
```

We will have just one instance of this for each type - normally a static const structure defined in the code. Whenever we insert (or lookup) something in the store, we pass the address of that 'types' structure.

We only compare items if they have the same type pointer, and any comparison is done using the cmp_key function pointer therein. In common with normal C idioms, 0 means match, non zero means different.

The keep_key and drop_key entries are used to implement reference counting of keys. Because keys can be an amalgam of several reference counted objects, the keep and drop functions provided here can take and drop these in sync.

The print function is purely for debugging purposes as part of calls to fz_print_store - it should generate a human readable summary of the key to the given fz_output stream.

The make_hash_key and needs_reap functions are explained in the following subsections.

5.4.3 Hashing

In order to ensure the Store performs well, we must ensure that certain processes run efficiently - notably searching for an existing entry, insertion and deletion.

Accordingly, the Store is implemented based on a hash table. For every 'key', we need to be able to form a hash, but this process is complicated slightly by the fact that every different fz_storable has a different type for the key.

We solve this by having the make_hash_key member of the fz_store_type structure convert whatever its key data is into a common structure:

```
typedef struct fz_store_hash_s
{
    fz_store_drop_fn *drop;
    union
    {
        struct
        {
            const void *ptr;
            int i;
        } pi;
        struct
        {
            const void *ptr;
            int i;
            fz_irect r;
        } pir;
        struct
        {
            int id;
            float m[4];
        } im;
    } u;
} fz_store_hash;
```

The caller will always arrange for this structure to be zero filled on entry to the `make_hash_key` call. On exit, it should have been updated with the key details. Implementers may extend the union found in this structure as required, though ideally the size of the overall structure should be minimised to avoid unnecessary work.

Once the Store has formed a `fz_store_hash` it can then generate the required hash for the hashtable as required.

5.4.4 Key storable items

Some objects can be used both as values within the Store, and as a component of keys within the Store. We refer to these objects as "key storable" objects. In this case, we need to take additional care to ensure that we do not end up keeping an item within the store, purely because its value is referred to by another key in the store.

An example of this are `fz_images` in PDF files. Each `fz_image` is placed into the Store to enable it to be easily reused. When the image is rendered, a pixmap is generated from the image, and the pixmap is placed into the Store so it can be reused on subsequent renders. The image forms part of the key for the pixmap.

When we close the pdf document (and any associated pages/display lists etc), we drop the images from the Store. This may leave us in the position of the images having non-zero reference counts purely because they are used as part of the keys for the pixmaps.

We therefore use special reference counting functions to implement these `fz_key_storable` items, `fz_keep_key_storable` and `fz_drop_key_storable` rather than the more usual `fz_keep_storable` and `fz_drop_storable`.

The sole difference is that these enable us to store the number of references to these items that are used in keys. This is achieved by callers taking and dropping references for use in keys with `fz_keep_key_storable_key` and `fz_drop_key_storable_key`.

This means that key storable items need to provide two sets of keep and drop functions, one for 'normal' callers, and one for use during key handling. For example:

```
fz_image *fz_keep_image(fz_context *ctx, fz_image *image);
void fz_drop_image(fz_context *ctx, fz_image *image);

fz_image *fz_keep_image_store_key(fz_context *ctx, fz_image *image);
void fz_drop_image_store_key(fz_context *ctx, fz_image *image);
```

The purpose of this extra work is to allow us to spot when we may need to check the Store for 'dead' entries - those that can never be 'found' by looking in the store.

5.4.5 Reap passes

When the number of references to a key storable object equals the number of references to an object from keys in the Store, we know that we can remove all the items which have that object as part of the key. This is done by running a pass over the store, 'reaping' those items.

If a key does not consist of any storable objects, then the `needs_reap` entry in its `fz_store_type` can safely be left as NULL. If it does, however, it must provide an implementation to check whether a reap pass is required. Essentially this needs to check if any of its constituent `fz_key_storable` objects need reaping, which can be done by a call to:

```
int fz_key_storable_needs_reaping(fz_context *ctx, const fz_key_storable
    *ks);
```

Reap passes are slower than we would like as they touch every item in the store. We therefore provide a way to 'batch' such reap passes together, using `fz_defer_reap_start` and `fz_defer_reap_end` to bracket a region in which many may be triggered.

5.5 Scavenging memory allocator

All allocations within MuPDF (and its sub-libraries) call `fz_malloc` and family. These functions ultimately call down to the custom allocator functions passed into the `fz_new_context` call (or to `malloc` and family if no custom allocators were supplied). (See chapter 3 The Context for details).

If a call to the underlying custom allocator fails, MuPDF will automatically seek to evict the least recently used objects from the store that are not currently being used, and then will retry the allocation. This can happen several times, with more and more objects being freed between each attempt.

Allocation failures are therefore only fatal to MuPDF if there are no remaining objects to be freed in the store.

This 'just in time' scavenging of memory means that the store limit can safely be set to a high level (or to be unlimited), and MuPDF will still operate within safe bounds.

5.6 Reacting to Out of Memory events

As a last resort, applications using MuPDF can react to low memory events by changing their strategy. For example, if we fail to render a band of data due to an allocation failure, we might back off and try a smaller band size. Alternatively, we might choose to dispense with the display list, and to reinterpret the underlying file directly each time, trading speed for memory.

To this end, all exceptions thrown due to allocation failures have the `FZ_ERROR_OOM` type, enabling callers to easily distinguish them using `fz_caught` and to react accordingly.

Chapter 6

The Document interface

6.1 Overview

Although MuPDF handles multiple different file formats, it offers a unified API for dealing with them. The `fz_document` API allows all the common operations to be performed on a document, hiding the implementation specifics away from the caller.

Not all functions are available on all document types (for instance, JPEG files do not support annotations), but the API returns sane values.

6.2 Opening/Closing a document

The simplest way to load a document is to load it from the local filing system:

```
/*
   fz_open_document: Open a PDF, XPS or CBZ document.

   Open a document file and read its basic structure so pages and
   objects can be located. MuPDF will try to repair broken
   documents (without actually changing the file contents).

   The returned fz_document is used when calling most other
   document related functions.

   filename: a path to a file as it would be given to open(2).
*/
fz_document *fz_open_document(fz_context *ctx, const char *filename);
```

For embedded systems, or secure applications, the use of a local filing system may be inappropriate, so an alternative is available whereby documents can be opened from an `fz_stream`. See chapter 10 The Stream interface for more details on `fz_stream`s.

```
/*
   fz_open_document_with_stream: Open a PDF, XPS or CBZ document.

   Open a document using the specified stream object rather than
   opening a file on disk.

   magic: a string used to detect document type; either a file name or
       mime-type.
*/
fz_document *fz_open_document_with_stream(fz_context *ctx, const char
    *magic, fz_stream *stream);
```

Almost any data source can be wrapped up as an `fz_stream`; see chapter 10 The Stream interface for more details.

In common with most other objects in MuPDF, `fz_document`s are reference counted:

```
/*
   fz_keep_document: Keep a reference to an open document.

   Does not throw exceptions.
*/
fz_document *fz_keep_document(fz_context *ctx, fz_document *doc);

/*
   fz_drop_document: Release an open document.

   The resource store in the context associated with fz_document
   is emptied, and any allocations for the document are freed when
   the last reference is dropped.

   Does not throw exceptions.
*/
void fz_drop_document(fz_context *ctx, fz_document *doc);
```

Once the last reference to the document is dropped, all resources used by that document will be released, including those in the Store.

6.3 Handling password protected documents

Some document types (such as PDF) can require passwords to allow the file to be opened. After you have obtained an `fz_document`, you should therefore check whether it needs a password using `fz_needs_password`:

```
/*
    fz_needs_password: Check if a document is encrypted with a
    non-blank password.

    Does not throw exceptions.
*/
int fz_needs_password(fz_context *ctx, fz_document *doc);
```

If a password is required, you can supply one using `fz_authenticate_password`:

```
/*
    fz_authenticate_password: Test if the given password can
    decrypt the document.

    password: The password string to be checked. Some document
    specifications do not specify any particular text encoding, so
    neither do we.

    Does not throw exceptions.
*/
int fz_authenticate_password(fz_context *ctx, fz_document *doc, const
     char *password);
```

6.4 Handling reflowable documents

Some document types (such as Epub) require the contents to be laid out before they can be rendered. This is done by calling `fz_layout_document`:

```
/*
    fz_layout_document: Layout reflowable document types.

    w, h: Page size in points.
    em: Default font size in points.
*/
void fz_layout_document(fz_context *ctx, fz_document *doc, float w,
     float h, float em);
```

Any non-reflowable document types (such as PDF) will ignore this layout request. The results of the layout will depend both upon a target width and height, a given font size, the CSS styles in effect. Documents can be laid out multiple times to allow changes in these properties to take effect.

MuPDF provides its own default CSS style sheet, but this can be overridden by the user CSS style sheet in the context:

```
/*
   fz_user_css: Get the user stylesheet source text.
*/
const char *fz_user_css(fz_context *ctx);

/*
   fz_set_user_css: Set the user stylesheet source text for use with
       HTML and EPUB.
*/
void fz_set_user_css(fz_context *ctx, const char *text);
```

The user CSS style sheet is supplied as a null terminated C string.

When the CSS or the screen size is changed, and the document relaid out, content moves. In order for applications to be able to not lose the readers place, MuPDF offers a mechanism for making a bookmark and then looking it up again after the content has been laid out to a new position.

```
/*
   Create a bookmark for the given page, which can be used to find the
   same location after the document has been laid out with different
   parameters.
*/
fz_bookmark fz_make_bookmark(fz_context *ctx, fz_document *doc, int
    page);

/*
   Find a bookmark and return its page number.
*/
int fz_lookup_bookmark(fz_context *ctx, fz_document *doc, fz_bookmark
    mark);
```

6.5 Getting Pages from a document

Once you have a laid out document, you presumably want to be able to do something with it. The first thing to know is how many pages it contains. This is achieved by calling `fz_count_pages`:

```
/*
   fz_count_pages: Return the number of pages in document
```

```
    May return 0 for documents with no pages.
*/
int fz_count_pages(fz_context *ctx, fz_document *doc);
```

For document types like images, they appear as a single page. If you forget to lay out a reflowable document, this will trigger a layout for a default size and return the required number of pages.

Once you know how many pages there are, you can fetch the `fz_page` object for each page required:

```
/*
    fz_load_page: Load a page.

    After fz_load_page is it possible to retrieve the size of the
    page using fz_bound_page, or to render the page using
    fz_run_page_*. Free the page by calling fz_drop_page.

    number: page number, 0 is the first page of the document.
*/
fz_page *fz_load_page(fz_context *ctx, fz_document *doc, int number);
```

The pages of a document with n pages are numbered from 0 to n-1.

In common with most other object types, `fz_page`s are reference counted:

```
/*
    fz_keep_page: Keep a reference to a loaded page.

    Does not throw exceptions.
*/
fz_page *fz_keep_page(fz_context *ctx, fz_page *page);

/*
    fz_drop_page: Free a loaded page.

    Does not throw exceptions.
*/
void fz_drop_page(fz_context *ctx, fz_page *page);
```

Once the last reference to a page is dropped, the resources it consumes are all released automatically.

6.6 Anatomy of a Page

In MuPDF terminology (largely borrowed from PDF) Pages consist of Page Contents, Annotations, and Links.

Page Contents (or just Contents) are typically the ordinary printed matter that you would get on a page; the text, illustrations, any headers or footers, and maybe some printers marks.

Annotations are normally extra information that is overlaid on the top of these page contents. Examples include freehand scribbles on the page, highlights/ underlines/strikeouts overlaid on the text, sticky notes etc. Annotations are typically added to a document by people reading the document after it has been published rather than by the original author.

Annotations can be enumerated from the page one at a time, by first calling `fz_first_annot`, and then `fz_next_annot`:

```
/*
   fz_first_annot: Return a pointer to the first annotation on a page.

   Does not throw exceptions.
*/
fz_annot *fz_first_annot(fz_context *ctx, fz_page *page);

/*
   fz_next_annot: Return a pointer to the next annotation on a page.

   Does not throw exceptions.
*/
fz_annot *fz_next_annot(fz_context *ctx, fz_annot *annot);
```

Annotations are reference counted and can be kept and dropped as usual. They can also be bounded, by passing a rectangle to `fz_bound_annot`:

```
/*
   fz_bound_annot: Return the bounding rectangle of the annotation.

   Does not throw exceptions.
*/
fz_rect *fz_bound_annot(fz_context *ctx, fz_annot *annot, fz_rect *rect);
```

On return, the rectangle is populated with the bounding box of the annotation.

Links describe 'active' regions on the page; if the user 'clicks' within such a region typically the viewer should respond. Some links move to other places in the document, others launch external clients such as mail or web sites.

The links on a page can be read by calling `fz_load_links`:

```
/*
   fz_load_links: Load the list of links for a page.

   Returns a linked list of all the links on the page, each with
   its clickable region and link destination. Each link is
```

```
    reference counted so drop and free the list of links by
    calling fz_drop_link on the pointer return from fz_load_links.

    page: Page obtained from fz_load_page.
*/
fz_link *fz_load_links(fz_context *ctx, fz_page *page);
```

This returns a linked list of `fz_link` structures. `link->next` gives the next one in the chain.

6.7 Rendering Pages

To render a page, you first need to know how big it is. This can be discovered by calling `fz_bound_page`, passing an `fz_rect` in to be populated:

```
/*
    fz_bound_page: Determine the size of a page at 72 dpi.

    Does not throw exceptions.
*/
fz_rect *fz_bound_page(fz_context *ctx, fz_page *page, fz_rect *rect);
```

MuPDF operates on page contents (and annotations) by processing them to a Device. There are various different devices in MuPDF (and you can implement your own). See chapter 7 The Device interface for more information. For now, just consider devices to be things that are called with each of the graphical items on the page in turn.

The simplest way to process a page is to call `fz_run_page`:

```
/*
    fz_run_page: Run a page through a device.

    page: Page obtained from fz_load_page.

    dev: Device obtained from fz_new_*_device.

    transform: Transform to apply to page. May include for example
    scaling and rotation, see fz_scale, fz_rotate and fz_concat.
    Set to fz_identity if no transformation is desired.

    cookie: Communication mechanism between caller and library
    rendering the page. Intended for multi-threaded applications,
    while single-threaded applications set cookie to NULL. The
    caller may abort an ongoing rendering of a page. Cookie also
    communicates progress information back to the caller. The
    fields inside cookie are continually updated while the page is
    rendering.
*/
void fz_run_page(fz_context *ctx, fz_page *page, fz_device *dev, const
    fz_matrix *transform, fz_cookie *cookie);
```

This will cause each graphical object from the page contents and annotations in turn to be transformed, and fed to the device.

For finer control, you may wish to run the page contents, and the annotations separately:

```
/*
    fz_run_page_contents: Run a page through a device. Just the main
    page content, without the annotations, if any.

    page: Page obtained from fz_load_page.

    dev: Device obtained from fz_new_*_device.

    transform: Transform to apply to page. May include for example
    scaling and rotation, see fz_scale, fz_rotate and fz_concat.
    Set to fz_identity if no transformation is desired.

    cookie: Communication mechanism between caller and library
    rendering the page. Intended for multi-threaded applications,
    while single-threaded applications set cookie to NULL. The
    caller may abort an ongoing rendering of a page. Cookie also
    communicates progress information back to the caller. The
    fields inside cookie are continually updated while the page is
    rendering.
*/
void fz_run_page_contents(fz_context *ctx, fz_page *page, fz_device
    *dev, const fz_matrix *transform, fz_cookie *cookie);

/*
    fz_run_annot: Run an annotation through a device.

    page: Page obtained from fz_load_page.

    annot: an annotation.

    dev: Device obtained from fz_new_*_device.

    transform: Transform to apply to page. May include for example
    scaling and rotation, see fz_scale, fz_rotate and fz_concat.
    Set to fz_identity if no transformation is desired.

    cookie: Communication mechanism between caller and library
    rendering the page. Intended for multi-threaded applications,
    while single-threaded applications set cookie to NULL. The
    caller may abort an ongoing rendering of a page. Cookie also
    communicates progress information back to the caller. The
    fields inside cookie are continually updated while the page is
    rendering.
*/
void fz_run_annot(fz_context *ctx, fz_annot *annot, fz_device *dev,
     const fz_matrix *transform, fz_cookie *cookie);
```

All three of these functions (`fz_run_page`, `fz_run_page_contents`, `fz_run_annot`) take an `fz_cookie` pointer. The Cookie is a lightweight way of controlling the processing of the page. For more details, see section 7.3 Cookie. For most simple cases this can be `NULL`.

Chapter 7

The Device interface

7.1 Overview

In many ways, the Device interface is the heart of MuPDF.

When any given document handler is told to run the page (`fz_run_page`) the appropriate document interpreter serialises the page contents as a series of graphical operations, and calls the device interface to perform these actions.

Many different implementations of the device interface exist within MuPDF. The most obvious one is the Draw device. When this is called, it renders the graphical objects in turn into a Pixmap.

Alternatively we have the Structured Text device that captures the text output and forms it into an easily processable structure (for searching, or text extraction).

Some devices, such as the SVG Output device, repackage the graphical objects into a different format. The end product of these devices is a new document with (as much as possible) the same overall appearance as the initial page.

Finally, devices such as the Display List device manage to be both implementers of the interface, and callers of it. Callers can run page contents to the Display List device just once, and then replay it quickly many times over to other devices; ideal for rendering pages in bands, or repeatedly redrawing as a viewer pans and zooms around a document.

By implementing new devices callers can tap the power of MuPDF in new and interesting ways, perhaps to harness specific hardware facilities of a device.

7.2 Device Methods

Every Device in MuPDF is an extension of the `fz_device` structure. This contains a series of function pointers to implement the handling of different types of graphical object.

These function pointers are exposed to callers via convenience functions. These convenience functions should **always** be used in preference to calling the function pointers direct, as they perform various behind the scenes housekeeping functions. They also cope with the function pointers being `NULL`, as can permissibly happen when a device is not interested in a particular class of graphical object.

We describe the convenience functions here; implementers of devices can trivially extrapolate the behaviour of the function pointers from these descriptions. For example, the `fz_fill_path` function described here is implemented by the `fill_path` function pointer in the `fz_device` that takes the identical arguments and has the same return conditions.

7.2.1 Line Art

Line Art is handled by the device functions to plot paths. See section 8.9 Paths for more information.

```
void fz_fill_path(fz_context *ctx, fz_device *dev, const fz_path *path,
    int even_odd, const fz_matrix *ctm, fz_colorspace *colorspace,
    const float *color, float alpha);
void fz_stroke_path(fz_context *ctx, fz_device *dev, const fz_path
    *path, const fz_stroke_state *stroke, const fz_matrix *ctm,
    fz_colorspace *colorspace, const float *color, float alpha);
void fz_clip_path(fz_context *ctx, fz_device *dev, const fz_path *path,
    int even_odd, const fz_matrix *ctm, const fz_rect *scissor);
void fz_clip_stroke_path(fz_context *ctx, fz_device *dev, const fz_path
    *path, const fz_stroke_state *stroke, const fz_matrix *ctm, const
    fz_rect *scissor);
```

7.2.2 Text

Text is handled by the device functions to plot text. See section 8.10 Text for more information.

```
void fz_fill_text(fz_context *ctx, fz_device *dev, const fz_text *text,
    const fz_matrix *ctm, fz_colorspace *colorspace, const float
    *color, float alpha);
```

```
void fz_stroke_text(fz_context *ctx, fz_device *dev, const fz_text
    *text, const fz_stroke_state *stroke, const fz_matrix *ctm,
    fz_colorspace *colorspace, const float *color, float alpha);
void fz_clip_text(fz_context *ctx, fz_device *dev, const fz_text *text,
    const fz_matrix *ctm, const fz_rect *scissor);
void fz_clip_stroke_text(fz_context *ctx, fz_device *dev, const fz_text
    *text, const fz_stroke_state *stroke, const fz_matrix *ctm, const
    fz_rect *scissor);
void fz_ignore_text(fz_context *ctx, fz_device *dev, const fz_text
    *text, const fz_matrix *ctm);
```

The `fz_clip_text` and `fz_clip_stroke_text` functions are used to start a clip. Subsequent operations will be clipped through the areas delimited by these, until an `fz_pop_clip` is seen. See subsection 7.2.5 Clipping and Masking for more details.

7.2.3 Images

Images are handled by the device functions to plot images. See section 8.6 Images for more information.

```
void fz_fill_image(fz_context *ctx, fz_device *dev, fz_image *image,
    const fz_matrix *ctm, float alpha);
void fz_fill_image_mask(fz_context *ctx, fz_device *dev, fz_image
    *image, const fz_matrix *ctm, fz_colorspace *colorspace, const
    float *color, float alpha);
void fz_clip_image_mask(fz_context *ctx, fz_device *dev, fz_image
    *image, const fz_matrix *ctm, const fz_rect *scissor);
```

The `fz_clip_image_mask` function is used to start a clip. Subsequent operations will be clipped through the area delimited by this, until an `fz_pop_clip` is seen. See subsection 7.2.5 Clipping and Masking for more details.

7.2.4 Shadings

Shaded areas (such as radial, linear and mesh based shadings) are rendered by filling a (normally) clipped region with a shade. This is achieved by calling `fz_fill_shade`. See section 8.11 Shadings for more details.

```
void fz_fill_shade(fz_context *ctx, fz_device *dev, fz_shade *shade,
    const fz_matrix *ctm, float alpha);
```

7.2.5 Clipping and Masking

Graphical objects can be restricted to a given area using Clipping. The area to clip to can be specified as paths, text, or images as explained in subsection 7.2.1 Line Art, subsection 7.2.2 Text, and subsection 7.2.3 Images.

Each call to such a function starts a clipping group, which will be terminated by calling:

```
void fz_pop_clip(fz_context *ctx, fz_device *dev);
```

Clipping groups can be nested, allowing complex graphical effects.

A related concept to clipping, is that of masking. Whereas clipping regions are simple on or off things, where content is chopped off at hard edges, masking allows for regions that allow just some proportion of the content to show through.

Masking operations take place in 2 stages; first the mask itself is defined, then the content to be masked.

Stage 1 begins by calling `fz_begin_mask` to start a mask definition group. Any series of graphical operations can now be sent to the device which will combine together to create the mask.

Stage 1 is terminated and Stage 2 begins by calling `fz_end_mask`. This converts the mask definition into a 'soft clip'. Any series of graphical operations can now be sent to the device which will combine together to create the mask contents.

The whole process is then completed by calling `fz_pop_clip`. This renders the mask contents through the soft clip, giving the final results.

```
void fz_begin_mask(fz_context *ctx, fz_device *dev, const fz_rect *area,
    int luminosity, fz_colorspace *colorspace, const float *bc);
void fz_end_mask(fz_context *ctx, fz_device *dev);
```

7.2.6 Groups and Transparency

Some document formats (such as PDF) offer a rich transparency model that allows graphical objects to be 'Grouped' together and imposed upon the page as if they have a given opacity, using a variety of different blend modes.

MuPDF implements this by using the `fz_begin_group` and `fz_end_group` calls.

```
void fz_begin_group(fz_context *ctx, fz_device *dev, const fz_rect
    *area, int isolated, int knockout, int blendmode, float alpha);
void fz_end_group(fz_context *ctx, fz_device *dev);
```

The exact details of PDF transparency are too complex to explain here; for a full explanation see The PDF Reference Manual.

7.2.7 Tiling

Many document formats allow for content to be tiled repeatedly. Frequently this is used to implement patterns for filling other graphical operations.

MuPDF implements this by allowing a group of content to be defined that is then tiled repeatedly across an area.

The content definition begins by calling `fz_begin_tile_id`, giving the area of the page to be filled (`area`), the area of a single tile (`view`), the x and y steps between repeats of the tile (`xstep` and `ystep`), the transformation to take all of these measurements out of pattern space to device space (`ctm`) and an integer `id`.

The purpose of `id` is to allow for efficient caching of rendered tiles. If `id` is 0, then no caching is performed. If it is non-zero, then it assumed to uniquely identify this tile. The tile can be safely placed into the Store (see chapter 5 Reference Counting, Memory Management and The Store) and future uses of this tile can short circuit the tile definition/rendering phase.

If a tile is found in the store, then `fz_begin_tile_id` will return non-zero and the caller can proceed directly to the call to `fz_end_tile`.

Any graphical operations sent to the device will be taken as part of the tile content, until `fz_end_tile` is called, whereupon these graphical operations will be imposed upon the output.

For the convenience of the caller, if no id is available (and hence no caching is possible), the `fz_begin_tile` variant can be used instead.

```
void fz_begin_tile(fz_context *ctx, fz_device *dev, const fz_rect *area,
    const fz_rect *view, float xstep, float ystep, const fz_matrix
    *ctm);
int fz_begin_tile_id(fz_context *ctx, fz_device *dev, const fz_rect
    *area, const fz_rect *view, float xstep, float ystep, const
    fz_matrix *ctm, int id);
void fz_end_tile(fz_context *ctx, fz_device *dev);
```

7.2.8 Render Flags

Every device has a set of render flags (a simple int, in which bits can be set or cleared).

These flags tend to have meanings specific to individual devices. In an ideal world they would not be required, but having this mechanism here can provide noticeable quality improvements.

```
void fz_render_flags(fz_context *ctx, fz_device *dev, int set, int
    clear);
```

The function basically does:

```
flags = (flags | set) & ~clear;
```

That is to say, the bits given in `set` are set, and then the bits given in `clear` are cleared.

The current only documented use of this is for the GProof device to request the Draw device to grid fit its tiled images.

The reason for using Render Flags rather than Device Hints (see section 7.4 Device Hints) is that Render Flags can be carried forward though display lists.

7.3 Cookie

The cookie is a lightweight mechanism for controlling and detecting the behaviour of a given interpretation call (i.e. `fz_run_page`, `fz_run_page_contents`, `fz_run_annot`, `fz_run_display_list` etc).

To use the cookie, a caller should simply define:

```
fz_cookie *cookie = { 0 };
```

set any required fields, for example:

```
cookie.incomplete_ok = 1;
```

and then pass `&cookie` as the last parameter to the interpretation call, for example:

```
fz_run_page(ctx, page, dev, transform, &cookie);
```

The contents and definition of `fz_cookie` are even more subject to change than other structures, so it is important to always initialise all the subfields to zero. The safest way to do this is as given above. If new fields are added to the structure, callers code should not need to change, and the default behaviour of zero-valued new fields will always remain the same.

7.3.1 Detecting errors

When displaying a page, if we hit an error, what should we do?

We could choose to stop interpretation entirely, but that would mean that a relatively unimportant error (such as a missing or broken font) would prevent us getting anything useful out of a page.

We could choose to ignore the errors and continue, but that would be a problem for (say) a print run, where it would undesirable for us to print 1000 copies of a document only to discover that it's missing an image.

The strategy taken by MuPDF is to swallow errors during interpretation, but keep a count of them in the `errors` field within the cookie. That way callers can check that `cookie.errors == 0` at the end to know whether a run completed without incident.

7.3.2 Using the cookie with threads

Content interpretations can take a (relatively) long time. Once one has been started, it can be useful a) to know how far through processing we are, and b) to be able to abort processing should the results of a run no longer be required.

As a run progresses, 2 fields in the cookie are updated. Firstly, `progress` will be set to a number that increases as progress is made. Think of this informally as being the number of objects that have been processed so far. In some cases (notably when processing a display list) we can know an upper bound for this value, and this value will be given as `progress_max`. In cases where no upper bound is known, `progress_max` will be set to -1. It is possible that the upper bound may start as -1, and then change to a known value later.

These values are intended to enable user feedback to be given, and should not be taken as guarantees of performance.

While running content, the interpreter periodically checks the abort field of the cookie. If it is discovered to be non zero, the rest of the content is ignored.

If the caller decides that it does not need the results of a run once it has been started (perhaps the user changes the page, or closes the file), then it should therefore set the `abort` field of the cookie to 1.

No guarantees are made about how often the cookie is checked, nor about how fast an interpreter will respond to the abort field once it is set. Setting the abort flag will never hurt, and will frequently help, however. Once the flag has been set to 1, it should never be reset to 0, as the results will be unpredictable.

Resources used by a run cannot be released until the end of a run, regardless of the setting of `abort`. Callers still need to wait for the `fz_run_page` (or other) call to complete before the page etc can be safely dropped.

7.3.3 Using the cookie to control partial rendering

The cookie also has a role to play when working in Progressive Mode. The `incomplete_ok` and `incomplete` fields are used for this. See chapter 16 Progressive Mode) for more details.

7.4 Device Hints

Device Hints are a mechanism that enables control over the behaviour of a device, and to interpreters calling to that device. Informally they offer hints about what a device is going to do and therefore what callers need to worry about.

Device hints take the form of bits in an int that can be enabled (set) or disabled (cleared). Callers can query these hints to customise their behaviour.

```
/*
    fz_enable_device_hints : Enable hints in a device.

    hints: mask of hints to enable.

    For example: By default the draw device renders shadings. For some
    purposes (perhaps rendering fast low quality thumbnails) you may want
    to tell it to ignore shadings. For this you would enable the
    FZ_IGNORE_SHADE hint.
*/
void fz_enable_device_hints(fz_context *ctx, fz_device *dev, int hints);

/*
    fz_disable_device_hints : Disable hints in a device.

    hints: mask of hints to disable.

    For example: By default the text extraction device ignores images.
    For some purposes however (such as extracting HTML) you may want to
    enable the capturing of image data too. For this you would disable
    the FZ_IGNORE_IMAGE hint.
*/
void fz_disable_device_hints(fz_context *ctx, fz_device *dev, int hints);
```

Some devices set the hints to non-zero default values.

For example, when running a text-extraction operation (as used to implement text search), there is little point in handling images, or shadings. The text extraction device therefore sets `FZ_IGNORE_IMAGE` and `FZ_IGNORE_SHADE`. The interpretation functions (such as `fz_run_page` or `fz_run_display_list` can then not bother to prepare images for calling into the device, improving performance.

If, however, you wish to extract the page content to an html file, you might want to include images in this output. So for this, you would disable the `FZ_IGNORE_IMAGE` hint before running the extraction, and the text extraction device would know to include them in its output structures.

The set of hints is subject to expansion in future, but is currently defined to be:

```
enum
{
    /* Hints */
    FZ_IGNORE_IMAGE = 1,
    FZ_IGNORE_SHADE = 2,
    FZ_DONT_INTERPOLATE_IMAGES = 4,
    FZ_MAINTAIN_CONTAINER_STACK = 8,
    FZ_NO_CACHE = 16,
};
```

`FZ_IGNORE_IMAGE` being enabled implies that a device will (or should) make no effort to handle images. For example, when playing back a display list (with `fz_run_display_list`), if the target device sets `FZ_IGNORE_IMAGE`, no image related calls will be made.

`FZ_IGNORE_SHADE` being enabled implies that a device will (or should) make no effort to handle shadings. For instance, if you are doing a quick pass across files trying to generate low quality thumbnail images, you may choose to disable shadings for speed.

`FZ_DONT_INTERPOLAGE_IMAGES` being enabled prevents the draw device performing interpolation. MuTool Draw uses this to inhibit interpolation when antialiasing is disabled. Finer control over this can now be given using the Tuning Context (see section 3.7 Tuning).

`FZ_MAINTAIN_CONTAINER_STACK` being enabled helps devices by causing MuPDF to maintain a stack of containers. This effectively moves some logic that would have to be in several devices into a place where it can be easily reused. Currently the only device that makes use of this is the SVG device, but it is hoped that more will use it in future.

`FZ_NO_CACHE` being enabled tells the interpreter to try to avoid caching any objects after the end of the content run. This can be used, for example, when searching a PDF for a text string to avoid pulling all the images, shadings, fonts etc and other resources for pages into memory at the expense of those that are used on the current page.

7.5 Inbuilt Devices

MuPDF comes with a selection of devices built in, though this should not be taken as a definitive list. It is expected that other devices will be written to extend MuPDF - indeed some embeddings of MuPDF already include their own devices.

7.5.1 BBox Device

The BBox device is a simple device that calculates the bbox of all the marking operations on a page.

```
/*
   fz_new_bbox_device: Create a device to compute the bounding
   box of all marks on a page.

   The returned bounding box will be the union of all bounding
   boxes of all objects on a page.
*/
fz_device *fz_new_bbox_device(fz_context *ctx, fz_rect *rectp);
```

The `fz_rect` passed to the `fz_new_bbox_device` must obviously stay in scope for the duration of the life of the device as it will be updated on exit with the bounding box for the contents.

7.5.2 Draw Device

The Draw device is the core renderer for MuPDF. Every draw device instance is constructed with a destination Pixmap (see section 8.3 Pixmaps for more details), and each graphical object passed to the device is rendered into that pixmap.

```
/*
   fz_new_draw_device: Create a device to draw on a pixmap.

   dest: Target pixmap for the draw device. See fz_new_pixmap*
   for how to obtain a pixmap. The pixmap is not cleared by the
   draw device, see fz_clear_pixmap* for how to clear it prior to
   calling fz_new_draw_device. Free the device by calling
   fz_drop_device.
*/
fz_device *fz_new_draw_device(fz_context *ctx, fz_pixmap *dest);
```

Most of the time we render complete pixmaps, but a mechanism exists to allow us to render a given bbox within a pixmap:

```
/*
   fz_new_draw_device_with_bbox: Create a device to draw on a pixmap.

   dest: Target pixmap for the draw device. See fz_new_pixmap*
   for how to obtain a pixmap. The pixmap is not cleared by the
   draw device, see fz_clear_pixmap* for how to clear it prior to
   calling fz_new_draw_device. Free the device by calling
   fz_drop_device.

   clip: Bounding box to restrict any marking operations of the
   draw device.
*/
fz_device *fz_new_draw_device_with_bbox(fz_context *ctx, fz_pixmap
    *dest, const fz_irect *clip);
```

This can be useful for updating particular areas of a page (for instance when an annotation has been edited or moved) without redrawing the whole thing.

During the course of rendering, the draw device may create new temporary internal pixmaps to cope with transparency and grouping. This is invisible to the caller, and can safely be considered an implementation detail, but should be considered when estimating the memory use for a given rendering operation. The exact number and size of internal pixmaps required depends on the exact complexity and makeup of the graphical objects being displayed.

To limit memory use, a typical strategy is to render pages in bands; rather than creating a single pixmap the size of the page and rendering that, create pixmaps for 'slices' across the page, and render them one at a time. The memory savings are not just seen in the cost of the basic pixmap, but also serve to limit the sizes of the internal pixmaps used during rendering.

The cost for this is that the page contents do need to be run through repeatedly. This can be achieved by reinterpreting directly from the file, but that can be expensive. The next device provides a route to help with this.

7.5.3 Display List Device

The Display list device simply records all the calls made to it in a list. This list can then be played back later, potentially multiple times and with different transforms, to other devices.

```
/*
   fz_new_list_device: Create a rendering device for a display list.

   When the device is rendering a page it will populate the
   display list with drawing commsnds (text, images, etc.). The
   display list can later be reused to render a page many times
   without having to re-interpret the page from the document file
```

```
    for each rendering. Once the device is no longer needed, free
    it with fz_drop_device.

    list: A display list that the list device takes ownership of.
*/
fz_device *fz_new_list_device(fz_context *ctx, fz_display_list *list);
```

For more details of the uses of Display Lists, see chapter 9 Display Lists.

7.5.4 PDF Output Device

The PDF Output device is still a work in progress, as its handling of fonts is incomplete. Nonetheless for certain classes of files it can be useful.

End users will probably prefer to use the document writer interface (see /r-jwrefDocumentWriter) which wraps this class up, rather than call it directly. Nonetheless this can be useful in specific circumstances when generating particular sections of a PDF file (such as appearance streams for annotations).

The PDF Output device takes the sequence of graphical operations it is called with, and forms it back into a sequence of PDF operations, together with a set of required resources. These can then be formed into a completely new PDF page (or a PDF annotation) which can then be inserted into a document.

```
/*
    pdf_page_write: Create a device that will record the
    graphical operations given to it into a sequence of
    pdf operations, together with a set of resources. This
    sequence/set pair can then be used as the basis for
    adding a page to the document (see pdf_add_page).

    doc: The document for which these are intended.

    mediabox: The bbox for the created page.

    presources: Pointer to a place to put the created
    resources dictionary.

    pcontents: Pointer to a place to put the created
    contents buffer.
*/
fz_device *pdf_page_write(fz_context *ctx, pdf_document *doc, const
    fz_rect *mediabox, pdf_obj **presources, fz_buffer **pcontents);
```

7.5.5 Structured Text Device

The Structured Text device is used to extract the text from a given graphical stream, together with the position it inhabits on the output page. It can also optionally include details of images and their positions within its output.

```
/*
   fz_new_stext_device: Create a device to extract the text on a page.

   Gather and sort the text on a page into spans of uniform style,
   arranged into lines and blocks by reading order. The reading order
   is determined by various heuristics, so may not be accurate.

   sheet: The text sheet to which styles should be added. This can
   either be a newly created (empty) text sheet, or one containing
   styles from a previous text device. The same sheet cannot be used
   in multiple threads simultaneously.

   page: The text page to which content should be added. This will
   usually be a newly created (empty) text page, but it can be one
   containing data already (for example when merging multiple pages, or
   watermarking).
*/
fz_device *fz_new_stext_device(fz_context *ctx, fz_stext_sheet *sheet,
    fz_stext_page *page);
```

This can be used as the basis for searching (including highlighting the text as matches are found), for exporting text files (or text and image based files such as HTML), or even to do more complex page analysis (such as spotting what regions of the page are text, what are graphics etc).

An (initially empty) `fz_stext_sheet` should be created using `fz_new_stext_sheet`, and an empty `fz_stext_page` created using `fz_new_stext_page`. These are used in the call to `fz_new_stext_device`. After the contents have been run to that device, the sheet will be populated with the common styles used by the page, and the page will be populated with details of the text extracted and its position.

7.5.6 SVG Output Device

The SVG output device is used to generate SVG pages from arbitrary input.

End users will probably prefer to use the document writer interface (see /rjwref-DocumentWriter) which wraps this class up, rather than call it directly.

```
/*
   fz_new_svg_device: Create a device that outputs (single page)
   SVG files to the given output stream.
```

```
	output: The output stream to send the constructed SVG page
	to.

	page_width, page_height: The page dimensions to use (in
	points).
*/
fz_device *fz_new_svg_device(fz_context *ctx, fz_output *out, float
	 page_width, float page_height);
```

The device currently generates SVG 1.1 compliant files. SVG Fonts are NOT used due to poor client support. Instead glyphs are sent as reusable symbols. Shadings are sent as rasterised images. JPEGs will be passed through unchanged, and all other images will be converted to PNG.

7.5.7 Test Device

The Test device, as its name suggests, tests a given set of page contents for which features are used. Currently this is restricted to testing for whether the graphical objects used are greyscale or colour. Testing for additional features may be added in future.

```
/*
	fz_new_test_device: Create a device to test for features.

	Currently only tests for the presence of non-grayscale colors.

	is_color: Possible values returned:
		0: Definitely greyscale
		1: Probably color (all colors were grey, but there
		were images or shadings in a non grey colorspace).
		2: Definitely color

	threshold: The difference from grayscale that will be tolerated.
	Typical values to use are either 0 (be exact) and 0.02 (allow an
	imperceptible amount of slop).

	options: A set of bitfield options, from the FZ_TEST_OPT set.

	passthrough: A device to pass all calls through to, or NULL.
	If set, then the test device can both test and pass through to
	an underlying device (like, say, the display list device). This
	means that a display list can be created and at the end we'll
	know if its color or not.

	In the absence of a passthrough device, the device will throw
	an exception to stop page interpretation when color is found.
*/
```

```
fz_device *fz_new_test_device(fz_context *ctx, int *is_color, float
    threshold, int options, fz_device *passthrough);
```

The expected purpose of the colour detecting functionality is to allow applications (e.g. printers) to easily detect if a given page requires the use of colour inks, or whether a greyscale rendering will suffice.

This device can either be used by itself, or in the form of a pass-through device.

Standalone use

In the simplest form, the device can be created standalone, by passing `passthrough` as NULL.

As each subsequent device call is made, the device will test the graphic object passed to it to see if it is within the given `threshold` of being a neutral colour. If it is, then the device continues. If not, then it sets the int pointed to by `is_color` to be non zero.

For graphical objects such as paths or text, this is an easy evaluation that takes almost no time. For Images or Shadings however, it is slightly trickier. An image may be defined in a colour space capable of non-neutral colours (perhaps RGB or CMYK) and yet the image itself may only use neutral colours within that space. To properly establish whether colours are required or not, requires much more CPU intensive processing.

Accordingly, the device will, by default, just look at the colour space. The value of `is_color` returned at the end may be examined to establish the confidence level of the test. 0 means "definitely greyscale", 1 means "probably colour" (i.e. "an image or shading was seen that potentially contains non neutral colours"), and 2 means "definitely colour".

If the caller wishes to spend the CPU cycles to get a definite answer, `options` can be set to `FZ_TEXT_OPT_IMAGES | FZ_TEXT_OPT_SHADINGS` and images and shadings will be exhaustively checked.

As an optimisation, given how much faster is is to check non-images and shadings, it can be worth running the device once without the options set, and then only running it again with them set if required.

If the device is run with `passthrough` as NULL, then as soon as it encounters a "definite" non-neutral colour, it will throw a `FZ_ABORT` error.

Passthrough use

As discussed above, the envisaged use case for this device is to detect whether page contents require colour or not to allow printers to decide whether to rasterise for colour inks or a faster/cheaper greyscale pass.

Such printers will normally be operating in banded mode, which requires (or at least greatly benefits from) the use of a display list. By using the device in `passthrough` mode, the testing can be performed at the same time as the list is built.

Simply create the display list device as you would normally, and pass it into `fz_new_test_device` as `passthrough`. Then run the page contents through the returned test device. The test device will pass each call through to the underlying list device and so the display list be built as normal.

When run in this mode, the device can no longer use the 'early-exit' optimisation of throwing a **FZ_ABORT** error.

7.5.8 Trace Device

The Trace device is a simple debugging device that allows an XML-like representation of the device calls made to be output.

```
/*
   fz_new_trace_device: Create a device to print a debug trace of all
       device calls.
*/
fz_device *fz_new_trace_device(fz_context *ctx, fz_output *out);
```

This is a useful tool to visualise the contents of display lists.

Chapter 8

Building Blocks

8.1 Overview

MuPDF uses many constructs and concepts that, while not deserving of chapters in their own rights, do deserve mention.

8.2 Colorspaces

In order to represent a given color for a graphical object, we need both the color component values and details of the colorspace that the color is specified in. Color values are defined simply as floats (between 0 and 1 inclusive), and colorspaces are defined using the `fz_colorspace` structure.

As with many other such structures in MuPDF, these are reference counted objects (see section 5.2 Reference Counting).

8.2.1 Basic Colorspaces

MuPDF contains a set of inbuilt colorspaces that cover most simple requirements. These can

8.3 Pixmaps

8.3.1 Overview

The `fz_pixmap` structure is used to represent a 2 dimensional array of contone pixels. This is used throughout MuPDF, as the target of rendering from the draw device, as internal buffers during processing, and during image decoding.

A pixmap can have an arbitrary number of colour components, together with an optional alpha plane. Every component sample is represented by an unsigned char.

The number of colour components corresponds to the colour space of a pixmap; pixmaps without a colour space must contain no colour planes, just a single alpha plane.

The data within a pixmap is always stored packed in 'chunky' format. For instance, an RGB pixmap would have data in the form: RGBRGBRGBRGB...

Alpha data is always sent as the last byte in the set corresponding to a pixel. An RGB pixmap with an alpha plane would be therefore have data of the form: RGBARGBARGBA...

To allow pixmaps to sensibly 'subset' one another, pixmaps have a 'stride' field. This gives the number of bytes difference from the address of the start of the representation of a pixel to the address of the start of the representation of the same pixel on the scanline below.

Normally you'd expect stride to be the same as width multiplied by the number of components in the image (including alpha), but for 'sub areas' of larger pixmaps this can be much larger.

Pixmaps can frequently map onto operating system specific bitmap representations, but these sometimes require each scanline to be word aligned - again the provision of stride allows for this. Bottom up bitmaps can be implemented using a negative stride.

8.3.2 Saving

Various functions exist to either save pixmaps to files, or to (more generally) write them to `fz_output` streams.

In general, these functions fall into 3 categories of increasing complexity.

Firstly, functions of the form `fz_save_pixmap_to_`.... take a pixmap and write it to a given filename in the local filesystem.

Secondly, functions of the form `fz_write_pixmap_to_`.... take a pixmap and write it to a given `fz_output`. The use of an `fz_output` allows for writing to memory buffers, or even potentially to encrypt or compress further as the write progresses.

Finally, we have an `fz_band_writer` class, that allows images to be written to file in 'bands', thus minimising the amount of memory required at any one time. Typically a band writer is created using a call such as `fz_new_png_band_writer`:

```
/*
    fz_new_png_band_writer: Obtain a fz_band_writer instance
    for producing PNG output.
*/
fz_band_writer *fz_new_png_band_writer(fz_context *ctx, fz_output *out);
```

First, an image header is written, using this writer:

```
/*
    fz_write_header: Cause a band writer to write the header for
    a banded image with the given properties/dimensions etc. This
    also configures the bandwriter for the format of the data to be
    passed in future calls.

    w, h: Width and Height of the entire page.

    n: Number of components (including alphas).

    alpha: Number of alpha components.

    xres, yres: X and Y resolutions in dpi.

    pagenum: Page number

    Throws exception if incompatible data format.
*/
void fz_write_header(fz_context *ctx, fz_band_writer *writer, int w, int
     h, int n, int alpha, int xres, int yres, int pagenum);
```

This has the effect of setting the size and format of the data for the complete image. The caller then proceeds to render the page in horizontal strips from the top to the bottom, and pass them in to `fz_write_band`:

```
/*
    fz_write_band: Cause a band writer to write the next band
    of data for an image.

    stride: The byte offset from the first byte of the data
    for a pixel to the first byte of the data for the same pixel
    on the row below.

    band_height: The number of lines in this band.
    samples: Pointer to first byte of the data.
*/
void fz_write_band(fz_context *ctx, fz_band_writer *writer, int stride,
    int band_height, const unsigned char *samples);
```

The band writer keeps track of how much data has been written, and when an entire page has been sent, it writes out any image trailer required.

For formats that can accommodate multiple pages, a new call to `fz_write_header` will start the process again. Otherwise (or after the final image), the band writer can be neatly discarded by calling:

```
void fz_drop_band_writer(fz_context *ctx, fz_band_writer *writer);
```

8.4 Bitmaps

The `fz_bitmap` structure is used to represent a 2 dimensional array of monochrome pixels. They are the 1 bit per component equivalent of the `fz_pixmap` structure.

The core rendering engine of MuPDF does not currently make use of `fz_bitmaps`, but rather they are used as a step along the way for outputting rendered information.

Functions exist within MuPDF to create `fz_bitmaps` from `fz_pixmaps` by halftoning. See section 8.5 Halftones.

```
/*
    fz_new_bitmap_from_pixmap: Make a bitmap from a pixmap and a
        halftone.

    pix: The pixmap to generate from. Currently must be a single color
    component + alpha (where the alpha is assumed to be solid).

    ht: The halftone to use. NULL implies the default halftone.

    Returns the resultant bitmap. Throws exceptions in the case of
    failure to allocate.
*/
fz_bitmap *fz_new_bitmap_from_pixmap(fz_context *ctx, fz_pixmap *pix,
    fz_halftone *ht);

fz_bitmap *fz_new_bitmap_from_pixmap_band(fz_context *ctx, fz_pixmap
    *pix, fz_halftone *ht, int band_start, int bandheight);
```

Both functions work by applying a `fz_halftone` to the contone values to make the bitmap. The latter function is a more general version of the former, that allows for correct operation when rendering in bands - namely that the correct offset into the halftone table is used.

The data for each `Bitmap` is packed into bytes most significant bit first. Multiple components are packed into the same byte, so a CMYK pixmap converted to a bitmap would have 2 pixels worth of data in the first byte, CMYKCMYK, with the first pixel in the highest nibble.

The usual reference counting behaviour applies to `fz_bitmap`s, with `fz_keep_bitmap` and `fz_drop_bitmap` claiming and releasing references respectively.

8.5 Halftones

The `fz_halftone` structure represents a set of tiles, one per component, each of a potentially different size. Each of these tiles is a 2-dimensional array of threshold values (actually implemented as a single component `fz_pixmap`). During the halftoning (bitmap creation) process, if the contone value is smaller than the threshold value, then it remains unset in the output. If it is larger or equal then it is set in the output.

For convenience, a `NULL` pointer can be used to signify the default halftone. The default halftone can also be fetched by using:

```
/*
   fz_default_halftone: Create a 'default' halftone structure
   for the given number of components.

   num_comps: The number of components to use.

   Returns a simple default halftone. The default halftone uses
   the same halftone tile for each plane, which may not be ideal
   for all purposes.
*/
fz_halftone *fz_default_halftone(fz_context *ctx, int num_comps);
```

The creation of halftones is a specialised field upon which much research has been done. The mechanisms in MuPDF are designed to allow people the freedom to create and tune the halftones for their particular application.

The usual reference counting behaviour applies to `fz_halftone`s, with `fz_keep_halftone` and `fz_drop_halftone` claiming and releasing references respectively.

8.6 Images

The `fz_image` structure is used to represent a generic Image object in MuPDF. It can be viewed as an encapsulation from which both a rendering of an image (as an `fz_pixmap`) and (often) the original source data can be retrieved.

The primary use of an `fz_image` is to allow a rendered pixmap to be retrieved. This is done by calling:

```
/*
    fz_get_pixmap_from_image: Called to get a handle to a pixmap from
    an image.

    image: The image to retrieve a pixmap from.

    subarea: The subarea of the image that we actually care about (or
    NULL to indicate the whole image).

    trans: Optional, unless subarea is given. If given, then on entry
    this is the transform that will be applied to the complete image.
    It should be updated on exit to the transform to apply to the given
    subarea of the image. This is used to calculate the desired
    width/height for subsampling.

    w: If non-NULL, a pointer to an int to be updated on exit to the
    width (in pixels) that the scaled output will cover.

    h: If non-NULL, a pointer to an int to be updated on exit to the
    height (in pixels) that the scaled output will cover.

    Returns a non NULL pixmap pointer. May throw exceptions.
*/
fz_pixmap *fz_get_pixmap_from_image(fz_context *ctx, fz_image *image,
     const fz_irect *subarea, fz_matrix *trans, int *w, int *h);
```

Frequently this will involve decoding the image from its source data, so should be considered a potentially expensive call, both in terms of CPU time, and memory usage.

To minimise the impact of such decodes, `fz_image`s make use of the Store (see chapter 5 Reference Counting, Memory Management and The Store) to cache decoded versions in. This means that (subject to enough memory being available) repeated calls to get a `fz_pixmap` from the same `fz_image` (with the same parameters) will return the same `fz_pixmap` each time, with no further decode being required.

The usual reference counting behaviour applies to `fz_image`s, with `fz_keep_image` and `fz_drop_image` claiming and releasing references respectively.

Depending on the size at which an fz_image is to be used, it may not be worth decoding it at full resolution; instead, decoding it at a smaller size can save memory (and frequently time). In addition, subsequent rendering operations can often be faster due to having to handle fewer pixels for no quality loss in the final output.

To facilitate this, fz_images will subsample images as appropriate. Subsampling involves an image being decoded to a size an integer power of 2 smaller than their native size. For instance, if an image has a native size of 400x300, and is to be rendered to a final size of 40x30, fz_get_pixmap_from_image may subsample the returned image by up to 8 in each direction, resulting in a 50x37 image.

Subsequent operations (such as smooth scaling and rendering) will proceed much faster due to fewer pixels being involved, and around one sixteenth of the memory will be required.

Various different implementations of fz_image exist within MuPDF.

8.6.1 Compressed Images

The fz_compressed_image structure is a specialisation of fz_image, that holds the source data for an image in an fz_compressed_buffer. This is the usual form for images created from PDF and XPS files.

The data for a compressed image can be retrieved by calling:

```
fz_compressed_buffer *fz_compressed_image_buffer(fz_context *ctx,
    fz_image *image);
```

If the supplied fz_image is not an fz_compressed_image then it will return NULL.

8.6.2 Pixmap Images

The fz_pixmap_image structure is a specialisation of fz_image, that has an fz_pixmap as its source data. This exists to allow fz_pixmaps from other sources to be easily fed into the MuPDF rendering engine.

8.7 Buffers

The fz_buffer structure is used to represent arbitrary buffers of data. Essentially they are a representation for arbitrary blocks of bytes (in whatever encoding required), with simple functions for extending, concatenating, and writing in byte, char, utf8 and bitwise fashion.

Both the internals and API level functions of MuPDF use `fz_buffer`s extensively.

The usual reference counting behaviour applies to `fz_buffer`s, with `fz_keep_buffer` and `fz_drop_buffer` claiming and releasing references respectively.

8.8 Transforms

The `fz_matrix` structure is used to represent 2 dimensional matrices used for transforming points, shapes and other geometry.

The six fields of the `fz_matrix` structure correspond to a matrix of the form:

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$$

Such transformation matrices can be used to represent a wide range of different operations, including translations, rotations, scales, sheers, and any combination thereof.

Typically, a matrix will be created for a specific purpose, such as a scale, or a translation. For this reason, we have dedicated construction calls.

```
/*
    fz_scale: Create a scaling matrix.

    The returned matrix is of the form [ sx 0 0 sy 0 0 ].

    m: Pointer to the matrix to populate

    sx, sy: Scaling factors along the X- and Y-axes. A scaling
    factor of 1.0 will not cause any scaling along the relevant
    axis.

    Returns m.

    Does not throw exceptions.
*/
fz_matrix *fz_scale(fz_matrix *m, float sx, float sy);

/*
    fz_shear: Create a shearing matrix.

    The returned matrix is of the form [ 1 sy sx 1 0 0 ].
    m: pointer to place to store returned matrix

    sx, sy: Shearing factors. A shearing factor of 0.0 will not
    cause any shearing along the relevant axis.

    Returns m.

    Does not throw exceptions.
*/
fz_matrix *fz_shear(fz_matrix *m, float sx, float sy);

/*
    fz_rotate: Create a rotation matrix.

    The returned matrix is of the form
    [ cos(deg) sin(deg) -sin(deg) cos(deg) 0 0 ].

    m: Pointer to place to store matrix

    degrees: Degrees of counter clockwise rotation. Values less
    than zero and greater than 360 are handled as expected.

    Returns m.

    Does not throw exceptions.
*/
fz_matrix *fz_rotate(fz_matrix *m, float degrees);

/*
    fz_translate: Create a translation matrix.

    The returned matrix is of the form [ 1 0 0 1 tx ty ].

    m: A place to store the created matrix.

    tx, ty: Translation distances along the X- and Y-axes. A
    translation of 0 will not cause any translation along the
    relevant axis.

    Returns m.

    Does not throw exceptions.
*/
fz_matrix *fz_translate(fz_matrix *m, float tx, float ty);
```

Mathematically, points are transformed by multiplying them (extended to 3 elements long). For example (`x'`,`y'`), the point given by mapping (x,y) through such a matrix is calculated as follows:

$$\begin{pmatrix} x' & y' & 1 \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$$

There are various functions in MuPDF to perform such transformations:

```
/*
   fz_transform_point: Apply a transformation to a point.

   transform: Transformation matrix to apply. See fz_concat,
   fz_scale, fz_rotate and fz_translate for how to create a
   matrix.

   point: Pointer to point to update.

   Returns transform (unchanged).

   Does not throw exceptions.
*/
fz_point *fz_transform_point(fz_point *restrict point, const fz_matrix
    *restrict transform);
fz_point *fz_transform_point_xy(fz_point *restrict point, const
    fz_matrix *restrict transform, float x, float y);
```

Rectangles can be transformed using the following function, which allows for the fact that the image of a rectangle may be flipped:

```
/*
   fz_transform_rect: Apply a transform to a rectangle.

   After the four corner points of the axis-aligned rectangle
   have been transformed it may not longer be axis-aligned. So a
   new axis-aligned rectangle is created covering at least the
   area of the transformed rectangle.

   transform: Transformation matrix to apply. See fz_concat,
   fz_scale and fz_rotate for how to create a matrix.

   rect: Rectangle to be transformed. The two special cases
   fz_empty_rect and fz_infinite_rect, may be used but are
   returned unchanged as expected.

   Does not throw exceptions.
*/
fz_rect *fz_transform_rect(fz_rect *restrict rect, const fz_matrix
    *restrict transform);
```

Also, it can be useful to transform a point, ignoring the translation components of a transformation, so we have a convenience function for this:

```
/*
   fz_transform_vector: Apply a transformation to a vector.

   transform: Transformation matrix to apply. See fz_concat,
   fz_scale and fz_rotate for how to create a matrix. Any
   translation will be ignored.

   vector: Pointer to vector to update.

   Does not throw exceptions.
*/
fz_point *fz_transform_vector(fz_point *restrict vector, const fz_matrix
    *restrict transform);
```

Transformations can be combined by multiplying their representative matrices together. Transforming a point by applying matrix A then matrix B, will give identical results to transforming the point by AB.

MuPDF provides an API for combining matrices in this way:

```
/*
   fz_concat: Multiply two matrices.

   The order of the two matrices are important since matrix
   multiplication is not commutative.

   Returns result.

   Does not throw exceptions.
*/
fz_matrix *fz_concat(fz_matrix *result, const fz_matrix *left, const
    fz_matrix *right);
```

Alternatively, operations can be specifically applied to existing matrices. Because of the non-commutative nature of matrix operations, it matters whether the new operation is applied before or after the existing matrix.

For example, if you have a matrix that performs a rotation, and you wish to combine that with a translation, you must decide whether you want the translation to occur before the rotation ('pre') or afterwards ('post').

MuPDF has various API functions for such operations:

```
/*
   fz_pre_scale: Scale a matrix by premultiplication.

   m: Pointer to the matrix to scale
```

```
    sx, sy: Scaling factors along the X- and Y-axes. A scaling
    factor of 1.0 will not cause any scaling along the relevant
    axis.

    Returns m (updated).

    Does not throw exceptions.
*/
fz_matrix *fz_pre_scale(fz_matrix *m, float sx, float sy);

/*
    fz_post_scale: Scale a matrix by postmultiplication.

    m: Pointer to the matrix to scale

    sx, sy: Scaling factors along the X- and Y-axes. A scaling
    factor of 1.0 will not cause any scaling along the relevant
    axis.

    Returns m (updated).

    Does not throw exceptions.
*/
fz_matrix *fz_post_scale(fz_matrix *m, float sx, float sy);

/*
    fz_pre_shear: Premultiply a matrix with a shearing matrix.

    The shearing matrix is of the form [ 1 sy sx 1 0 0 ].

    m: pointer to matrix to premultiply

    sx, sy: Shearing factors. A shearing factor of 0.0 will not
    cause any shearing along the relevant axis.

    Returns m (updated).

    Does not throw exceptions.
*/
fz_matrix *fz_pre_shear(fz_matrix *m, float sx, float sy);

/*
    fz_pre_rotate: Rotate a transformation by premultiplying.

    The premultiplied matrix is of the form
    [ cos(deg) sin(deg) -sin(deg) cos(deg) 0 0 ].

    m: Pointer to matrix to premultiply.
    degrees: Degrees of counter clockwise rotation. Values less
    than zero and greater than 360 are handled as expected.

    Returns m (updated).

    Does not throw exceptions.
*/
fz_matrix *fz_pre_rotate(fz_matrix *m, float degrees);

/*
    fz_pre_translate: Translate a matrix by premultiplication.

    m: The matrix to translate

    tx, ty: Translation distances along the X- and Y-axes. A
    translation of 0 will not cause any translation along the
    relevant axis.

    Returns m.

    Does not throw exceptions.
*/
fz_matrix *fz_pre_translate(fz_matrix *m, float tx, float ty);
```

Finally, sometimes it is useful to find the matrix that would represent the reverse of a given transformation. This can be achieved by 'inverting' the matrix. This is not possible in all cases, but can be achieved for most 'well-behaved' transformations.

```
/*
    fz_invert_matrix: Create an inverse matrix.

    inverse: Place to store inverse matrix.

    matrix: Matrix to invert. A degenerate matrix, where the
    determinant is equal to zero, can not be inverted and the
    original matrix is returned instead.

    Returns inverse.

    Does not throw exceptions.
*/
fz_matrix *fz_invert_matrix(fz_matrix *inverse, const fz_matrix *matrix);

/*
    fz_try_invert_matrix: Attempt to create an inverse matrix.

    inverse: Place to store inverse matrix.
    matrix: Matrix to invert. A degenerate matrix, where the
    determinant is equal to zero, can not be inverted.

    Returns 1 if matrix is degenerate (singular), or 0 otherwise.

    Does not throw exceptions.
*/
 int fz_try_invert_matrix(fz_matrix *inverse, const fz_matrix *matrix);
```

8.9 Paths

Postscript (or PDF) style paths are represented using the `fz_path` structure. A postscript path consists of a sequence of instructions describing the movement of a 'pen' around a given path.

The first instruction is always a 'move' to a specified location. Subsequent instructions move the pen position onwards to new positions on the page, either via straight lines, or via curves described by given control points. Such instructions can either be made with the pen up or down.

Once created paths can then be rendered by MuPDF either by being filled, or by being stroked. The path itself has no knowledge of how it will be used - the details of the fill or the stroke attributes are supplied externally to this structure. A description of the exact rules used for filling and stroking are beyond the scope of this document. For more information see "The PDF Reference Manual" or "The Postscript Language Reference Manual".

Paths are reference counted objects, with the implicit understanding that once more than one reference exists to a path, it will no longer be modified.

8.9.1 Creation

A reference to a new empty path can be created using `fz_new_path`:

```
/*
    fz_new_path: Create an empty path, and return
    a reference to it.

    Throws exception on failure to allocate.
*/
fz_path *fz_new_path(fz_context *ctx);
```

Once a path exists, commands can be added to it. The first command must always be a 'move'.

```
/*
    fz_moveto: Append a 'moveto' command to a path.

    path: The path to modify.

    x, y: The coordinate to move to.

    Throws exceptions on failure to allocate.
*/
void fz_moveto(fz_context *ctx, fz_path *path, float x, float y);
```

Once we have moved to a point, subsequent commands can be added, such as lines, quads (quadratic beziers) and curves (cubic beziers).

```
/*
    fz_lineto: Append a 'lineto' command to a path.

    path: The path to modify.

    x, y: The coordinate to line to.

    Throws exceptions on failure to allocate.
*/
void fz_lineto(fz_context *ctx, fz_path *path, float x, float y);

/*
    fz_quadto: Append a 'quadto' command to a path. (For a
    quadratic bezier).

    path: The path to modify.

    x0, y0: The control coordinates for the quadratic curve.

    x1, y1: The end coordinates for the quadratic curve.

    Throws exceptions on failure to allocate.
*/
void fz_quadto(fz_context *ctx, fz_path *path, float x0, float y0, float
     x1, float y1);

/*
    fz_curveto: Append a 'curveto' command to a path. (For a
    cubic bezier).

    path: The path to modify.

    x0, y0: The coordinates of the first control point for the
    curve.
    x1, y1: The coordinates of the second control point for the
    curve.

    x2, y2: The end coordinates for the curve.

    Throws exceptions on failure to allocate.
*/
void fz_curveto(fz_context *ctx, fz_path *path, float x0, float y0,
     float x1, float y1, float x2, float y2);
```

In addition, we have 2 functions for adding curves (cubic beziers) where one of the control points is coincident with the neighbouring endpoints. These functions mirror the usage in PDF, but offer no benefits other than convenience as such curves are detected automatically as part of an `fz_curveto` call.

```
/*
    fz_curvetov: Append a 'curvetov' command to a path. (For a
    cubic bezier with the first control coordinate equal to
    the start point).

    path: The path to modify.

    x1, y1: The coordinates of the second control point for the
    curve.

    x2, y2: The end coordinates for the curve.

    Throws exceptions on failure to allocate.
*/
void fz_curvetov(fz_context *ctx, fz_path *path, float x1, float y1,
     float x2, float y2);

/*
    fz_curvetoy: Append a 'curvetoy' command to a path. (For a
    cubic bezier with the second control coordinate equal to
    the end point).

    path: The path to modify.

    x0, y0: The coordinates of the first control point for the
    curve.

    x2, y2: The end coordinates for the curve (and the second
    control coordinate).

    Throws exceptions on failure to allocate.
*/
void fz_curvetoy(fz_context *ctx, fz_path *path, float x0, float y0,
     float x2, float y2);
```

At any point after the initial move, we can close the path using `fz_closepath`:

```
/*
   fz_closepath: Close the current subpath.

   path: The path to modify.

   Throws exceptions on failure to allocate, and illegal
   path closes.
*/
void fz_closepath(fz_context *ctx, fz_path *path);
```

After a path has been closed, the only acceptable next command is a move. A path need not be closed before a second or subsequent move is sent.

Finally, we have one additional path construction function, `fz_rectto`. This appends a rectangle to the current path. This rectangle is equivalent to a move, 3 lines and a closepath, and so is the one exception to the rule that paths must begin with a move (as one is implicit within the rectangle command).

```
/*
   fz_rectto: Append a 'rectto' command to a path.

   The rectangle is equivalent to:
      moveto x0 y0
      lineto x1 y0
      lineto x1 y1
      lineto x0 y1
      closepath

   path: The path to modify.

   x0, y0: First corner of the rectangle.

   x1, y1: Second corner of the rectangle.

   Throws exceptions on failure to allocate.
*/
void fz_rectto(fz_context *ctx, fz_path *path, float x0, float y0, float
    x1, float y1);
```

Finally, during path construction, the coordinate at which the notional path cursor has reached can be read using the `fz_currentpoint` function.

```
/*
   fz_currentpoint: Return the current point that a path has
   reached or (0,0) if empty.

   path: path to return the current point of.
```

```
*/
fz_point fz_currentpoint(fz_context *ctx, fz_path *path);
```

8.9.2 Reference counting

As stated before, fz_paths are reference counted objects. Once one has been created, references can be created/destroyed using the standard keep/drop conventions:

```
/*
   fz_keep_path: Take an additional reference to
   a path.

   No modifications should be carried out on a path
   to which more than one reference is held, as
   this can cause race conditions.

   Never throws exceptions.
*/
fz_path *fz_keep_path(fz_context *ctx, const fz_path *path);

/*
   fz_drop_path: Drop a reference to a path,
   destroying the path if it is the last
   reference.

   Never throws exceptions.
*/
void fz_drop_path(fz_context *ctx, const fz_path *path);
```

A path with more than one reference is considered to be ‘frozen’ or ‘immutable’. It is not safe to modify such a path, as the other holder of a reference to it may not expect it to be being changed. That is to say that modification operations on paths are not atomic between threads.

If you have a path that you wish to be able to modify, simply call fz_clone_path to obtain a reference to a copy of the path that is safe to modify:

```
/*
   fz_clone_path: Clone the data for a path.

   This is used in preference to fz_keep_path when a whole
   new copy of a path is required, rather than just a shared
   pointer. This probably indicates that the path is about to
   be modified.

   path: path to clone.
```

```
    Throws exceptions on failure to allocate.
*/
fz_path *fz_clone_path(fz_context *ctx, fz_path *path);
```

8.9.3 Storage

Because Paths are such a crucial part of MuPDF, and are used so widely in document content, we take particular care to allow them to be expressed and accessed efficiently.

This means that at path construction time, we spot simple cases where we can optimise the path representation. For example, a move immediately following a move can cause the first move to be dropped. Similarly, a curve with both control points coincident with the endpoints can be expressed as a line.

This means that if you read a path out after construction (see subsection 8.9.7 Walking) you cannot rely on the exact representation being the same.

In addition, after constructing a path, there are some simple things that can be done to minimise the memory used.

As paths are constructed, the data buffers within them grow. For efficiency, these grow with some slack in them, so at the end of construction there can be a non-trivial amount of space wasted.

If you intend to simply use the path, and then discard it, this does not matter. If instead you intend to keep the path around for a while, it may be worth calling `fz_trim_path` to shrink the storage buffers as much as possible.

```
/*
    fz_trim_path: Minimise the internal storage
    used by a path.

    As paths are constructed, the internal buffers
    grow. To avoid repeated reallocations they
    grow with some spare space. Once a path has
    been fully constructed, this call allows the
    excess space to be trimmed.

    Never throws exceptions.
*/
void fz_trim_path(fz_context *ctx, fz_path *path);
```

MuPDF automatically calls this function when `fz_keep_path` is called for the first time as having more than one reference to a path is considered a good indication of it being kept around for a while.

For cases where large numbers of paths are kept around for a long period of time, for example in a `fz_display_list` (see chapter 9 Display Lists), it can be advantageous to 'pack' paths to further minimise the space they use.

To pack a path, first call `fz_packed_path_size` to obtain the number of bytes required to pack a path:

```
/*
    fz_packed_path_size: Return the number of
    bytes required to pack a path.

    Never throws exceptions.
*/
int fz_packed_path_size(const fz_path *path);
```

Then, call `fz_pack_path` with some (suitably aligned) memory of the appropriate size to actually pack the path:

```
/*
    fz_pack_path: Pack a path into the given block.

    To minimise the size of paths, this function allows them to be
    packed into a buffer with other information.

    pack: Pointer to a block of data to pack the path into. Should
    be aligned by the caller to the same alignment as required for
    an fz_path pointer.

    max: The number of bytes available in the block.
    If max < sizeof(fz_path) then an exception will
    be thrown. If max >= the value returned by
    fz_packed_path_size, then this call will never
    fail, except in low memory situations with large
    paths.

    path: The path to pack.

    Paths can be 'unpacked', 'flat', or 'open'. Standard paths, as
    created are 'unpacked'. Paths that will pack into less than max
    bytes will be packed as 'flat', unless they are too large (where
    large indicates that they exceed some private implementation
    defined limits, currently including having more than 256
    256 coordinates or commands).

    Large paths are 'open' packed as a header into the given block,
    plus pointers to other data blocks. Paths can be used
    interchangably regardless of how they are packed.

    Returns the number of bytes within the block used. Callers can
    access the packed path data by casting the value of pack on
    entry to be an fz_path *.

    Throws exceptions on failure to allocate, or if
    max < sizeof(fz_path).
*/
int fz_pack_path(fz_context *ctx, uint8_t *pack, int max, const fz_path
     *path);
```

After a successful call to `fz_pack_path`, the pointer to the block of memory can be cast to an `fz_path *` and used as normal.

All the path routines recognise packed paths and will use them interchangeably. Packed paths may not be modified once created, however.

8.9.4 Transformation

Once a path has been constructed, a common operation is to apply a transformation to it. This is equivalent to transforming every point in the existing path. A path can be transformed using `fz_transform_path`.

```
/*
    fz_transform_path: Transform a path by a given
    matrix.

    path: The path to modify (must not be a packed path).

    transform: The transform to apply.

    Throws exceptions if the path is packed, or on failure
    to allocate.
*/
void fz_transform_path(fz_context *ctx, fz_path *path, const fz_matrix
     *transform);
```

This counts as modifying a path of course, so ensure that you are the only reference holder, or `fz_clone_path` it first.

8.9.5 Bounding

Sometimes it can be desirable to know the area covered by a path. The `fz_bound_path` function enables exactly this, both for filled and stroked path. For details of the `fz_stroke_state` structure, see subsection 8.9.6 Stroking.

```
/*
    fz_bound_path: Return a bounding rectangle for a path.
    path: The path to bound.

    stroke: If NULL, the bounding rectangle given is for
    the filled path. If non-NULL the bounding rectangle
    given is for the path stroked with the given attributes.

    ctm: The matrix to apply to the path during stroking.

    r: Pointer to a fz_rect which will be used to hold
    the result.
*/
fz_rect *fz_bound_path(fz_context *ctx, const fz_path *path, const
     fz_stroke_state *stroke, const fz_matrix *ctm, fz_rect *r);
```

8.9.6 Stroking

Where filling a path simply requires details of the fill to be used, stroking a path can radically alter its appearance. The details of the stroke attributes are passed in a `fz_stroke_state` structure.

Stroke states are created and managed with reference counting using the functions described below, but unlike other structures, the definition of the structure itself is public. Callers are expected to alter the different fields in the structure themselves. The sole exception to this is the `refs` field, that should only be altered using the usual `fz_keep_stroke_state` and `fz_drop_stroke_state` mechanisms.

```
typedef struct fz_stroke_state_s fz_stroke_state;

typedef enum fz_linecap_e
{
    FZ_LINECAP_BUTT = 0,
    FZ_LINECAP_ROUND = 1,
    FZ_LINECAP_SQUARE = 2,
    FZ_LINECAP_TRIANGLE = 3
} fz_linecap;

typedef enum fz_linejoin_e
{
    FZ_LINEJOIN_MITER = 0,
    FZ_LINEJOIN_ROUND = 1,
    FZ_LINEJOIN_BEVEL = 2,
    FZ_LINEJOIN_MITER_XPS = 3
} fz_linejoin;

struct fz_stroke_state_s
{
    int refs;
    fz_linecap start_cap, dash_cap, end_cap;
    fz_linejoin linejoin;
    float linewidth;
    float miterlimit;
    float dash_phase;
    int dash_len;
    float dash_list[32];
};
```

It is hoped that the meaning of the individual fields within a `fz_stroke_state` structure are self evident to anyone working in this field. If you are unfamiliar with any of the concepts here, see "The PDF Reference Manual" or "The Postscript Language Reference Manual" for more details.

Most simply a reference to a stroke state structure can be obtained by calling `fz_new_stroke_state`:

```
/*
    fz_new_stroke_state: Create a new (empty) stroke state
    structure (with no dash data) and return a reference to it.

    Throws exception on failure to allocate.
*/
fz_stroke_state *fz_new_stroke_state(fz_context *ctx);
```

For stroke states that include dash information, call:

```
/*
    fz_new_stroke_state_with_dash_len: Create a new (empty)
    stroke state structure, with room for dash data of the
    given length, and return a reference to it.

    len: The number of dash elements to allow room for.

    Throws exception on failure to allocate.
*/
fz_stroke_state *fz_new_stroke_state_with_dash_len(fz_context *ctx, int
     len);
```

Once obtained, references can be kept or dropped in the usual fashion:

```
/*
    fz_keep_stroke_state: Take an additional reference to
    a stroke state structure.

    No modifications should be carried out on a stroke
    state to which more than one reference is held, as
    this can cause race conditions.
    Never throws exceptions.
*/
fz_stroke_state *fz_keep_stroke_state(fz_context *ctx, const
    fz_stroke_state *stroke);

/*
    fz_drop_stroke_state: Drop a reference to a stroke
    state structure, destroying the structure if it is
    the last reference.

    Never throws exceptions.
*/
void fz_drop_stroke_state(fz_context *ctx, const fz_stroke_state
    *stroke);
```

Once more than one reference is held to a stroke state, it should be considered ‘frozen’ or ‘immutable’ as other reference holders may be confused by changes to it. Accordingly, we provide functions to ensure that we are holding a reference to an ‘unshared’ stroke state:

```
/*
    fz_unshare_stroke_state: Given a reference to a
    (possibly) shared stroke_state structure, return
    a reference to an equivalent stroke_state structure
    that is guaranteed to be unshared (i.e. one that can
    safely be modified).

    shared: The reference to a (possibly) shared structure
    to unshare. Ownership of this reference is passed in
    to this function, even in the case of exceptions being
    thrown.

    Exceptions may be thrown in the event of failure to
    allocate if required.
*/
fz_stroke_state *fz_unshare_stroke_state(fz_context *ctx,
    fz_stroke_state *shared);

/*
    fz_unshare_stroke_state_with_dash_len: Given a reference to a
    (possibly) shared stroke_state structure, return a reference
    to a stroke_state structure (with room for a given amount of
    dash data) that is guaranteed to be unshared (i.e. one that
    can safely be modified).

    shared: The reference to a (possibly) shared structure
    to unshare. Ownership of this reference is passed in
    to this function, even in the case of exceptions being
    thrown.
    Exceptions may be thrown in the event of failure to
    allocate if required.
*/
fz_stroke_state *fz_unshare_stroke_state_with_dash_len(fz_context *ctx,
    fz_stroke_state *shared, int len);
```

Finally, we have a simple function to clone a stroke state and return a new reference to it:

```
/*
    fz_clone_stroke_state: Create an identical stroke_state
    structure and return a reference to it.

    stroke: The stroke state reference to clone.

    Exceptions may be thrown in the event of a failure to
    allocate.
*/
fz_stroke_state *fz_clone_stroke_state(fz_context *ctx, fz_stroke_state
    *stroke);
```

8.9.7 Walking

Given a path, it can be useful to be able to read it out again. MuPDF uses this internally in a output devices such as the PDF or SVG devices (see subsection 7.5.4 PDF Output Device or subsection 7.5.6 SVG Output Device) to convert paths to a new representation, and in the draw device (see subsection 7.5.2 Draw Device) for rendering.

To isolate callers from the implementation specifics of paths, MuPDF offers a mechanism to 'walk' an fz_path, getting a callback for each command in the path.

```
typedef struct
{
    /* Compulsory ones */
    void (*moveto)(fz_context *ctx, void *arg, float x, float y);
    void (*lineto)(fz_context *ctx, void *arg, float x, float y);
    void (*curveto)(fz_context *ctx, void *arg, float x1, float y1,
        float x2, float y2, float x3, float y3);
    void (*closepath)(fz_context *ctx, void *arg);
    /* Optional ones */
    void (*quadto)(fz_context *ctx, void *arg, float x1, float y1, float
        x2, float y2);
    void (*curvetov)(fz_context *ctx, void *arg, float x2, float y2,
        float x3, float y3);
    void (*curvetoy)(fz_context *ctx, void *arg, float x1, float y1,
        float x3, float y3);
    void (*rectto)(fz_context *ctx, void *arg, float x1, float y1, float
        x2, float y2);
} fz_path_walker;

/*
    fz_walk_path: Walk the segments of a path, calling the
    appropriate callback function from a given set for each
    segment of the path.

    path: The path to walk.

    walker: The set of callback functions to use. The first
    4 callback pointers in the set must be non-NULL. The
    subsequent ones can either be supplied, or can be left
    as NULL, in which case the top 4 functions will be
    called as appropriate to simulate them.

    arg: An opaque argument passed in to each callback.

    Exceptions will only be thrown if the underlying callback
    functions throw them.
*/
void fz_walk_path(fz_context *ctx, const fz_path *path, const
    fz_path_walker *walker, void *arg);
```

This function is called by giving a pointer to a structure containing callback functions, one for each type of path segment type. The function will walk the path structure and call a for each 'segment' of the path in turn.

Callers of this function should not rely on getting exactly the same sequence of path segments out as was used to construct the path; the internal representation may have been optimised to an equivalent form on construction, and this will be reflected in the callbacks received. The path passed back will however be entirely identical (modulo possible infinitesimal rounding issues).

For example, MuPDF is capable of spotting that a cubic or quadratic bezier is actually a line; in such cases it may represent it as a line internally, saving memory and processing power.

Not all path consumers can cope with the full range of segment types that MuPDF natively supports, so some of the callback entries may be left blank (i.e. set to NULL). Rather than calling such an entry, MuPDF will decompose the path segment into one of the more basic types.

For example, if a path contains a quadratic segment and the `quadto` callback entry is NULL, MuPDF will automatically decompose it to a bezier segment and call the `curveto` entry instead.

8.10 Text

8.10.1 Overview

MuPDFs central text type is an `fz_text` structure. The exact definition of this structure has evolved considerably in the past to accommodate the needs of different input formats, and it is possible this will continue in future. Accordingly we have hidden the implementation behind an interface.

Nonetheless, it is worthwhile mentioning some of the design goals that have influenced the development of this area of the code.

As `fz_text` objects are the only text objects passed across the device interface, they need to encode several layers of information. For simple rendering devices, they need to be expressive enough to allow us to exactly render the exact specified glyphs. For text output devices, they need to be expressive enough to allow the unicode values to be extracted.

Ideally, given any input format we would like to be able to output any output format from it (including the same format) with no loss of data. This means that our `fz_text` objects need to be expressive enough to represent the super-set of functionality of all input formats out there, even if we do not currently make use of all the information.

Some input formats only specify glyph ids, whereas some only specify unicode values. In order to allow us to transmute one format into another, we require `fz_text` objects to encapsulate both forms of data at the same time.

Some formats, that make use of glyph shaping, require us to cope with unicode and glyph id equivalents that have no direct 1-1 correspondence.

For bidirectional text, the order in which text may be displayed on the page may not be trivially related to the order in which it is specified in the source file. We both need to render efficiently, and to maintain the ability to recreate the initial logical order of the text.

Accordingly, our `fz_text` object represents a block of text in logical order, including font style and position, together with both unicode and glyph data (subject to the availability of the information in the original file).

If more information is required, then details of the current implementation are included in subsection 8.10.7 Implementation, otherwise just use it as a simple black box.

```
typedef struct fz_text_s fz_text;
```

Text objects are reference, with the implicit understanding that once more than one reference exists to an object, it will no longer be modified.

8.10.2 Creation

Empty `fz_text` objects can be created using the `fz_new_text` call:

```
/*
   fz_new_text: Create a new empty fz_text object.

   Throws exception on failure to allocate.
*/
fz_text *fz_new_text(fz_context *ctx);
```

Additional references can be taken/released in the usual manner:

```
/*
   fz_keep_text: Add a reference to an fz_text.

   text: text object to keep a reference to.

   Return the same text pointer.
*/
fz_text *fz_keep_text(fz_context *ctx, const fz_text *text);

/*
   fz_drop_text: Drop a reference to the object, freeing
   if if is the last one.

   text: Object to drop the reference to.
*/
void fz_drop_text(fz_context *ctx, const fz_text *text);
```

8.10.3 Population

Once created, characters can be added to the `fz_text` object either singly:

```
/*
   fz_show_glyph: Add a glyph/unicode value to a text object.

   text: Text object to add to.

   font: The font the glyph should be added in.

   trm: The transform to use for the glyph.

   glyph: The glyph id to add.

   unicode: The unicode character for the glyph.

   wmode: 1 for vertical mode, 0 for horizontal.
```

```
    bidi_level: The bidirectional level for this glyph.

    markup_dir: The direction of the text as specified in the
    markup.

    language: The language in use (if known, 0 otherwise)
    (e.g. FZ_LANG_zh_Hans).

    Throws exception on failure to allocate.
*/
void fz_show_glyph(fz_context *ctx, fz_text *text, fz_font *font, const
     fz_matrix *trm, int glyph, int unicode, int wmode, int bidi_level,
     fz_bidi_direction markup_dir, fz_text_language language);
```

or a (unicode) string at a time:

```
/*
    fz_show_string: Add a UTF8 string to a text object.

    text: Text object to add to.

    font: The font the string should be added in.

    trm: The transform to use. Will be updated according
    to the advance of the string on exit.

    s: The utf-8 string to add.

    wmode: 1 for vertical mode, 0 for horizontal.

    bidi_level: The bidirectional level for this glyph.

    markup_dir: The direction of the text as specified in the
    markup.

    language: The language in use (if known, 0 otherwise)
    (e.g. FZ_LANG_zh_Hans).

    Throws exception on failure to allocate.
*/
void fz_show_string(fz_context *ctx, fz_text *text, fz_font *font,
     fz_matrix *trm, const char *s, int wmode, int bidi_level,
     fz_bidi_direction markup_dir, fz_text_language language);
```

8.10.4 Measurement

Once a `fz_text` object has been created we can measure the area it will cover on the page:

```
/*
   fz_bound_text: Find the bounds of a given text object.

   text: The text object to find the bounds of.

   stroke: Pointer to the stroke attributes (for stroked
   text), or NULL (for filled text).

   ctm: The matrix in use.

   r: pointer to storage for the bounds.

   Returns a pointer to r, which is updated to contain the
   bounding box for the text object.
*/
fz_rect *fz_bound_text(fz_context *ctx, const fz_text *text, const
    fz_stroke_state *stroke, const fz_matrix *ctm, fz_rect *r);
```

8.10.5 Cloning

As stated before, `fz_text` objects are referenced counted. Changes or manipulations cannot safely be carried out on an object which might be shared with someone else, so we provide a mechanism to clone an object. Once cloned an object is guaranteed to be safe to modify.

```
/*
   fz_clone_text: Clone a text object.

   text: The text object to clone.

   Throws an exception on allocation failure.
*/
fz_text *fz_clone_text(fz_context *ctx, const fz_text *text);
```

8.10.6 Language

Some formats include a declaration of which language is being used for a given piece of text. This can be used to influence aspects of the text layout, including the exact choice of glyphs used in a given font. While we make relatively little use of this at present, we try to preserve the information as part of our philosophy of not losing any information unnecessarily.

Accordingly, we use ISO 639 language specification strings, for example:

```
typedef enum fz_text_language_e
{
    FZ_LANG_UNSET = 0,
    FZ_LANG_ur = FZ_LANG_TAG2('u','r'),
    FZ_LANG_urd = FZ_LANG_TAG3('u','r','d'),
    FZ_LANG_ko = FZ_LANG_TAG2('k','o'),
    FZ_LANG_ja = FZ_LANG_TAG2('j','a'),
    FZ_LANG_zh = FZ_LANG_TAG2('z','h'),
    FZ_LANG_zh_Hans = FZ_LANG_TAG3('z','h','s'),
    FZ_LANG_zh_Hant = FZ_LANG_TAG3('z','h','t'),
} fz_text_language;
```

To save space we pack these into 15 bits. Accordingly, we provide a way to pack/unpack these to/from the more normal string representations:

```
/*
    Convert ISO 639 (639-{1,2,3,5}) language specification
    strings losslessly to a 15 bit fz_text_language code.

    No validation is carried out. Obviously invalid (out
    of spec) codes will be mapped to FZ_LANG_UNSET, but
    well-formed (but undefined) codes will be blithely
    accepted.
*/
fz_text_language fz_text_language_from_string(const char *str);


/*
    Recover ISO 639 (639-{1,2,3,5}) language specification
    strings losslessly from a 15 bit fz_text_language code.

    No validation is carried out. See note above.
*/
char *fz_string_from_text_language(char str[8], fz_text_language lang);
```

8.10.7 Implementation

A fz_text structure represents a block of text. At the lowest level the constituents of a block are fz_text_items.

```
typedef struct fz_text_item_s fz_text_item;

struct fz_text_item_s
```

```
{
    float x, y;
    int gid; /* -1 for one gid to many ucs mappings */
    int ucs; /* -1 for one ucs to many gid mappings */
};
```

The items can be thought of as the individual 'characters' that make up the display, together with their position. Where possible, we attempt to give both the glyph id (`gid`) and the unicode value (`ucs`) for the character, but there are various cases where a 1-1 mapping is not possible.

Some unicode characters can result in a string of glyphs. The glyph ids will be sent in a series of `fz_text_item`s, in which the first `ucs` value will be the source unicode character, and subsequent ones will be -1.

Some sequences of unicode characters can result in a single glyph. Again, a sequence of `fz_text_items` will be sent listing the unicode values, but all but the first item will have the `gid` value set to -1.

In more complex cases, sequences of unicode characters can be transformed into a sequence of glyphs, with no direct correspondence between the source text and the output characters. In this case as many `fz_text_item`s as are required are used, with either the `gid` or `ucs` values padded out by -1s as necessary.

Different input formats offer the text in different forms. With PDF, the data within the file is (typically) in the form of glyph ids, and mechanisms are optionally provided to infer unicode values from them. Glyphs are sent in any order, and absolutely positioned on the page.

With XPS the input can be either in the form of unicode or glyph ids, and directionality information is encoded in the file. This means that the logical ordering of the glyphs is well defined.

Some formats, such as EPub and HTML, send unicode text with even less positioning information, and rely on the interpreter to perform layout. Part of this process involves inferring directional information from the source text, and then using shaping mechanisms embedded within the font to do complex conversions to give the final positioned glyph sequences.

In all such cases MuPDF will preserve the logical ordering of the unicode entries, at the cost of drawing glyphs non-monotonically onto the page.

Sequences of `fz_text_item`s that share the same characteristics are gathered together into `fz_text_span`s:

```
struct fz_text_span_s
{
    fz_font *font;
    fz_matrix trm;
    unsigned wmode : 1;   /* 0 horizontal, 1 vertical */
    unsigned bidi_level : 7; /* The bidirectional level of text */
    unsigned markup_dir : 2; /* The direction of text as marked in the
        original document */
    unsigned language : 15;  /* The language as marked in the original
        document */
    int len, cap;
    fz_text_item *items;
    fz_text_span *next;
};
```

Sequences of these spans are then gathered up into a linked list rooted in a `fz_text`.

```
struct fz_text_s
{
    int refs;
    fz_text_span *head, *tail;
};
```

8.11 Shadings

8.11.1 Overview

One of the most powerful graphical effects within PDF and other input formats is that of Shadings. Our central type representing shadings, `fz_shade` is all that we have to pass details of shadings across the `fz_device` interface.

Consequently, we need `fz_shade` to be expressive enough to cope with shadings from all possible sources, and yet we would like to avoid having to reproduce the shade handling code in all devices.

Accordingly, `fz_shade` is defined to be expressive enough to encapsulate all the different shading representations found in PDF with the data essentially unchanged. PDF is currently the super-set of shadings found in other formats. If this changes, `fz_shade` will be extended as required.

```
typedef struct fz_shade_s
{
    fz_storable storable;

    fz_rect bbox;     /* can be fz_infinite_rect */
    fz_colorspace *colorspace;

    fz_matrix matrix; /* matrix from pattern dict */
    int use_background; /* background color for fills but not 'sh' */
    float background[FZ_MAX_COLORS];
    int use_function;
    float function[256][FZ_MAX_COLORS + 1];

    int type; /* function, linear, radial, mesh */
    union
    {
        struct
        {
            int extend[2];
            float coords[2][3]; /* (x,y,r) twice */
        } l_or_r;
        struct
        {
            int vprow;
            int bpflag;
            int bpcoord;
            int bpcomp;
            float x0, x1;
            float y0, y1;
            float c0[FZ_MAX_COLORS];
            float c1[FZ_MAX_COLORS];
        } m;
        struct[]
        {
            fz_matrix matrix;
            int xdivs;
            int ydivs;
            float domain[2][2];
            float *fn_vals;
        } f;
    } u;

    fz_compressed_buffer *buffer;
} fz_shade;
```

8.11.2 Creation

Currently, there is no defined API for creating a shading due to the public nature of the structure. Just call `fz_malloc_struct(ctx, fz_shade)` and initialise the fields accordingly.

We may look to add convenience functions in the future, as this is likely to be desirable for the JNI (and other) bindings.

Shading objects are reference counted, with the implicit understanding that once more than one reference exists to a `fz_shade`, it will no longer be modified.

Additional references can be taken and dropped as usual:

```
/*
   fz_keep_shade: Add a reference to an fz_shade.

   shade: The reference to keep.

   Returns shade.
*/
fz_shade *fz_keep_shade(fz_context *ctx, fz_shade *shade);

/*
   fz_drop_shade: Drop a reference to an fz_shade.

   shade: The reference to drop. If this is the last
   reference, shade will be destroyed.
*/
void fz_drop_shade(fz_context *ctx, fz_shade *shade);
```

We also provide a function to process a given shading, by calling

8.11.3 Bounding

Once created, we can ask for the bounds of a given shade under a given transformation. This can sometimes be infinite.

```
/*
   fz_bound_shade: Bound a given shading.

   shade: The shade to bound.

   ctm: The transform to apply to the shade before bounding.

   r: Pointer to storage to put the bounds in.

   Returns r, updated to contain the bounds for the shading.
*/
fz_rect *fz_bound_shade(fz_context *ctx, fz_shade *shade, const
    fz_matrix *ctm, fz_rect *r);
```

8.11.4 Painting

For devices that require shadings as rasterised objects, we provide a function to paint a shading to an `fz_pixmap`:

```
/*
```

```
    fz_paint_shade: Render a shade to a given pixmap.

    shade: The shade to paint.

    ctm: The transform to apply.

    dest: The pixmap to render into.

    bbox: Pointer to a bounding box to limit the rendering
    of the shade.
*/
void fz_paint_shade(fz_context *ctx, fz_shade *shade, const fz_matrix
     *ctm, fz_pixmap *dest, const fz_irect *bbox);
```

This is currently used by the draw and SVG devices.

8.11.5 Decomposition

For devices that wish to get access to a higher level representation of a shading, but do not wish to access the internals of a shading directly, we provide a function to decompose a shading to a mesh.

This is called with functions to ‘prepare’ and ‘fill’ vertices respectively. The mesh is decomposed to triangles internally, each vertex is ‘prepared’ and each triangle ‘filled’ in turn.

The ordering of these calls is not guaranteed, other than the fact that a vertex will always be prepared before it is used as part of a triangle to be filled.

```
typedef struct fz_vertex_s fz_vertex;

struct fz_vertex_s
{
    fz_point p;
    float c[FZ_MAX_COLORS];
};

/*
    fz_shade_prepare_fn: Callback function type for use with
    fz_process_shade.

    arg: Opaque pointer from fz_process_shade caller.

    v: Pointer to a fz_vertex structure to populate.

    c: Pointer to an array of floats to use to populate v.
*/
```

```
typedef void (fz_shade_prepare_fn)(fz_context *ctx, void *arg, fz_vertex
    *v, const float *c);

/*
   fz_shade_process_fn: Callback function type for use with
   fz_process_shade.

   arg: Opaque pointer from fz_process_shade caller.

   av, bv, cv: Pointers to a fz_vertex structure describing
   the corner locations and colors of a triangle to be
   filled.
*/
typedef void (fz_shade_process_fn)(fz_context *ctx, void *arg, fz_vertex
    *av, fz_vertex *bv, fz_vertex *cv);

/*
   fz_process_shade: Process a shade, using supplied callback
   functions. This decomposes the shading to a mesh (even ones
   that are not natively meshes, such as linear or radial
   shadings), and processes triangles from those meshes.

   shade: The shade to process.

   ctm: The transform to use

   prepare: Callback function to 'prepare' each vertex.
   This function is passed an array of floats, and populates
   an fz_vertex structure.

   process: This function is passed 3 pointers to vertex
   structures, and actually performs the processing (typically
   filling the area between the vertexes).

   process_arg: An opaque argument passed through from caller
   to callback functions.
*/
void fz_process_shade(fz_context *ctx, fz_shade *shade, const fz_matrix
    *ctm,
           fz_shade_prepare_fn *prepare, fz_shade_process_fn *process,
               void *process_arg);
```

This function is used internally as part of `fz_paint_shade`, but is intended to also allow extraction of arbitrary shading data.

Chapter 9

Display Lists

9.0.1 Overview

While MuPDF is engineered to be as fast as possible at interpreting page contents, there is inevitably some overhead in converting from the documents native format to the stream of graphical operations (calls over the `fz_device` interface).

If you are planning to redraw the same page several times (perhaps because you are panning and zooming around a page in a viewer), then it can be advantageous to use a display List.

A display list is simply a way of packaging up a stream of graphical operations so that they can be efficiently played back, possibly with different transforms or clip rectangles.

Display lists are optimised to use as little memory as possible, but clearly are (typically) a greater user of memory than just reinterpreting the file. The big advantage of display lists, other than their speed, is that they can safely be played back without touching the underlying file. This means they can be used in other threads without having to worry about contention.

Display lists are implemented within using MuPDF using the `fz_display_list` type.

9.0.2 Creation

An empty display list can be created by the `fz_new_display_list` call.

```
/*
    fz_new_display_list: Create an empty display list.

    A display list contains drawing commands (text, images, etc.).
    Use fz_new_list_device for populating the list.

    mediabox: Bounds of the page (in points) represented by the display
        list.
*/
fz_display_list *fz_new_display_list(fz_context *ctx, const fz_rect
    *mediabox);
```

Once created it can be populated by creating a display list device instance that writes to it.

```
/*
    fz_new_list_device: Create a rendering device for a display list.

    When the device is rendering a page it will populate the
    display list with drawing commands (text, images, etc.). The
    display list can later be reused to render a page many times
    without having to re-interpret the page from the document file
    for each rendering. Once the device is no longer needed, free
    it with fz_drop_device.

    list: A display list that the list device takes ownership of.
*/
fz_device *fz_new_list_device(fz_context *ctx, fz_display_list *list);
```

Once you have created such a display list device, any calls made to that device (such as by calling `fz_run_page` or similar) will be recorded into the display list.

When you have finished writing to the display list (remembering to call `fz_close_device`), you dispose of the device as normal (by calling `fz_drop_device`). This leaves you holding the sole reference to the display list itself.

Writing to a display list is not thread safe. That is to say, do not attempt to write to a display list from more than one thread at a time. Similarly, do not attempt to read from display lists while write operations are ongoing.

9.0.3 Playback

To playback from a list, just call `fz_run_display_list`.

```
/*
    fz_run_display_list: (Re)-run a display list through a device.

    list: A display list, created by fz_new_display_list and
    populated with objects from a page by running fz_run_page on a
    device obtained from fz_new_list_device.
    dev: Device obtained from fz_new_*_device.

    ctm: Transform to apply to display list contents. May include
    for example scaling and rotation, see fz_scale, fz_rotate and
    fz_concat. Set to fz_identity if no transformation is desired.

    area: Only the part of the contents of the display list
    visible within this area will be considered when the list is
    run through the device. This does not imply for tile objects
    contained in the display list.

    cookie: Communication mechanism between caller and library
    running the page. Intended for multi-threaded applications,
    while single-threaded applications set cookie to NULL. The
    caller may abort an ongoing page run. Cookie also communicates
    progress information back to the caller. The fields inside
    cookie are continually updated while the page is being run.
*/
void fz_run_display_list(fz_context *ctx, fz_display_list *list,
    fz_device *dev, const fz_matrix *ctm, const fz_rect *area,
    fz_cookie *cookie);
```

9.0.4 Reference counting

In common with most other objects in MuPDF, fz_display_lists are reference counted. This means that once you have finished with a reference to a display list, it can safely be disposed of by calling fz_drop_display_list.

```
/*
    fz_drop_display_list: Drop a reference to a display list, freeing it
    if the reference count reaches zero.

    Does not throw exceptions.
*/
void fz_drop_display_list(fz_context *ctx, fz_display_list *list);
```

Should you wish to keep a new reference to a display list, you can generate one using fz_keep_display_list.

```
/*
    fz_keep_display_list: Keep a reference to a display list.

    Does not throw exceptions.
*/
fz_display_list *fz_keep_display_list(fz_context *ctx, fz_display_list
    *list);
```

In general, it is rare for you to want to make a new reference to a display list until write operations on one have finished. It is good form to avoid this.

9.0.5 Miscellaneous operations

There are a few other operations that can be performed efficiently on a display list. Firstly, one can request the bounds of a list.

```
/*
   fz_bound_display_list: Return the bounding box of the page recorded
       in a display list.
*/
fz_rect *fz_bound_display_list(fz_context *ctx, fz_display_list *list,
    fz_rect *bounds);
```

Secondly, one can create a new `fz_image` from a display list. This is useful for creating scalable content to embed in other document types; for instance MuPDF makes use of this to turn SVG files embedded within epub files (for illustrations and cover pages etc) into convenient objects for adding into the flow of text.

```
/*
   Create a new image from a display list.

   w, h: The conceptual width/height of the image.

   transform: The matrix that needs to be applied to the given
   list to make it render to the unit square.

   list: The display list.
*/
fz_image *fz_new_image_from_display_list(fz_context *ctx, float w, float
    h, fz_display_list *list);
```

Finally, it is possible to very quickly check if a given display list is empty or not.

```
/*
   Check for a display list being empty

   list: The list to check.

   Returns true if empty, false otherwise.
*/
int fz_display_list_is_empty(fz_context *ctx, const fz_display_list
    *list);
```

Chapter 10

The Stream interface

10.1 Overview

MuPDF is designed to run in a variety of different environments. As such, this means input can come from many different sources. On desktop computers input may come as files on backing store. For web served files, input may be streamed over a network. For systems with DRM embedded, the data may need to be decoded on the fly.

Similarly, data can be encapsulated within different formats in different ways, with multiple layers of encoding.

Accordingly, MuPDF abstracts the idea of an 'input stream' to a reusable class, `fz_stream`. Many implementations of `fz_stream`s are given by default in the core library, but the abstract nature of this class allows callers to provide implementations of their own to seamlessly extend the systems capabilities as required.

10.2 Creation

The exact mechanism for creating a stream depends upon the source for that particular stream, but typically it will involve a call to a creation function, such as `fz_open_file`.

```
/*
   fz_open_file: Open the named file and wrap it in a stream.

   filename: Path to a file. On non-Windows machines the filename should
   be exactly as it would be passed to fopen(2). On Windows machines,
   the path should be UTF-8 encoded so that non-ASCII characters can be
```

```
    represented. Other platforms do the encoding as standard anyway (and
    in most cases, particularly for MacOS and Linux, the encoding they
    use is UTF-8 anyway).
*/
fz_stream *fz_open_file(fz_context *ctx, const char *filename);
```

Alternative functions exist to allow creating streams from C level `FILE` pointers:

```
/*
    fz_open_file: Wrap an open file descriptor in a stream.

    file: An open file descriptor supporting bidirectional
    seeking. The stream will take ownership of the file
    descriptor, so it may not be modified or closed after the call
    to fz_open_file_ptr. When the stream is closed it will also close
    the file descriptor.
*/
fz_stream *fz_open_file_ptr(fz_context *ctx, FILE *file);
```

from direct memory blocks:

```
/*
    fz_open_memory: Open a block of memory as a stream.

    data: Pointer to start of data block. Ownership of the data block is
    NOT passed in.

    len: Number of bytes in data block.

    Returns pointer to newly created stream. May throw exceptions on
    failure to allocate.
*/
fz_stream *fz_open_memory(fz_context *ctx, unsigned char *data, size_t
     len);
```

and from `fz_buffer`s:

```
/*
    fz_open_buffer: Open a buffer as a stream.

    buf: The buffer to open. Ownership of the buffer is NOT passed in
    (this function takes its own reference).

    Returns pointer to newly created stream. May throw exceptions on
    failure to allocate.
*/
fz_stream *fz_open_buffer(fz_context *ctx, fz_buffer *buf);
```

There are too many other options for creating streams to list them all here, but their use should be self evident from the header file definitions. Once created, all streams can be used in the same ways.

10.3 Usage

10.3.1 Reading bytes

The simplest way to read bytes from a stream is to call `fz_read_byte` to read the next byte from a file. Akin to the standard `fgetc`, this returns -1 for end of data, or the next byte available.

```
/*
   fz_read_byte: Read the next byte from a stream.

   stm: The stream t read from.

   Returns -1 for end of stream, or the next byte. May
   throw exceptions.
*/
int fz_read_byte(fz_context *ctx, fz_stream *stm);
```

To read more than 1 byte at a time, there are two different options.

Firstly, and most efficiently, bytes can be read directly from the streams underlying buffer. For a given `fz_stream *stm`, the current position in the stream is pointed to by `stm->rp`. Bytes can simply be read out, and the pointer incremented by the number read.

To do this, you must first know how many bytes there are available to be read out. This is achieved by calling `fz_available`. If there are no bytes already decoded and awaiting reading, this call will trigger a refill of the underlying buffer, which may take noticeable time.

```
/*
   fz_available: Ask how many bytes are available immediately from
   a given stream.

   stm: The stream to read from.

   max: A hint for the underlying stream; the maximum number of
   bytes that we are sure we will want to read. If you do not know
   this number, give 1.

   Returns the number of bytes immediately available between the
   read and write pointers. This number is guaranteed only to be 0
   if we have hit EOF. The number of bytes returned here need have
   no relation to max (could be larger, could be smaller).
```

```
*/
size_t fz_available(fz_context *ctx, fz_stream *stm, size_t max);
```

To avoid needless work, a 'max' value can be supplied as a hint, telling any buffer refill operation that is triggered how many bytes are actually required. Specifying a max value does **not** guarantee you anything about the number of bytes actually made available.

Some callers may find this awkward - the need to potentially repeatedly call until you get enough bytes to fill a buffer of the required length may be tedious. Therefore as an alternative, we provide a simpler call, `fz_read`.

Designed to be similar to the standard `fread` call, this attempts to read as many bytes as possible into a supplied data block, returning the actual number of bytes successfully read.

```
/*
   fz_read: Read from a stream into a given data block.

   stm: The stream to read from.

   data: The data block to read into.

   len: The length of the data block (in bytes).

   Returns the number of bytes read. May throw exceptions.
*/
size_t fz_read(fz_context *ctx, fz_stream *stm, unsigned char *data,
    size_t len);
```

Typically the only reason that `fz_read` will not return the requested number of bytes is if we hit the end of the stream. This implies that calls to `fz_read` will block until such data is ready. For streams based on 'fast' sources like files or memory, this is an unimportant distinction.

For streams based on (say) an http download, this might result in significant delays, and an unacceptable user experience. To alleviate this problem we have a mechanism whereby such streams can signal a temporary end of data by throwing the `FZ_ERROR_TRYLATER` error. See chapter 16 Progressive Mode for more details.

To facilitate reading without blocking (or using buffers larger than required), `fz_available` can be called to find out the number of bytes that can safely be requested.

If data within a stream is not required, it can be skipped over using `fz_skip`:

```
/*
   fz_skip: Read from a stream discarding data.
```

```
    stm: The stream to read from.

    len: The number of bytes to read.

    Returns the number of bytes read. May throw exceptions.
*/
size_t fz_skip(fz_context *ctx, fz_stream *stm, size_t len);
```

As a special case, after a single byte is read, it can be pushed back into the stream, using `fz_unread_byte`:

```
/*
    fz_unread_byte: Unread the single last byte successfully
    read from a stream. Do not call this without having
    successfully read a byte.
*/
void fz_unread_byte(fz_context *ctx FZ_UNUSED, fz_stream *stm);
```

The act of reading a byte, and then, if successful pushing it back again is encapsulated in a convenience function, `fz_peek_byte`:

```
/*
    fz_peek_byte: Peek at the next byte in a stream.

    stm: The stream to peek at.

    Returns -1 for EOF, or the next byte that will be read.
*/
int fz_peek_byte(fz_context *ctx, fz_stream *stm);
```

10.3.2 Reading objects

Often, when parsing different document formats, it can be useful to read specific objects from streams, so convenience functions exist for this too. Firstly, integers of different size and endianness are catered for:

```
/*
    fz_read_[u]int(16|24|32|64)(_le)?

    Read a 16/32/64 bit signed/unsigned integer from stream,
    in big or little-endian byte orders.

    Throws an exception if EOF is encountered.
*/
uint16_t fz_read_uint16(fz_context *ctx, fz_stream *stm);
uint32_t fz_read_uint24(fz_context *ctx, fz_stream *stm);
uint32_t fz_read_uint32(fz_context *ctx, fz_stream *stm);
```

```
uint64_t fz_read_uint64(fz_context *ctx, fz_stream *stm);

uint16_t fz_read_uint16_le(fz_context *ctx, fz_stream *stm);
uint32_t fz_read_uint24_le(fz_context *ctx, fz_stream *stm);
uint32_t fz_read_uint32_le(fz_context *ctx, fz_stream *stm);
uint64_t fz_read_uint64_le(fz_context *ctx, fz_stream *stm);

int16_t fz_read_int16(fz_context *ctx, fz_stream *stm);
int32_t fz_read_int32(fz_context *ctx, fz_stream *stm);
int64_t fz_read_int64(fz_context *ctx, fz_stream *stm);

int16_t fz_read_int16_le(fz_context *ctx, fz_stream *stm);
int32_t fz_read_int32_le(fz_context *ctx, fz_stream *stm);
int64_t fz_read_int64_le(fz_context *ctx, fz_stream *stm);
```

We have functions to read both C style strings, and newline/return terminated lines:

```
/*
   fz_read_string: Read a null terminated string from the stream into
   a buffer of a given length. The buffer will be null terminated.
   Throws on failure (including the failure to fit the entire string
   including the terminator into the buffer).
*/
void fz_read_string(fz_context *ctx, fz_stream *stm, char *buffer, int
    len);

/*
   fz_read_line: Read a line from stream into the buffer until either a
   terminating newline or EOF, which it replaces with a null byte
       ('\0').

   Returns buf on success, and NULL when end of file occurs while no
       characters
   have been read.
*/
char *fz_read_line(fz_context *ctx, fz_stream *stm, char *buf, size_t
    max);
```

10.3.3 Reading bits

Streams (or sections of streams) can be treated as a string of bits, packed either most significant or least significant bits first.

To read from an msb packed stream, use `fz_read_bits`:

```
/*
   fz_read_bits: Read the next n bits from a stream (assumed to
	be packed most significant bit first).

	stm: The stream to read from.

	n: The number of bits to read, between 1 and 8*sizeof(int)
	inclusive.

	Returns (unsigned int)-1 for EOF, or the required number of bits.
*/
unsigned int fz_read_bits(fz_context *ctx, fz_stream *stm, int n);
```

Conversely, to read from a lsb packed stream, use `fz_read_rbits`:

```
/*
	fz_read_rbits: Read the next n bits from a stream (assumed to
	be packed least significant bit first).

	stm: The stream to read from.

	n: The number of bits to read, between 1 and 8*sizeof(int)
	inclusive.

	Returns (unsigned int)-1 for EOF, or the required number of bits.
*/
unsigned int fz_read_rbits(fz_context *ctx, fz_stream *stm, int n);
```

;

Whichever of these is used, reading n bits will return the results in the lowest n bits of the returned value.

After reading bits using these functions, if a return to reading bytewise (or objectwise) is required, then `fz_sync_bits` must be called.

```
/*
	fz_sync_bits: Called after reading bits to tell the stream
	that we are about to return to reading bytewise. Resyncs
	the stream to whole byte boundaries.
*/
void fz_sync_bits(fz_context *ctx FZ_UNUSED, fz_stream *stm);
```

10.3.4 Reading whole streams

As a convenience function, MuPDF provides a mechanism for reading the entire contents of a stream into an `fz_buffer`.

```
/*
	fz_read_all: Read all of a stream into a buffer.
```

```
    stm: The stream to read from

    initial: Suggested initial size for the buffer.

    Returns a buffer created from reading from the stream. May throw
    exceptions on failure to allocate.
*/
fz_buffer *fz_read_all(fz_context *ctx, fz_stream *stm, size_t initial);
```

This will throw an error (and hence not return any data) if an error is encountered during the decode of the stream. Sometimes it can be preferable to 'do the best we can' and tolerate problematic data. For such cases, we provide `fz_read_best`:

```
/*
    fz_read_best: Attempt to read a stream into a buffer. If truncated
    is NULL behaves as fz_read_all, otherwise does not throw exceptions
    in the case of failure, but instead sets a truncated flag.

    stm: The stream to read from.

    initial: Suggested initial size for the buffer.

    truncated: Flag to store success/failure indication in.

    Returns a buffer created from reading from the stream.
*/
fz_buffer *fz_read_best(fz_context *ctx, fz_stream *stm, size_t initial,
     int *truncated);
```

10.3.5 Seeking

Most stream operations simply advance the stream pointer as the stream is read. The current stream position can always be obtained using `fz_tell` (deliberately similar to the standard `ftell` call):

```
/*
    fz_tell: return the current reading position within a stream
*/
fz_off_t fz_tell(fz_context *ctx, fz_stream *stm);
```

Some streams allow you to seek within them, that is, to change the current stream pointer to a given offset. To do this, use `fz_seek` (deliberately similar to `fseek`):

```
/*
```

```
    fz_seek: Seek within a stream.

    stm: The stream to seek within.

    offset: The offset to seek to.

    whence: From where the offset is measured (see fseek).
*/
void fz_seek(fz_context *ctx, fz_stream *stm, fz_off_t offset, int
    whence);
```

In the event that a stream does not support seeking, an error will be thrown.

10.3.6 Meta data

Occasionally, it can be useful to interrogate the properties of a stream, for example the length of the stream, or whether it is coming from a progressive source (see chapter 16 Progressive Mode).

While not implemented currently, perhaps in future a particular stream user might want to interrogate information about the Mimetype of the stream, or its compression ratios.

To allow this, we have an extensible system to request Meta operations on a stream. The `fz_stream_meta` function allows such calls to be made, with a reason code to identify the required operation, and pointer and size parameters to identify data to be passed:

```
/*
    fz_stream_meta: Perform a meta call on a stream (typically to
    request meta information about a stream).

    stm: The stream to query.

    key: The meta request identifier.

    size: Meta request specific parameter - typically the size of
    the data block pointed to by ptr.

    ptr: Meta request specific parameter - typically a pointer to
    a block of data to be filled in.

    Returns -1 if this stream does not support this meta operation,
    or a meta operation specific return value.
*/
int fz_stream_meta(fz_context *ctx, fz_stream *stm, int key, int size,
    void *ptr);
```

10.3.7 Destruction

In common with most other MuPDF objects, `fz_stream`s are reference counted.

As such additional references can be taken using `fz_keep_stream` and they can be destroyed using `fz_drop_stream`.

Note that care must be taken not to use `fz_stream` objects simultaneously in more than one thread. Not only does the act of reading in one thread upset the point at which the next read will happen in another thread, no protection is provided to make operations atomic - thus the internal data can become corrupted and cause crashes.

10.4 Implementing an `fz_stream`

The above, relatively rich, set of functions are implemented on a fairly simple basic structure.

To implement your own `fz_stream`, simply define a creation function, of the form:

```
fz_stream *fz_new_stream_foo(fz_context *ctx, <more parameters here>)
{
    fz_stream *stm;
    foo_state *state;

    state = <create structure to hold foo specific stream state>
    stm = fz_new_stream(ctx, state, foo_next, foo_close);
    <set stm->seek if required>
    <set stm->meta if required>
    return stm;
}
```

Note that some `fz_try`/`fz_catch`-ery may be required as part of the setup for state.

The hard work for this function is done using `fz_new_stream`, and two ‘foo’ specific functions, `foo_next` and `foo_close`. First let’s look at `fz_new_stream`:

```
/*
    fz_new_stream: Create a new stream object with the given
    internal state and function pointers.

    state: Internal state (opaque to everything but implementation).

    next: Should provide the next set of bytes (up to max) of stream
    data. Return the number of bytes read, or EOF when there is no
    more data.
    close: Should clean up and free the internal state. May not
    throw exceptions.
*/
fz_stream *fz_new_stream(fz_context *ctx, void *state, fz_stream_next_fn
     *next, fz_stream_close_fn *close);
```

This creates the main `fz_stream` structure, populates it with the given pointers (`state`, `foo_next` and `foo_close`) and sets the internal buffer pointers up to indicate an empty buffer.

As soon as anyone tries to read from the buffer (or to find out how many bytes are available), the MuPDF stream functions will cause `foo_next` to be called. This is a function of the following type:

```
/*
    fz_stream_next_fn: A function type for use when implementing
    fz_streams. The supplied function of this type is called
    whenever data is required, and the current buffer is empty.

    stm: The stream to operate on.

    max: a hint as to the maximum number of bytes that the caller
    needs to be ready immediately. Can safely be ignored.

    Returns -1 if there is no more data in the stream. Otherwise,
    the function should find its internal state using stm->state,
    refill its buffer, update stm->rp and stm->wp to point to the
    start and end of the new data respectively, and then
    "return *stm->rp++".
*/
typedef int (fz_stream_next_fn)(fz_context *ctx, fz_stream *stm, size_t
     max);
```

When the stream is closed, the `foo_close` function will be called. This should be a function of type `fz_stream_close_fn`:

```
/*
    fz_stream_close_fn: A function type for use when implementing
    fz_streams. The supplied function of this type is called
    when the stream is closed, to release the stream specific
    state information.

    state: The stream state to release.
*/
```

In our example, if the state was created by a simple `fz_malloc_struct(ctx, foo_state)` then `foo_close` might be as simple as an `fz_free(ctx, state)`. If the internal state of the stream is more complex then the destructor will be similarly more complex.

These three functions (creation, next and close) are all that is required to define a stream.

Optionally, you can also define a seek and/or a meta function, using functions of the following types:

```
/*
   fz_stream_seek_fn: A function type for use when implementing
   fz_streams. The supplied function of this type is called when
   fz_seek is requested, and the arguments are as defined for
   fz_seek.

   The stream can find its private state in stm->state.
*/
typedef void (fz_stream_seek_fn)(fz_context *ctx, fz_stream *stm,
    fz_off_t offset, int whence);

/*
   fz_stream_meta_fn: A function type for use when implementing
   fz_streams. The supplied function of this type is called when
   fz_meta is requested, and the arguments are as defined for
   fz_meta.

   The stream can find its private state in stm->state.
*/
typedef int (fz_stream_meta_fn)(fz_context *ctx, fz_stream *stm, int
    key, int size, void *ptr);
```

Chapter 11

The Output interface

11.1 Overview

In the same way as `fz_stream`s abstracts input streams, MuPDF uses a reusable class, `fz_output`, to abstract output streams.

11.2 Creation

The exact function to call to create an output stream depends on the specific stream required, but they generally follow a similar format. Some common examples are:

```
/*
    fz_new_output_with_file: Open an output stream that writes to a
    FILE *.

    file: The file to write to.

    close: non-zero if we should close the file when the fz_output
    is closed.
*/
fz_output *fz_new_output_with_file_ptr(fz_context *ctx, FILE *file, int
     close);

/*
    fz_new_output_with_path: Open an output stream that writes to a
    given path.

    filename: The filename to write to (specified in UTF-8).
    append: non-zero if we should append to the file, rather than
    overwriting it.
*/
fz_output *fz_new_output_with_path(fz_context *, const char *filename,
     int append);

/*
    fz_new_output_with_buffer: Open an output stream that appends
    to a buffer.

    buf: The buffer to append to.
*/
fz_output *fz_new_output_with_buffer(fz_context *ctx, fz_buffer *buf);
```

One of the most common use cases is to get an output stream that goes to `stdout` or `stderr`, and we provide convenience functions for exactly this. In addition we allow the streams for `stdout` and `stderr` to be replaced by other `fz_outputs`, thus allowing redirection to be changed simply for any of our existing tools:

```
/*
    fz_stdout: The standard out output stream. By default
    this stream writes to stdout. This may be overridden
    using fz_set_stdout.
*/
fz_output *fz_stdout(fz_context *ctx);

/*
    fz_stderr: The standard error output stream. By default
    this stream writes to stderr. This may be overridden
    using fz_set_stderr.
*/
fz_output *fz_stderr(fz_context *ctx);

/*
    fz_set_stdout: Replace default standard output stream
    with a given stream.

    out: The new stream to use.
*/
void fz_set_stdout(fz_context *ctx, fz_output *out);

/*
    fz_set_stderr: Replace default standard error stream
    with a given stream.

    err: The new stream to use.
*/
void fz_set_stderr(fz_context *ctx, fz_output *err);
```

11.3 Usage

11.3.1 Writing bytes

Single bytes can be written to `fz_output` streams using `fz_write_byte`:

```
/*
   fz_write_byte: Write a single byte.

   out: stream to write to.

   x: value to write
*/
void fz_write_byte(fz_context *ctx, fz_output *out, unsigned char x);
```

Blocks of bytes can be written to `fz_output` streams using `fz_write`:

```
/*
   fz_write: Write data to output. Designed to parallel
   fwrite.

   out: Output stream to write to.

   data: Pointer to data to write.

   size: Length of data to write.
*/
void fz_write(fz_context *ctx, fz_output *out, const void *data, size_t
    size);
```

11.3.2 Writing objects

We have convenience functions for outputting 16 and 32bit integers in both big and little endian forms:

```
/*
   fz_write_int32_be: Write a big-endian 32-bit binary integer.
*/
void fz_write_int32_be(fz_context *ctx, fz_output *out, int x);

/*
   fz_write_int32_le: Write a little-endian 32-bit binary integer.
*/
void fz_write_int32_le(fz_context *ctx, fz_output *out, int x);

/*
   fz_write_int16_be: Write a big-endian 16-bit binary integer.
```

```
*/
void fz_write_int16_be(fz_context *ctx, fz_output *out, int x);

/*
    fz_write_int16_le: Write a little-endian 16-bit binary integer.
*/
void fz_write_int16_le(fz_context *ctx, fz_output *out, int x);
```

And a function for outputting utf-8 encoded unicode characters:

```
/*
    fz_write_rune: Write a UTF-8 encoded unicode character.
*/
void fz_write_rune(fz_context *ctx, fz_output *out, int rune);
```

11.3.3 Writing strings

To output printable strings, we have the simple `fputc`, `fputs` and `fputrune` equivalents:

```
/*
    fz_putc: fputc equivalent for output streams.
*/
#define fz_putc(C,O,B) fz_write_byte(C, O, B)

/*
    fz_puts: fputs equivalent for output streams.
*/
#define fz_puts(C,O,S) fz_write(C, O, (S), strlen(S))

/*
    fz_putrune: fputrune equivalent for output streams.
*/
#define fz_putrune(C,O,R) fz_write_rune(C, O, R)
```

We also provide a family of enhanced output functions, patterned after `fprintf`:

```
/*
    fz_vsnprintf: Our customised vsnprintf routine.
    Takes %c, %d, %o, %s, %u, %x, as usual.
    Modifiers are not supported except for zero-padding
    ints (e.g. %02d, %03o, %04x, etc).
    %f and %g both output in "as short as possible hopefully lossless
    non-exponent" form, see fz_ftoa for specifics.
    %C outputs a utf8 encoded int.
    %M outputs a fz_matrix*.
    %R outputs a fz_rect*.
    %P outputs a fz_point*.
    %q and %( output escaped strings in C/PDF syntax.
    %ll{d,u,x} indicates that the values are 64bit.
    %z{d,u,x} indicates that the value is a size_t.
    %Z{d,u,x} indicates that the value is a fz_off_t.
*/
size_t fz_vsnprintf(char *buffer, size_t space, const char *fmt, va_list
    args);

/*
    fz_snprintf: The non va_list equivalent of fz_vsnprintf.
*/
size_t fz_snprintf(char *buffer, size_t space, const char *fmt, ...);

/*
    fz_printf: fprintf equivalent for output streams. See fz_snprintf.
*/
void fz_printf(fz_context *ctx, fz_output *out, const char *fmt, ...);

/*
    fz_vprintf: vfprintf equivalent for output streams. See fz_vsnprintf.
*/
void fz_vprintf(fz_context *ctx, fz_output *out, const char *fmt,
    va_list ap);
```

11.3.4 Seeking

As with `fz_streams`, `fz_outputs` normally move linearly, but in special cases, can be seekable.

```
/*
    fz_seek_output: Seek to the specified file position. See fseek
    for arguments.

    Throw an error on unseekable outputs.
*/
void fz_seek_output(fz_context *ctx, fz_output *out, fz_off_t off, int
    whence);
```

Unlike `fz_streams`, which support `fz_tell` in all cases, `fz_outputs` can only `fz_tell_output` if they are seekable:

```
/*
    fz_tell_output: Return the current file position. Throw an error
    on unseekable outputs.
*/
fz_off_t fz_tell_output(fz_context *ctx, fz_output *out);
```

11.4 Implementing a `fz_output`

The above, relatively rich, set of functions are implemented on a fairly simple basic structure.

To implement your own `fz_output`, simply define a creation function of the form:

```
fz_output *fz_new_output_foo(fz_context *ctx, <more parameters here>)
{
    fz_output *out = fz_new_output(ctx, <state>, foo_write, foo_close);
    <optionally set out->seek = foo_seek>
    <optionally set out->tell = foo_tell>
    return out;
}
```

This has parallels with the implementation of `fz_stream`s, but is not quite identical.

If ¡state¿ needs no destruction, then we can use NULL in place of `foo_close`. Otherwise `foo_close` should be a function of type:

```
/*
    fz_output_close_fn: A function type for use when implementing
    fz_outputs. The supplied function of this type is called
    when the output stream is closed, to release the stream specific
    state information.

    state: The output stream state to release.
*/
typedef void (fz_output_close_fn)(fz_context *ctx, void *state);
```

This can be as simple as doing `fz_free(ctx, state)`, or (depending on the complexity of the state structure) can require more involved operations to clean up.

The most important function and the only non-optional one is `foo_write`. This is a function of type:

```
/*
    fz_output_write_fn: A function type for use when implementing
    fz_outputs. The supplied function of this type is called
    whenever data is written to the output.

    state: The state for the output stream.

    data: a pointer to a buffer of data to write.

    n: The number of bytes of data to write.
```

```
*/
typedef void (fz_output_write_fn)(fz_context *ctx, void *state, const
    void *data, size_t n);
```

Optionally we can choose to have our output stream support `fz_seek_output` and `fz_tell_output`. To do that we must implement `foo_seek` and `foo_tell` respectively, and assign them `out->seek` and `out->tell` during creation.

```
/*
   fz_output_seek_fn: A function type for use when implementing
   fz_outputs. The supplied function of this type is called when
   fz_seek_output is requested.

   state: The output stream state to seek within.

   offset, whence: as defined for fs_seek_output.
*/
typedef void (fz_output_seek_fn)(fz_context *ctx, void *state, fz_off_t
    offset, int whence);

/*
   fz_output_tell_fn: A function type for use when implementing
   fz_outputs. The supplied function of this type is called when
   fz_tell_output is requested.

   state: The output stream state to report on.

   Returns the offset within the output stream.
*/
typedef fz_off_t (fz_output_tell_fn)(fz_context *ctx, void *state);
```

Chapter 12

Rendered Output Formats

12.1 Overview

MuPDFs built in renderer (see /rjwrefDrawDevice) produces in-memory arrays of contone values for areas of document pages. The MuPDF library includes routines to be able to output these areas to a number of different output formats.

Typically these devices all follow a similar pattern, enabling either full page or banded rendering to be performed according to the requirements of the particular application.

For a given format `XXX`, there tend to be 3 functions defined:

```
void fz_save_pixmap_as_XXX(fz_context *ctx, fz_pixmap *pixmap, char
    *filename);

void fz_write_pixmap_as_XXX(fz_context *ctx, fz_output *out, fz_pixmap
    *pixmap);

fz_band_writer *fz_new_XXX_band_writer(fz_context *ctx, fz_output *out);
```

The first function outputs a pixmap to a utf-8 encoded filename as an `XXX` formatted file. If the pixmap is not in a suitable colorspace/alpha configuration, then an exception will be thrown.

The second function does the same thing, but to a given `fz_output` rather than to a named file.

The third function returns an `fz_band_writer` to do the same thing.

12.2 Band Writers

The purpose of the `fz_band_writer` mechanism is to allow banded rendering; rather than having to allocate a pixmap large enough to hold the entire page at once, we instead render bands across the page and feed those to the `fz_band_writer` which assembles them into a properly formed `XXX` format output stream.

Having created a `fz_band_writer` using one of the creation functions defined in the following sections, the page output starts by calling `fz_write_header`. This both configures the band writer for the type of data that is being sent, and triggers the output of the file header.

```
/*
   fz_write_header: Cause a band writer to write the header for
   a banded image with the given properties/dimensions etc. This
   also configures the bandwriter for the format of the data to be
   passed in future calls.

   w, h: Width and Height of the entire page.

   n: Number of components (including alphas).

   alpha: Number of alpha components.

   xres, yres: X and Y resolutions in dpi.

   pagenum: Page number

   Throws exception if incompatible data format.
*/
void fz_write_header(fz_context *ctx, fz_band_writer *writer, int w, int
    h, int n, int alpha, int xres, int yres, int pagenum);
```

Next, the caller should render bands of the page in turn, and pass them in to `fz_write_band`.

```
/*
   fz_write_band: Cause a band writer to write the next band
   of data for an image.

   stride: The byte offset from the first byte of the data
   for a pixel to the first byte of the data for the same pixel
   on the row below.

   band_height: The number of lines in this band.

   samples: Pointer to first byte of the data.
*/
```

```
void fz_write_band(fz_context *ctx, fz_band_writer *writer, int stride,
    int band_height, const unsigned char *samples);
```

Once enough data has been sent, the band writer automatically writes any trailer for the file.

At this point, the caller can either call `fz_write_header` to start a new page, or they can call `fz_drop_band_writer` to clean up.

```
void fz_drop_band_writer(fz_context *ctx, fz_band_writer *writer);
```

12.3 PNM

The simplest output format supported is that of PNM. The pixmap can be greyscale, or RGB, with or without alpha (though the alpha plane is always ignored on writing).

```
/*
   fz_save_pixmap_as_pnm: Save a pixmap as a PNM image file.
*/
void fz_save_pixmap_as_pnm(fz_context *ctx, fz_pixmap *pixmap, char
    *filename);

void fz_write_pixmap_as_pnm(fz_context *ctx, fz_output *out, fz_pixmap
    *pixmap);

fz_band_writer *fz_new_pnm_band_writer(fz_context *ctx, fz_output *out);
```

12.4 PAM

Related to PNM we have PAM. The pixmap formats here can be greyscale, RGB or CMYK, with or without alpha (and the alpha plane is written to the file). The TUPLTYPE in the image header reflects the color and alpha configuration, though not all readers support all variants.

```
/*
   fz_save_pixmap_as_pam: Save a pixmap as a PAM image file.
*/
void fz_save_pixmap_as_pam(fz_context *ctx, fz_pixmap *pixmap, char
    *filename);

void fz_write_pixmap_as_pam(fz_context *ctx, fz_output *out, fz_pixmap
    *pixmap);
```

```
fz_band_writer *fz_new_pam_band_writer(fz_context *ctx, fz_output *out);
```

12.5 PBM

Bitmaps suitable for output to the PBM format are generated by drawing to greyscale contone (with no alpha), and then halftoning down to monochrome.

```
/*
    fz_save_bitmap_as_pbm: Save a bitmap as a PBM image file.
*/
void fz_save_bitmap_as_pbm(fz_context *ctx, fz_bitmap *bitmap, char
    *filename);

void fz_write_bitmap_as_pbm(fz_context *ctx, fz_output *out, fz_bitmap
    *bitmap);

fz_band_writer *fz_new_pbm_band_writer(fz_context *ctx, fz_output *out);
```

12.6 PKM

Bitmaps suitable for output to the PKM format are generated by drawing to CMYK contone (with no alpha), and then halftoning down to give 1bpc cmyk.

```
/*
    fz_save_bitmap_as_pkm: Save a 4bpp cmyk bitmap as a PAM image file.
*/
void fz_save_bitmap_as_pkm(fz_context *ctx, fz_bitmap *bitmap, char
    *filename);

void fz_write_bitmap_as_pkm(fz_context *ctx, fz_output *out, fz_bitmap
    *bitmap);

fz_band_writer *fz_new_pkm_band_writer(fz_context *ctx, fz_output *out);
```

12.7 PNG

The PNG format will accept either greyscale or RGB pixmaps, with or without alpha. As a special case, alpha only pixmaps are accepted and written as greyscale.

```
/*
```

```
    fz_save_pixmap_as_png: Save a pixmap as a PNG image file.
*/
void fz_save_pixmap_as_png(fz_context *ctx, fz_pixmap *pixmap, const
     char *filename);

/*
    Write a pixmap to an output stream in PNG format.
*/
void fz_write_pixmap_as_png(fz_context *ctx, fz_output *out, const
     fz_pixmap *pixmap);

/*
    fz_new_png_band_writer: Obtain a fz_band_writer instance
    for producing PNG output.
*/
fz_band_writer *fz_new_png_band_writer(fz_context *ctx, fz_output *out);
```

Because PNG is such a useful and widely used format, we have another couple of functions. These take either an `fz_image` or an `fz_pixmap` and produce an `fz_buffer` containing a PNG encoded version. This is very useful when converting between document formats as we can frequently use a PNG version of an image as a replacement for other image formats that may not be supported.

```
/*
    Create a new buffer containing the image/pixmap in PNG format.
*/
fz_buffer *fz_new_buffer_from_image_as_png(fz_context *ctx, fz_image
     *image);
fz_buffer *fz_new_buffer_from_pixmap_as_png(fz_context *ctx, fz_pixmap
     *pixmap);
```

12.8 PWG/CUPS

The PWG format is intended to encapsulate output for printers. As such there are many values that can be set in the headers. To allow for this, we expose these fields as an options structure that can be fed into the output functions.

```
typedef struct fz_pwg_options_s fz_pwg_options;

struct fz_pwg_options_s
{
    /* These are not interpreted as CStrings by the writing code, but
     * are rather copied directly out. */
    char media_class[64];
    char media_color[64];
    char media_type[64];
    char output_type[64];

    unsigned int advance_distance;
    int advance_media;
    int collate;
    int cut_media;
    int duplex;
    int insert_sheet;
    int jog;
    int leading_edge;
    int manual_feed;
    unsigned int media_position;
    unsigned int media_weight;
    int mirror_print;
    int negative_print;
    unsigned int num_copies;
    int orientation;
    int output_face_up;
    unsigned int PageSize[2];
    int separations;
    int tray_switch;
    int tumble;

    int media_type_num;
    int compression;
    unsigned int row_count;
    unsigned int row_feed;
    unsigned int row_step;

    /* These are not interpreted as CStrings by the writing code, but
     * are rather copied directly out. */
    char rendering_intent[64];
    char page_size_name[64];
};
```

No documentation for these fields is given here - for more information see the PWG specification.

There are 2 sets of output functions available for PWG, those that take `fz_pixmaps` (for contone output) and those that take `f_bitmaps` (for halftoned output).

PWG files are structured as a header (to identify the format), followed by a stream of pages (images). Those functions that save (or write) a complete file include the file header as part of their output. If the option is used to append to a file, then the header is not added, as we presume we are appending new page information to the end of an existing file.

In circumstances when the header is not output automatically (such as when using the band writer) the header output must be triggered manually, by calling:

```
/*
   Output the file header to a pwg stream, ready for pages to follow it.
*/
void fz_write_pwg_file_header(fz_context *ctx, fz_output *out);
```

12.8.1 Contone

The PWG writer can accept pixmaps in greyscale, RGB and CMYK format, with no alpha planes.

PWG files can be saved to a file using:

```
/*
   fz_save_pixmap_as_pwg: Save a pixmap as a pwg

   filename: The filename to save as (including extension).

   append: If non-zero, then append a new page to existing file.

   pwg: NULL, or a pointer to an options structure (initialised to zero
   before being filled in, for future expansion).
*/
void fz_save_pixmap_as_pwg(fz_context *ctx, fz_pixmap *pixmap, char
    *filename, int append, const fz_pwg_options *pwg);
```

The file header will only be sent in the case where we are not appending to an existing file.

Alternatively, pages may be sent to an output stream. Two functions exist to do this. The first always sends a complete PWG file (including header):

```
/*
   Output a pixmap to an output stream as a pwg raster.
*/
void fz_write_pixmap_as_pwg(fz_context *ctx, fz_output *out, const
    fz_pixmap *pixmap, const fz_pwg_options *pwg);
```

The second sends just the page data, and is therefore suitable for sending the second or subsequent pages in a file. Alternatively, the header can be sent manually, and then this function can be used for all the pages in a file.

```
/*
   Output a page to a pwg stream to follow a header, or other pages.
*/
void fz_write_pixmap_as_pwg_page(fz_context *ctx, fz_output *out, const
    fz_pixmap *pixmap, const fz_pwg_options *pwg);
```

Finally, a standard band writer can be used:

```
/*
   fz_new_pwg_band_writer: Generate a new band writer for
   contone PWG format images.
*/
fz_band_writer *fz_new_pwg_band_writer(fz_context *ctx, fz_output *out,
    const fz_pwg_options *pwg);
```

In all cases, a NULL value can be sent for the `fz_pwg_options` field, in which case default values will be used.

12.8.2 Mono

The monochrome version of the PWG writer parallels the contone one. It can accept monochrome bitmaps only.

PWG files can be saved to a file using:

```
/*
   fz_save_bitmap_as_pwg: Save a bitmap as a pwg

   filename: The filename to save as (including extension).

   append: If non-zero, then append a new page to existing file.

   pwg: NULL, or a pointer to an options structure (initialised to zero
   before being filled in, for future expansion).
*/
void fz_save_bitmap_as_pwg(fz_context *ctx, fz_bitmap *bitmap, char
    *filename, int append, const fz_pwg_options *pwg);
```

The file header will only be sent in the case where we are not appending to an existing file.

Alternatively, pages may be sent to an output stream. Two functions exist to do this. The first always sends a complete PWG file (including header):

```
/*
   Output a bitmap to an output stream as a pwg raster.
*/
void fz_write_bitmap_as_pwg(fz_context *ctx, fz_output *out, const
    fz_bitmap *bitmap, const fz_pwg_options *pwg);
```

The second sends just the page data, and is therefore suitable for sending the second or subsequent pages in a file. Alternatively, the header can be sent manually, and then this function can be used for all the pages in a file.

```
/*
   Output a bitmap page to a pwg stream to follow a header, or other
       pages.
*/
void fz_write_bitmap_as_pwg_page(fz_context *ctx, fz_output *out, const
    fz_bitmap *bitmap, const fz_pwg_options *pwg);
```

Finally, a standard band writer can be used:

```
/*
   fz_new_mono_pwg_band_writer: Generate a new band writer for
   PWG format images.
*/
fz_band_writer *fz_new_mono_pwg_band_writer(fz_context *ctx, fz_output
    *out, const fz_pwg_options *pwg);
```

In all cases, a NULL value can be sent for the `fz_pwg_options` field, in which case default values will be used.

12.9 TGA

The TGA writer can accept pixmaps in greyscale, RGB and BGR formats, with and without alpha.

```
/*
   fz_save_pixmap_as_tga: Save a pixmap as a TGA image file.
   Can accept RGB, BGR or Grayscale pixmaps, with or without
   alpha.
*/
void fz_save_pixmap_as_tga(fz_context *ctx, fz_pixmap *pixmap, const
    char *filename);

/*
   Write a pixmap to an output stream in TGA format.
   Can accept RGB, BGR or Grayscale pixmaps, with or without
   alpha.
*/
void fz_write_pixmap_as_tga(fz_context *ctx, fz_output *out, fz_pixmap
    *pixmap);

/*
   fz_new_tga_band_writer: Generate a new band writer for TGA
   format images. Note that image must be generated vertically
   flipped for use with this writer!

   Can accept RGB, BGR or Grayscale pixmaps, with or without
   alpha.
   is_bgr: True, if the image is generated in bgr format.
*/
fz_band_writer *fz_new_tga_band_writer(fz_context *ctx, fz_output *out,
    int is_bgr);
```

12.10 PCL

PCL is not a standard image format, rather it is a page description language for printers. Unfortunately, the exact implementation of PCL varies from printer to printer, so it can be necessary to tweak the output according to the exact intended destination.

Accordingly, we have a `pcl_options` structure to allow this to happen. To use this, you simply define a `pcl_options` structure on the stack:

```
pcl_options options = { 0 };
```

Next you populate those options. Typically this is done by requesting a preset from our current defined set.

```
/*
    fz_pcl_preset: Retrieve a set of fz_pcl_options suitable for a given
    preset.

    opts: pointer to options structure to populate.

    preset: Preset to fetch. Currently defined presets include:
       ljet4   HP DeskJet
       dj500   HP DeskJet 500
       fs600   Kyocera FS-600
       lj      HP LaserJet, HP LaserJet Plus
       lj2     HP LaserJet IIp, HP LaserJet IId
       lj3     HP LaserJet III
       lj3d    HP LaserJet IIId
       lj4     HP LaserJet 4
       lj4pl   HP LaserJet 4 PL
       lj4d    HP LaserJet 4d
       lp2563b HP 2563B line printer
       oce9050 Oce 9050 Line printer

   Throws exception on unknown preset.
*/
void fz_pcl_preset(fz_context *ctx, fz_pcl_options *opts, const char
    *preset);
```

These options can then be tweaked further using `fz_pcl_option`:

```
/*
   fz_pcl_option: Set a given PCL option to a given value in the
   supplied options structure.

   opts: The option structure to modify,

   option: The option to change.

   val: The value that the option should be set to. Acceptable ranges of
   values depend on the option in question.

   Throws an exception on attempt to set an unknown option, or an
   illegal value.

   Currently defined options/values are as follows:

      spacing,0            No vertical spacing capability
      spacing,1            PCL 3 spacing (<ESC>*p+<n>Y)
      spacing,2            PCL 4 spacing (<ESC>*b<n>Y)
      spacing,3            PCL 5 spacing (<ESC>*b<n>Y and clear seed
                           row)
      mode2,0 or 1         Disable/Enable mode 2 graphics compression
      mode3,0 or 1         Disable/Enable mode 3 graphics compression
      mode3,0 or 1         Disable/Enable mode 3 graphics compression
      eog_reset,0 or 1     End of graphics (<ESC>*rB) resets all
                           parameters
      has_duplex,0 or 1    Duplex supported (<ESC>&l<duplex>S)
      has_papersize,0 or 1 Papersize setting supported
                           (<ESC>&l<sizecode>A)
      has_copies,0 or 1    Number of copies supported
                           (<ESC>&l<copies>X)
      is_ljet4pjl,0 or 1   Disable/Enable HP 4PJL model-specific output
      is_oce9050,0 or 1    Disable/Enable Oce 9050 model-specific
                           output
*/
void fz_pcl_option(fz_context *ctx, fz_pcl_options *opts, const char
    *option, int val);
```

12.10.1 Color

Color PCL output can be generated from RGB pixmaps with alpha (though the alpha is ignored) using:

```
void fz_save_pixmap_as_pcl(fz_context *ctx, fz_pixmap *pixmap, char
    *filename, int append, const fz_pcl_options *pcl);
```

```
void fz_write_pixmap_as_pcl(fz_context *ctx, fz_output *out, const
    fz_pixmap *pixmap, const fz_pcl_options *pcl);

fz_band_writer *fz_new_color_pcl_band_writer(fz_context *ctx, fz_output
    *out, const fz_pcl_options *options);
```

This is 24bpp RGB output, relying on the printers ability to dither. Blank lines are skipped, repeated lines are coded efficiently, and other lines are coded using deltas. Nonetheless file sizes can still be large with this output method.

12.10.2 Mono

Monochrome PCL output can be generated from monochrome bitmaps. These are generated by rendering to greyscale (no alpha) pixmaps and dithering down. The functions in question are:

```
fz_band_writer *fz_new_mono_pcl_band_writer(fz_context *ctx, fz_output
    *out, const fz_pcl_options *options);

void fz_write_bitmap_as_pcl(fz_context *ctx, fz_output *out, const
    fz_bitmap *bitmap, const fz_pcl_options *pcl);

void fz_save_bitmap_as_pcl(fz_context *ctx, fz_bitmap *bitmap, char
    *filename, int append, const fz_pcl_options *pcl);
```

12.11 Postscript

Postscript output is currently done as image output rather than high-level objects.

Pixmaps suitable for PS image output are greyscale, RGB or CMYK with no alpha.

```
void fz_write_pixmap_as_ps(fz_context *ctx, fz_output *out, const
    fz_pixmap *pixmap);

void fz_save_pixmap_as_ps(fz_context *ctx, fz_pixmap *pixmap, char
    *filename, int append);

fz_band_writer *fz_new_ps_band_writer(fz_context *ctx, fz_output *out);
```

Postscript requires file level headers and trailers, over and above that produced by the band writer itself. These can be generated using the following functions:

```
void fz_write_ps_file_header(fz_context *ctx, fz_output *out);
void fz_write_ps_file_trailer(fz_context *ctx, fz_output *out, int
    pages);
```

Chapter 13

The Image interface

13.1 Overview

Images are ubiquitous in document formats, and come in a huge variety of formats, ranging from full colour to monochrome, compressed to uncompressed, large to small. The ability to efficiently represent and decode 2d arrays of pixels is vital.

MuPDF represents images using an abstract type, `fz_image`. This takes the form of a base class, upon which different implementations can be built. All `fz_image`s are reference counted, using the standard `fz_keep` and `fz_drop` conventions:

```
/*
   fz_drop_image: Drop a reference to an image.

   image: The image to drop a reference to.
*/
void fz_drop_image(fz_context *ctx, fz_image *image);

/*
   fz_keep_image: Increment the reference count of an image.

   image: The image to take a reference to.

   Returns a pointer to the image.
*/
fz_image *fz_keep_image(fz_context *ctx, fz_image *image);
```

The key operation required is to be able to request a decoded version of a subarea of that image (yielding a `fz_pixmap`), suitable for rendering at a given size:

```
/*
   fz_get_pixmap_from_image: Called to get a handle to a pixmap from an
       image.

   image: The image to retrieve a pixmap from.

   subarea: The subarea of the image that we actually care about (or
       NULL
   to indicate the whole image).

   trans: Optional, unless subarea is given. If given, then on entry
       this is
   the transform that will be applied to the complete image. It should
       be
   updated on exit to the transform to apply to the given subarea of the
   image. This is used to calculate the desired width/height for
       subsampling.

   w: If non-NULL, a pointer to an int to be updated on exit to the
   width (in pixels) that the scaled output will cover.

   h: If non-NULL, a pointer to an int to be updated on exit to the
   height (in pixels) that the scaled output will cover.

   Returns a non NULL pixmap pointer. May throw exceptions.
*/
fz_pixmap *fz_get_pixmap_from_image(fz_context *ctx, fz_image *image,
    const fz_irect *subarea, fz_matrix *trans, int *w, int *h);
```

Many images have a resolution encoded within them. This may or may not be honoured in the way they are positioned on the page, and it will certainly not be honoured when zooming is taken into account, but for some operations it is useful to be able to request it.

```
/*
   fz_image_resolution: Request the natural resolution
   of an image.

   xres, yres: Pointers to ints to be updated with the
   natural resolution of an image (or a sensible default
   if not encoded).
*/
void fz_image_resolution(fz_image *image, int *xres, int *yres);
```

If no resolution is specified within the image, sensible defaults are returned.

A key ability of `fz_images` is that they are automatically cached in the `fz_store` when decoded - repeated requests for pixmaps from the same image will (not necessarily) require the image to be decoded again and again.

13.2 Standard Image Types

13.2.1 Compressed

The most common type of `fz_image` is `fz_compressed_image` - that is, an image based upon a `fz_buffer` of data in a standard compressed format, such as JPEG, PNG, TIFF, and others.

With such images, the data is held in an `fz_compressed_buffer`:

```
typedef struct fz_compressed_buffer_s
{
    fz_compression_params params;
    fz_buffer *buffer;
} fz_compressed_buffer;
```

The data is held in the buffer field, and the details of the compression used are given in the params field, of type `fz_compression_params`:

```
struct fz_compression_params_s
{
    int type;
    union {
        struct {
            int color_transform; /* Use -1 for unset */
        } jpeg;
        struct {
            int smask_in_data;
        } jpx;
        struct {
            int columns;
            int rows;
            int k;
            int end_of_line;
            int encoded_byte_align;
            int end_of_block;
            int black_is_1;
            int damaged_rows_before_error;
        } fax;
        struct
        {
            int columns;
            int colors;
            int predictor;
            int bpc;
        }
        flate;
        struct
        {
            int columns;
            int colors;
            int predictor;
            int bpc;
            int early_change;
        } lzw;
    } u;
};
```

The choice of which of the union clauses is used is made by the type field:

```
enum
{
    FZ_IMAGE_UNKNOWN = 0,

    /* Uncompressed samples */
    FZ_IMAGE_RAW,

    /* Compressed samples */
    FZ_IMAGE_FAX,
    FZ_IMAGE_FLATE,
    FZ_IMAGE_LZW,
    FZ_IMAGE_RLD,

    /* Full image formats */
    FZ_IMAGE_BMP,
    FZ_IMAGE_GIF,
    FZ_IMAGE_JPEG,
    FZ_IMAGE_JPX,
    FZ_IMAGE_JXR,
    FZ_IMAGE_PNG,
    FZ_IMAGE_PNM,
    FZ_IMAGE_TIFF,
};
```

To determine if an `fz_image` is a compressed image, call:

```
/*
    fz_compressed_image_buffer: Retrieve the underlying compressed
    data for an image.

    Returns a pointer to the underlying data buffer for an image,
    or NULL if this image is not based upon a compressed data
    buffer.
    This is not a reference counted structure, so no reference is
    returned. Lifespan is limited to that of the image itself.
*/
fz_compressed_buffer *fz_compressed_image_buffer(fz_context *ctx,
     fz_image *image);
```

The easiest way to tell if an image is a compressed image is to request its underlying buffer. If it returns NULL, you know it is not this sort of image.

13.2.2 Decoded

The next most common type of image is based upon a decoded `fz_pixmap`. These are generally only used if the pixmap takes less storage than the compressed data would.

```
/*
    fz_pixmap_image_tile: Retried the underlying fz_pixmap
    for an image.

    Returns a pointer to the underlying fz_pixmap for an image,
    or NULL if this image is not based upon an fz_pixmap.

    No reference is returned. Lifespan is limited to that of
    the image itself. If required, use fz_keep_pixmap to take
    a reference to keep it longer.
*/
fz_pixmap *fz_pixmap_image_tile(fz_context *ctx, fz_pixmap_image *cimg);
```

The easiest way to tell if an image is a decoded image is to request its underlying tile. If it returns NULL, you know it is not this sort of image.

13.2.3 Display List

The final standard sort of image in MuPDF (though more types may of course be added in future) is that based upon a display list.

we use this to easily embed one file format within another. For example, epub files frequently contain SVG images for title pages. We open the SVG image as a separate document, run it to a display list, and close the document. We can then create an image from the display list, and use this in the HTML flow of the epub document.

These images maintain the properties of the original (vector-based) document in that they remain scalable even after conversion to an image.

13.3 Creating Images

To create an image from a standard type, simply call the appropriate function. For example, if you have an `fz_buffer` with the source data:

```
/*
    fz_new_image_from_buffer: Create a new image from a
    buffer of data, inferring its type from the format
    of the data.
*/
fz_image *fz_new_image_from_buffer(fz_context *ctx, fz_buffer *buffer);
```

If the data is in a file, use:

```
/*
    fz_image_from_file: Create a new image from the contents
    of a file, inferring its type from the format of the
    data.
*/
fz_image *fz_new_image_from_file(fz_context *ctx, const char *path);
```

This loads the data into memory, and calls `fz_new_image_from_buffer` internally.

If the data cannot be recognised from its header, and more information is required, then the data can be formed in an `fz_compressed_buffer`, and an image created with:

```
/*
    fz_new_image_from_compressed_buffer: Create an image based on
    the data in the supplied compressed buffer.

    w,h: Width and height of the created image.

    bpc: Bits per component.

    colorspace: The colorspace (determines the number of components,
    and any color conversions required while decoding).

    xres, yres: The X and Y resolutions respectively.

    interpolate: 1 if interpolation should be used when decoding
    this image, 0 otherwise.

    imagemask: 1 if this is an imagemask (i.e. transparent), 0
    otherwise.

    decode: NULL, or a pointer to to a decode array. The default
    decode array is [0 1] (repeated n times, for n color components).
    colorkey: NULL, or a pointer to a colorkey array. The default
    colorkey array is [0 255] (repeatd n times, for n color
    components).

    buffer: Buffer of compressed data and compression parameters.
    Ownership of this reference is passed in.

    mask: NULL, or another image to use as a mask for this one.
    Supplying a masked image as a mask to another image is
    illegal!
*/
fz_image *fz_new_image_from_compressed_buffer(fz_context *ctx, int w,
     int h, int bpc, fz_colorspace *colorspace, int xres, int yres, int
     interpolate, int imagemask, float *decode, int *colorkey,
     fz_compressed_buffer *buffer, fz_image *mask);
```

Finally, if we have a decoded `fz_pixmap`, we can form a new image from it:

```
/*
    fz_new_image_from_pixmap: Create an image from the given
    pixmap.

    pixmap: The pixmap to base the image upon. A new reference
    to this is taken.

    mask: NULL, or another image to use as a mask for this one.
    A new reference is taken to this image. Supplying a masked
    image as a mask to another image is illegal!
*/
fz_image *fz_new_image_from_pixmap(fz_context *ctx, fz_pixmap *pixmap,
     fz_image *mask);
```

13.4 Implementing an Image Type

Support for a new type of image, can be implemented fairly simply, by defining a structure derived from an `fz_image`, perhaps:

```
typedef struct
{
    fz_image super;
    <foo specific fields>
} foo_image;
```

Then we'd define a new image creation function, `fz_new_image_from_foo`, of the form:

```
fz_image *fz_new_image_from_foo(fz_context *ctx, <foo specific
    parameters>) {
   foo_image *foo = fz_new_image(ctx, ..., foo_image, foo_get,
       foo_size, foo_drop);
   if (!foo)
      return NULL;

   <initialise foo specific fields from foo specific parameters>

   return &foo->super;
}
```

The key call here is the call to `fz_new_image`. This is a macro which wraps a call to `fz_new_image_of_size`:

```
/*
   fz_new_image_of_size: Internal function to make a new fz_image
   structure for a derived class.

   w,h: Width and height of the created image.

   bpc: Bits per component.

   colorspace: The colorspace (determines the number of components,
   and any color conversions required while decoding).

   xres, yres: The X and Y resolutions respectively.

   interpolate: 1 if interpolation should be used when decoding
   this image, 0 otherwise.

   imagemask: 1 if this is an imagemask (i.e. transparent), 0
   otherwise.

   decode: NULL, or a pointer to to a decode array. The default
   decode array is [0 1] (repeated n times, for n color components).

   colorkey: NULL, or a pointer to a colorkey array. The default
   colorkey array is [0 255] (repeatd n times, for n color
   components).

   mask: NULL, or another image to use as a mask for this one.
   A new reference is taken to this image. Supplying a masked
   image as a mask to another image is illegal!

   size: The size of the required allocated structure (the size of
   the derived structure).

   get: The function to be called to obtain a decoded pixmap.
```

```
    get_size: The function to be called to return the storage size
    used by this image.

    drop: The function to be called to dispose of this image once
    the last reference is dropped.

    Returns a pointer to an allocated structure of the required size,
    with the first sizeof(fz_image) bytes initialised as appropriate
    given the supplied parameters, and the other bytes set to zero.
*/
fz_image *fz_new_image_of_size(fz_context *ctx, int w, int h, int bpc,
     fz_colorspace *colorspace, int xres, int yres, int interpolate, int
     imagemask, float *decode, int *colorkey, fz_image *mask, int size,
     fz_image_get_pixmap_fn *get, fz_image_get_size_fn *get_size,
     fz_drop_image_fn *drop);

#define fz_new_image(CTX,W,H,B,CS,X,Y,I,IM,D,C,M,T,G,S,Z) \
((T*)Memento_label(fz_new_image_of_size(CTX,W,H,B,CS,X,Y,I,IM,D,C,M,sizeof(T),G,S,Z),#T))
```

The macro takes identical parameters to the function other than passing the structure type in place of the structure type saved, and performing a typecast to simplify the typical enclosing code.

Both function and macro take pointers to 3 functions that need to be defined for the new format. Firstly, `foo_get` is of the following type:

```
/*
    fz_get_pixmap_fn: Function type to get a decoded pixmap
    for an image.

    im: The image to decode.

    subarea: NULL, or the subarea of the image required. Expressed
    in terms of a rectangle in the original width/height of the
    image. If non NULL, this should be updated by the function to
    the actual subarea decoded - which must include the requested
    area!

    w, h: The actual width and height that the whole image would
    need to be decoded to.

    l2factor: On entry, the log 2 subsample factor required. If
    possible the decode process can take care of (all or some) of
    this subsampling, and must then update the value so the caller
    knows what remains to be done.

    Returns a reference to a decoded pixmap that satisfies the
    requirements of the request.
```

```
*/
typedef fz_pixmap *(fz_image_get_pixmap_fn)(fz_context *ctx, fz_image
    *im, fz_irect *subarea, int w, int h, int *l2factor);
```

Secondly, `foo_get_size` will be of type:

```
/*
   fz_image_get_size_fn: Function type to get the given storage
   size for an image.

   Returns the size in bytes used for a given image.
*/
typedef size_t (fz_image_get_size_fn)(fz_context *, fz_image *);
```

Finally, `foo_drop` will be of type:

```
/*
   fz_drop_image_fn: Function type to destroy an images data
   when it's reference count reaches zero.
*/
typedef void (fz_drop_image_fn)(fz_context *ctx, fz_image *image);
```

The actual deallocation of the `fz_image` block and its associated resources will be done on return from this function. The `fz_drop_image_fn` is responsible just for deallocating its implementation specific resources (i.e. the contents of `foo_image` rather than `fz_image`).

13.5 Image Caching

While caching of decoded images happens automatically within MuPDF, it is perhaps worth saying a small amount about it.

Whenever a decoded image is requested, MuPDF searches in the store (see chapter 5 Reference Counting, Memory Management and The Store) to see if a suitable pixmap exists there already. If one is found, the store remembers that is has been reused, and returned immediately - no decoding is done.

If no suitable pixmap is found, MuPDF calculates how large the image would be on a rendered page. By comparing this size to the native size of the image, it calculates a log 2 subsampling factor to use. That is, it attempts to avoid decoding the image at full size, when one 1/2 (or 1/4 etc) of the width/height would do.

A log 2 subsampling is used because a) some compression formats such as JPEG can achieve this as part of their decompression run, and b) it is easy to rapidly shrink decompressed pixmaps in this way.

The decoded and subsampled image is then placed into the store so that it will (hopefully) be found the next time a decode of the image is requested.

Chapter 14

The Document Handler interface

14.1 Overview

MuPDF is written as an extensible framework for handling different document types. Each different document format provides an fz_document_handler structure that provides the required callbacks to recognise and open files of its supported type. For example:

```
extern fz_document_handler pdf_document_handler;
extern fz_document_handler xps_document_handler;
extern fz_document_handler svg_document_handler;
...
```

At startup, the calling program must register the required document handlers. It can either register them each individually, by repeatedly calling fz_register_document_handler:

```
/*
   fz_register_document_handler: Register a handler
   for a document type.

   handler: The handler to register.
*/
void fz_register_document_handler(fz_context *ctx, const
    fz_document_handler *handler);
```

For example:

```
    fz_register_document_handler(ctx, &pdf_document_handler);
    fz_register_document_handler(ctx, &xps_document_handler);
    fz_register_document_handler(ctx, &svg_document_handler);
    ...
```

or, it can use a convenience function to register all the standard handlers enabled in a given build:

```
/*
    fz_register_document_handler: Register handlers
    for all the standard document types supported in
    this build.
*/
void fz_register_document_handlers(fz_context *ctx);
```

14.2 Implementing a Document Handler

14.2.1 Recognize and Open

To implement a new document handler, a new `fz_document_handler` structure is required. There are 3 components to such a structure, all function pointers:

```
typedef struct fz_document_handler_s
{
    fz_document_recognize_fn *recognize;
    fz_document_open_fn *open;
    fz_document_open_with_stream_fn *open_with_stream;
} fz_document_handler;
```

The first is a function to recognize a document from a magic string, typically a mimetype or a filename:

```
/*
    fz_document_recognize_fn: Recognize a document type from
    a magic string.

    magic: string to recognise - typically a filename or mime
    type.

    Returns a number between 0 (not recognized) and 100
    (fully recognized) based on how certain the recognizer
    is that this is of the required type.
*/
typedef int (fz_document_recognize_fn)(fz_context *ctx, const char
     *magic);
```

The second is a function to open a document from a filename:

```
/*
   fz_document_open_fn: Function type to open a document from a
   file.

   filename: file to open

   Pointer to opened document. Throws exception in case of error.
*/
typedef fz_document *(fz_document_open_fn)(fz_context *ctx, const char
    *filename);
```

This function can permissibly be NULL, as it can be synthesized automatically from the third entry, a function to open a document from a stream:

```
/*
   fz_document_open_with_stream_fn: Function type to open a
   document from a file.

   stream: fz_stream to read document data from. Must be
   seekable for formats that require it.

   Pointer to opened document. Throws exception in case of error.
*/
typedef fz_document *(fz_document_open_with_stream_fn)(fz_context *ctx,
    fz_stream *stream);
```

To create an `fz_document` use the `fz_new_document` macro. For a document of type foo, typically a `foo_document` structure would be defined as below:

```
typedef struct
{
   fz_document super;
   <foo specific fields>
} foo_document;
```

This would then be created using a call to `fz_new_document`, such as:

```
  foo_document *foo = fz_new_document(ctx, foo_document);
```

This returns an empty document structure with `super` populated with default values, and the foo specific fields initialized to 0. The document handler then needs to fill in the document level functions.

14.2.2 Document Level Functions

The `fz_document` structure contains a list of functions used to implement the document level calls:

```
typedef struct fz_document_s
{
    int refs;
    fz_document_drop_fn *drop_document;
    fz_document_needs_password_fn *needs_password;
    fz_document_authenticate_password_fn *authenticate_password;
    fz_document_has_permission_fn *has_permission;
    fz_document_load_outline_fn *load_outline;
    fz_document_layout_fn *layout;
    fz_document_make_bookmark_fn *make_bookmark;
    fz_document_lookup_bookmark_fn *lookup_bookmark;
    fz_document_resolve_link_fn *resolve_link;
    fz_document_count_pages_fn *count_pages;
    fz_document_load_page_fn *load_page;
    fz_document_lookup_metadata_fn *lookup_metadata;
    int did_layout;
    int is_reflowable;
} fz_document;
```

Implementations must fill in the `drop_document` field, with a pointer to a function called to free any resources help by the document when the reference count drops to 0. In the unlikely event that your implementation has no resources, this field can be left NULL.

```
/*
    fz_document_drop_fn: Called when the reference count for
    the fz_document drops to 0. The implementation should
    release any resources held by the document. The actual
    document pointer will be freed by the caller.
*/
typedef void (fz_document_drop_fn)(fz_context *ctx, fz_document *doc);
```

If your document handler is capable of handling password protected documents, then you must fill in the `needs_password` field with a pointer to a function called to enquire whether a given document needs a password:

```
/*
    fz_document_needs_password_fn: Type for a function to be
    called to enquire whether the document needs a password
    or not. See fz_needs_password for more information.
*/
typedef int (fz_document_needs_password_fn)(fz_context *ctx, fz_document
     *doc);
```

If your document handler is capable of handling password protected documents, then you must fill in the `authenticate_password` field with a pointer to a function called to attempt to authenticate a password:

```
/*
   fz_document_authenticate_password_fn: Type for a function to be
   called to attempt to authenticate a password. See
   fz_authenticate_password for more information.
*/
typedef int (fz_document_authenticate_password_fn)(fz_context *ctx,
    fz_document *doc, const char *password);
```

Certain document types encode permissions within them to say what users are allowed to do with them (printing, extracting etc). If your document handler's format has this concept, then you must fill in the `has_permission` field with a pointer to a function called to attempt to query such permissions:

```
/*
   fz_document_has_permission_fn: Type for a function to be
   called to see if a document grants a certain permission. See
   fz_document_has_permission for more information.
*/
typedef int (fz_document_has_permission_fn)(fz_context *ctx, fz_document
    *doc, fz_permission permission);
```

Certain document types can optionally include outline (table of contents) information within them. If your document handler's format has this concept, then you must fill in the `load_outline` field with a pointer to a function called to attempt to load such information if it is there:

```
/*
   fz_document_load_outline_fn: Type for a function to be called to
   load the outlines for a document. See fz_document_load_outline
   for more information.
*/
typedef fz_outline *(fz_document_load_outline_fn)(fz_context *ctx,
    fz_document *doc);
```

If your document format requires a layout pass before it can be viewed, then you must fill in the `layout` field with a pointer to a function called to perform such a layout:

```
/*
   fz_document_layout_fn: Type for a function to be called to lay
   out a document. See fz_layout_document for more information.
*/
typedef void (fz_document_layout_fn)(fz_context *ctx, fz_document *doc,
    float w, float h, float em);
```

If your document requires a layout pass, you should provide functions to both make and resolve bookmarks to enable reader positions to be kept over layout changes. Accordingly the `make_bookmark` and `lookup_bookmark` fields should be filled out:

```
/*
   fz_document_make_bookmark_fn: Type for a function to make
   a bookmark. See fz_make_bookmark for more information.
*/
typedef fz_bookmark (fz_document_make_bookmark_fn)(fz_context *ctx,
    fz_document *doc, int page);

/*
   fz_document_lookup_bookmark_fn: Type for a function to lookup
   a bookmark. See fz_lookup_bookmark for more information.
*/
typedef int (fz_document_lookup_bookmark_fn)(fz_context *ctx,
    fz_document *doc, fz_bookmark mark);
```

Some document formats can encode internal links that point to another page in the document. If your document supports this concept, then you must fill in the `resolve_link` field with a pointer to a function called to resolve a textual link to a page number, and location on that page:

```
/*
   fz_document_resolve_link_fn: Type for a function to be called to
   resolve an internal link to a page number. See fz_resolve_link
   for more information.
*/
typedef int (fz_document_resolve_link_fn)(fz_context *ctx, fz_document
    *doc, const char *uri, float *xp, float *yp);
```

All document formats must fill in the `count_pages` field with a pointer to a function called to return the number of pages in a document:

```
/*
   fz_document_count_pages_fn: Type for a function to be called to
   count the number of pages in a document. See fz_count_pages for
   more information.
*/
typedef int (fz_document_count_pages_fn)(fz_context *ctx, fz_document
    *doc);
```

Different document formats encode different types of metadata. We therefore have an extensible function to allow such data to be queried. If your document handler wishes to support this, then the `lookup_metadata` field must be filled in with a pointer to a function to perform such lookups:

```
/*
    fz_document_lookup_metadata_fn: Type for a function to query
    a documents metadata. See fz_lookup_metadata for more
    information.
*/
typedef int (fz_document_lookup_metadata_fn)(fz_context *ctx,
     fz_document *doc, const char *key, char *buf, int size);
```

All document formats must fill in the `load_page` field with a pointer to a function called to return a reference to a `fz_page` structure:

```
/*
    fz_document_load_page_fn: Type for a function to load a given
    page from a document. See fz_load_page for more information.
*/
typedef fz_page *(fz_document_load_page_fn)(fz_context *ctx, fz_document
     *doc, int number);
```

To create a `fz_page` use the `fz_new_page` macro. For a document of type foo, typically a `foo_page` structure would be defined as below:

```
typedef struct
{
    fz_page super;
    <foo specific fields>
} foo_page;
```

This would then be created using a call to `fz_new_page`, such as:

```
   foo_page *foo = fz_new_page(ctx, foo_page);
```

This returns an empty document structure with `super` populated with default values, and the foo specific fields initialized to 0. The document handler implementation then needs to fill in the page level functions.

14.2.3 Page Level Functions

The `fz_page` structure contains a list of functions used to implement the page level calls:

```
typedef struct fz_page_s
{
    int refs;
    fz_page_drop_page_fn *drop_page;
    fz_page_bound_page_fn *bound_page;
    fz_page_run_page_contents_fn *run_page_contents;
    fz_page_load_links_fn *load_links;
    fz_page_first_annot_fn *first_annot;
    fz_page_page_presentation_fn *page_presentation;
    fz_page_control_separation_fn *control_separation;
    fz_page_separation_disabled_fn *separation_disabled;
    fz_page_count_separations_fn *count_separations;
    fz_page_get_separation_fn *get_separation;
} fz_page;
```

The fz_page (and hence derived foo_page) structures are reference counted. The refs field is used to keep the reference count in. All the reference counting is handled by the core library, and all that is required of the implementation is that it should supply a drop_page function that will be called when the reference count reaches zero. This is of type:

```
/*
    fz_page_drop_page_fn: Type for a function to release all the
    resources held by a page. Called automatically when the
    reference count for that page reaches zero.
*/
typedef void (fz_page_drop_page_fn)(fz_context *ctx, fz_page *page);
```

Implementations must fill in the bound_page field with the address of a function to return the pages bounding box, of type:

```
/*
    fz_page_bound_page_fn: Type for a function to return the
    bounding box of a page. See fz_bound_page for more
    information.
*/
typedef fz_rect *(fz_page_bound_page_fn)(fz_context *ctx, fz_page *page,
    fz_rect *);
```

Implementations must fill in the run_page_contents field with the address of a function to interpret the contents of a page, of type:

```
/*
    fz_page_run_page_contents_fn: Type for a function to run the
    contents of a page. See fz_run_page_contents for more
    information.
*/
typedef void (fz_page_run_page_contents_fn)(fz_context *ctx, fz_page
    *page, fz_device *dev, const fz_matrix *transform, fz_cookie
    *cookie);
```

If a document format supports internal or external hyperlinks, then its implementation must fill in the load_links field with the address of a function to load the links from a page, of type:

```
/*
   fz_page_load_links_fn: Type for a function to load the links
   from a page. See fz_load_links for more information.
*/
typedef fz_link *(fz_page_load_links_fn)(fz_context *ctx, fz_page *page);
```

If a document format supports annotations, then its implementation must fill in the `first_annot` field with the address of a function to load the annotations from a page, of type:

```
/*
   fz_page_first_annot_fn: Type for a function to load the
   annotations from a page. See fz_first_annot for more
   information.
*/
typedef fz_annot *(fz_page_first_annot_fn)(fz_context *ctx, fz_page
    *page);
```

Some document formats can encode information that specifies how pages should be presented to the user as a slideshow - how long they should be displayed, and which transition to use when moving to the next page etc. In implementations of document handlers for such formats, they should fill in the `page_presentation` field with the address of a function to obtain this information, of type:

```
/*
   fz_page_page_presentation_fn: Type for a function to
   obtain the details of how this page should be presented when
   in presentation mode. See fz_page_presentation for more
   information.
*/
typedef fz_transition *(fz_page_page_presentation_fn)(fz_context *ctx,
    fz_page *page, fz_transition *transition, float *duration);
```

Some document formats can encapsulate multiple color separations. In order to allow proofing of such formats, MuPDF allows such separations to be enumerated and enabled/disabled. In document handlers for such document formats, the `control_separation`, `separation_disabled`, `count_separations` and `get_separation` fields should be filled in with functions of the following types respectively:

```
/*
   fz_page_control_separation: Type for a function to enable/
   disable separations on a page. See fz_control_separation for
   more information.
*/
typedef void (fz_page_control_separation_fn)(fz_context *ctx, fz_page
    *page, int separation, int disable);
```

```
/*
   fz_page_separation_disabled_fn: Type for a function to detect
   whether a given separation is enabled or disabled on a page.
   See fz_separation_disabled for more information.
*/
typedef int (fz_page_separation_disabled_fn)(fz_context *ctx, fz_page
    *page, int separation);

/*
   fz_page_count_separations_fn: Type for a function to count
   the number of separations on a page. See fz_count_separations
   for more information.
*/
typedef int (fz_page_count_separations_fn)(fz_context *ctx, fz_page
    *page);

/*
   fz_page_get_separation_fn: Type for a function to retrieve
   details of a separation on a page. See fz_get_separation
   for more information.
*/
typedef const char *(fz_page_get_separation_fn)(fz_context *ctx, fz_page
    *page, int separation, uint32_t *rgb, uint32_t *cmyk);
```

14.3 Standard Document Handlers

MuPDF contains a range of document handlers for different formats. Which of these are built/enabled by default depends on configuration options in the `include/mupdf/fitz/config.h` file.

14.3.1 PDF

Support for PDF (Portable Document Format) is provided by `pdf_document_handler`. All current versions at the time of writing (i.e up to and including PDF 1.7) are supported.

MuPDF contains functionality to allow deeper access to the contents and structure of a PDF file than is exposed through the standard `fz_` prefixed functions, by using `pdf_` prefixed functions.

The library provides a `pdf_specifics` function to safely promote a `fz_document` pointer to a `pdf_document` pointer. This will return NULL if the document is not a PDF, indicating that the `pdf_` functions cannot be used.

14.3.2 XPS

Support for XPS (Open XML Paper Specification) is provided by `xps_document_handler`. All current versions at the time of writing are supported.

14.3.3 EPUB

Support for EPub v2 is provided by `epub_document_handler`. Tables are not currently supported, but is planned. Support for v3 is not planned.

The same document handler supports the FB2 (Fiction Book 2) electronic book format.

14.3.4 HTML

Support for basic HTML + simple CSS is provided by `htdoc_document_handler`. Tables are not currently supported, but is planned.

14.3.5 SVG

Support for SVG (Scalable Vector Graphics) is provided by `svg_document_handler`. Support is incomplete, but sufficient for many files.

14.3.6 Image

Support for a range of common image types (including PNG, JPEG, TIFF, JPEG2000, BMP and GIF) is provided by `image_document_handler`.

14.3.7 CBZ

Support for CBZ (Comic Book Archive) format is provided by `cbz_document_handler`. This supports files in .zip or .tar format.

Chapter 15

The Document Writer interface

15.1 Usage

As well as opening existing documents, MuPDF contains functions to allow the easy creation of new documents. The most general form of this functionality takes the form of the `fz_document_writer` interface.

A document writer is obtained by calling a generation function. The most general purpose one is:

```
/*
   fz_new_document_writer: Create a new fz_document_writer, for a
   file of the given type.

   path: The document name to write (or NULL for default)

   format: Which format to write (currently cbz, pdf, pam, pbm,
   pgm, pkm, png, ppm, pnm, svg, tga)

   options: NULL, or pointer to comma separated string to control
   file generation.
*/
fz_document_writer *fz_new_document_writer(fz_context *ctx, const char
    *path, const char *format, const char *options);
```

Alternatively, direct calls to generate specific document writers can be used, such as:

```
fz_document_writer *fz_new_cbz_writer(fz_context *ctx, const char *path,
    const char *options);
fz_document_writer *fz_new_pdf_writer(fz_context *ctx, const char *path,
    const char *options);
fz_document_writer *fz_new_svg_writer(fz_context *ctx, const char *path,
    const char *options);
fz_document_writer *fz_new_png_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_tga_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_pam_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_pnm_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_pgm_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_ppm_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_pbm_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
fz_document_writer *fz_new_pkm_pixmap_writer(fz_context *ctx, const char
    *path, const char *options);
```

Once a `fz_document_writer` has been created, pages can be written to the document one at a time. The process is started by calling `fz_begin_page`:

```
/*
   fz_begin_page: Called to start the process of writing a page to
   a document.

   mediabox: page size rectangle in points.

   Returns a fz_device to write page contents to.
*/
fz_device *fz_begin_page(fz_context *ctx, fz_document_writer *wri, const
    fz_rect *mediabox);
```

This function returns a `fz_device` pointer that should be used to write the page contents to. This can be done by making a sequence of normal device calls (see chapter 7 The Device interface) to paint the page with its content. One of the most common ways of doing this is by calling `fz_run_page_contents` on another open document. This therefore offers a quick mechanism for converting documents from one format to another.

Once the page contents have all been written, the page is finalized by calling `fz_end_page`:

```
/*
   fz_end_page: Called to end the process of writing a page to a
   document.
```

```
*/
void fz_end_page(fz_context *ctx, fz_document_writer *wri);
```

At this point, many formats will allow more pages to be written, simply by repeating the `fz_begin_page`, output, `fz_end_page` loop.

When all the pages have been written, the produced document can be finalized by calling `fz_close_document_writer`:

```
/*
   fz_close_document_writer: Called to end the process of writing
   pages to a document.

   This writes any file level trailers required. After this
   completes successfully the file is up to date and complete.
*/
void fz_close_document_writer(fz_context *ctx, fz_document_writer *wri);
```

Finally, the document writer itself can be freed in the usual fashion by calling `fz_drop_document_writer`:

```
/*
   fz_drop_document_writer: Called to discard a fz_document_writer.
   This may be called at any time during the process to release all
   the resources owned by the writer.

   Calling drop without having previously called drop may leave
   the file in an inconsistent state.
*/
void fz_drop_document_writer(fz_context *ctx, fz_document_writer *wri);
```

15.2 Implementation

Support for a new type of document writer requires a new structure, derived from `fz_document_writer`:

```
typedef struct
{
   fz_document_writer_begin_page_fn *begin_page;
   fz_document_writer_end_page_fn *end_page;
   fz_document_writer_close_writer_fn *close_writer;
   fz_document_writer_drop_writer_fn *drop_writer;
   fz_device *dev;
} fz_document_writer;
```

For instance:

```
typedef struct
{
    fz_document_writer super;
    <foo specific fields>
} foo_document_writer;
```

A generator function should be defined to return such an instance, perhaps:

```
fz_document_writer *fz_new_foo_document_writer(fz_context *ctx, const
     char *path, <foo specific params>) {
    foo_document_writer *foo = fz_new_derived_document_writer(ctx,
        foo_document_writer, foo_begin_page, foo_end_page, foo_close,
        foo_drop);

    <initialise foo specific fields>

    return &foo->super;
}
```

This uses a friendly macro that allocates a structure of the required size, initialises the function pointers as required, and zeroes the extra values in the structure.

```
/*
    fz_new_document_writer_of_size: Internal function to allocate a
    block for a derived document_writer structure, with the base
    structure's function pointers populated correctly, and the extra
    space zero initialised.
*/
fz_document_writer *fz_new_document_writer_of_size(fz_context *ctx,
     size_t size, fz_document_writer_begin_page_fn *begin_page,
    fz_document_writer_end_page_fn *end_page,
        fz_document_writer_close_writer_fn *close,
        fz_document_writer_drop_writer_fn *drop);

#define
     fz_new_derived_document_writer(CTX,TYPE,BEGIN_PAGE,END_PAGE,CLOSE,DROP)
     \
    ((TYPE
        *)Memento_label(fz_new_document_writer_of_size(CTX,sizeof(TYPE),BEGIN_PAGE,END_PAGE,CLOSE,DROP
```

The actual work for the document writer is done in the functions that are passed to `fz_new_derived_document_writer`. In the example above these were `foo_begin_page`, `foo_end_page`, `foo_close`, and `foo_drop`. These have the following 4 types respectively.

```
/*
    fz_document_writer_begin_page_fn: Function type to start
    the process of writing a page to a document.
```

```
   mediabox: page size rectangle in points.

   Returns a fz_device to write page contents to.
*/
typedef fz_device *(fz_document_writer_begin_page_fn)(fz_context *ctx,
    fz_document_writer *wri, const fz_rect *mediabox);

/*
   fz_document_writer_end_page_fn: Function type to end the
   process of writing a page to a document.

   dev: The device created by the begin_page function.
*/
typedef void (fz_document_writer_end_page_fn)(fz_context *ctx,
    fz_document_writer *wri, fz_device *dev);

/*
   fz_document_writer_close_writer_fn: Function type to end
   the process of writing pages to a document.

   This writes any file level trailers required. After this
   completes successfully the file is up to date and complete.
*/
typedef void (fz_document_writer_close_writer_fn)(fz_context *ctx,
    fz_document_writer *wri);

/*
   fz_document_writer_drop_writer_fn: Function type to discard
   an fz_document_writer. This may be called at any time during
   the process to release all the resources owned by the writer.

   Calling drop without having previously called close may leave
   the file in an inconsistent state.
*/
typedef void (fz_document_writer_drop_writer_fn)(fz_context *ctx,
    fz_document_writer *wri);
```

Once defined, if this is intended to be a generally useful document writer, it should probably be hooked into `fz_new_document_writer`, where it can be selected by appropriate format and options strings.

Chapter 16

Progressive Mode

16.1 Overview

When used in the normal way, MuPDF requires the entirety of a file to be present before it can be opened. For some applications, this can be a significant restriction - for instance, when downloading a PDF file over a slow internet link, being able to view just the first page or two may be enough to know whether it is the correct file or not.

Normal PDF files require the end of the file to be present before file reading can begin, as this is where the 'trailer' lives (effectively the index for the entire file). In an effort to allow early display of the first page, Adobe (the originators of the PDF format) introduced the concept of a 'linearized' PDF file. This is a PDF file that, while constructed in accordance with the original specification, also has some extra information contained within the file to allow fast access to the first page. This information is known as the 'hint stream'. In addition, extra constraints are placed upon the ordering of data within the file in an effort to ensure that the first page will download quickly.

Unfortunately, Linearized PDF files are far from a panacea. The specification is overly-complex, unclear and consequently poorly supported in both readers and writers of the format. Even when implemented correctly, it is of limited use for pages other than the first one.

MuPDF therefore attempts to solve the problem using a combination of mechanisms, known together as "progressive mode". When run in this mode, MuPDF can not only take advantage of the linearization information (if present) in a file, but is also capable of directing the actual download mechanism used by a file. By controlling the order in which sections of a file are fetched, any page required can be viewed before the whole fetch is complete.

For optimum performance a file should be both linearized and be available over a byte-range supporting link, but benefits can still be had with either one of these alone.

Coupled with the ability to render pages ignoring (and detecting) errors, this means that 'rough renderings' of pages can be given even before all the content (such as images and fonts) for a page have been downloaded.

16.2 Implementation

MuPDF has made various extensions to its mechanisms for handling progressive loading. They rely on some special properties built into a type of `fz_stream` known as a 'progressive' stream.

16.2.1 Progressive Streams

At its lowest level MuPDF reads file data from a `fz_stream`, using the `fz_open_document_with_stream` call. The alternative entrypoint `fz_open_document` is implemented by calling this.

The PDF interpreter uses the `fz_lookup_metadata` call to check for its stream being progressive or not. Any non-progressive stream will be read as normal, with the system assuming that the entire file is present immediately.

If it is found to be progressive, another `fz_lookup_metadata` call is made to find out what the length of the stream will be once the entire file is fetched. An HTTP fetcher can know this by consulting the Content-Length header before any data has been fetched.

With this information MuPDF can decide whether a file is linearized or not. (Technically, knowing the length enables us to check with the length value given in a linearized object - if these differ, the assumption is that an incremental save has taken place, thus the file is no longer linearized.)

Other than supporting the required metadata responses, the key thing that marks a stream as being progressive, is that it will not block when attempting to read data it does not have. Instead, it will throw a `FZ_ERROR_TRYLATER` error. This particular error code will be interpreted by the caller as an indication that it should retry the parsing of the current objects at a later time.

When a MuPDF call is made on a progressive stream, such as `fz_open_document_with_stream`, or `fz_load_page`, the caller should be prepared to handle a `FZ_ERROR_TRYLATER` error as meaning that more data is required before it can continue. No indication is directly given as to exactly how much more data is required, but as the caller will be implementing the progressive `fz_stream` that it has passed into MuPDF to start with, it can reasonably be expected to figure out an estimate for itself.

With these mechanisms in place, a caller can repeatedly try to render each page in turn until it gets a successful result.

16.2.2 Rough renderings

Once a page has been loaded, if its contents are to be 'run' as normal (using e.g. `fz_run_page`) any error (such as failing to read a font, or an image, or even a content stream belonging to the page) will result in a rendering that aborts with an `FZ_ERROR_TRYLATER` error. The caller can catch this and display a placeholder instead.

If each pages data was entirely self-contained and sent in sequence this would perhaps be acceptable, with each page appearing one after the other. Unfortunately, the linearization procedure as laid down by Adobe does NOT do this: objects shared between multiple pages (other than the first) are not sent with the pages themselves, but rather AFTER all the pages have been sent.

This means that a document that has a title page, then contents that share a font used on pages 2 onwards, will not be able to correctly display page 2 until after the font has arrived in the file, which will not be until all the page data has been sent.

To mitigate against this, MuPDF provides a way whereby callers can indicate that they are prepared to accept an 'incomplete' rendering of the file (perhaps with missing images, or with substitute fonts).

Callers prepared to tolerate such renderings should set the '`incomplete_ok`' flag in the cookie, then call `fz_run_page` etc as normal. If a `FZ_ERROR_TRYLATER` error is thrown at any point during the page rendering, the error will be swallowed, the '`incomplete`' field in the cookie will become non-zero and rendering will continue. When control returns to the caller the caller can check the value of the '`incomplete`' field and know that the rendering it received is not authoritative.

16.2.3 Directed downloads

If the caller has control over the fetch of the file (be it http or some other protocol), then it is possible to use byte range requests to fetch the document 'out of order'. This enables non-linearized files to be progressively displayed as they download, and fetches complete renderings of pages earlier than would otherwise be the case. This process requires no changes within MuPDF itself, but rather in the way the progressive stream learns from the attempts MuPDF makes to fetch data.

Consider for example, an attempt to fetch a hypothetical file from a server.

- The initial http request for the document is sent with a "Range:" header to pull down the first (say) 4k of the file.
- As soon as we get the header in from this initial request, we can respond to meta stream operations to give the length, and whether byte requests are accepted.
  - If the header indicates that byte ranges are acceptable the stream proceeds to go into a loop fetching chunks of the file at a time (not necessarily in-order). Otherwise the server will ignore the Range: header, and just serve the whole file.
  - If the header indicates a content-length, the stream returns that.
- MuPDF can then decide how to proceed based upon these flags and whether the file is linearized or not. (If the file contains a linearized object, and the content length matches, then the file is considered to be linear, otherwise it is not).

  If the file is linear:

  - We proceed to read objects out of the file as it downloads. This will provide us the first page and all its resources. It will also enable us to read the hint streams (if present).
  - Once we have read the hint streams, we unpack (and sanity check) them to give us a map of where in the file each object is predicted to live, and which objects are required for each page. If any of these values are out of range, we treat the file as if there were no hint streams.
  - If we have hints, any attempt to load a subsequent page will cause MuPDF to attempt to read exactly the objects required. This will cause a sequence of seeks in the `fz_stream` followed by reads. If the stream does not have the data to satisfy that request yet, the stream code should remember the location that was fetched (and fetch that block in the background so that future retries will succeed) and should raise an `FZ_ERROR_TRYLATER` error.

    [Typically therefore when we jump to a page in a linear file on a byte request capable link, we will quickly see a rough rendering, which will improve fairly fast as images and fonts arrive.]
  - Regardless of whether we have hints or byte requests, on every `fz_load_page` call MuPDF will attempt to process more of the stream (that is assumed to be being downloaded in the background). As linearized files are guaranteed to have pages in order, pages will gradually become available. In the absence of byte requests and hints however, we have no way of getting resources early, so the renderings

for these pages will remain incomplete until much more of the file has arrived.

[Typically therefore when we jump to a page in a linear file on a non byte request capable link, we will see a rough rendering for that page as soon as data arrives for it (which will typically take much longer than would be the case with byte range capable downloads), and that will improve much more slowly as images and fonts may not appear until almost the whole file has arrived.]

– When the whole file has arrived, then we will attempt to read the outlines for the file.

For a non-linearized PDF on a byte request capable stream:

– MuPDF will immediately seek to the end of the file to attempt to read the trailer. This will fail with a `FZ_ERROR_TRYLATER` due to the data not being here yet, but the stream code should remember that this data is required and it should be prioritized in the background fetch process.

– Repeated attempts to open the stream should eventually succeed therefore. As MuPDF jumps through the file trying to read first the xrefs, then the page tree objects, then the page contents themselves etc, the background fetching process will be driven by the attempts to read the file in the foreground.

[Typically therefore the opening of a non-linearized file will be slower than a linearized one, as the xrefs/page trees for a non-linear file can be 20%+ of the file data. Once past this initial point however, pages and data can be pulled from the file almost as fast as with a linearized file.]

For a non-linearized PDF on a non-byte request capable stream:

– MuPDF will immediately seek to the end of the file to attempt to read the trailer. This will fail with a `FZ_ERROR_TRYLATER` due to the data not being here yet. Subsequent retries will continue to fail until the whole file has arrived, whereupon the whole file will be instantly available.

[This is the worst case situation - nothing at all can be displayed until the entire file has downloaded.]

16.2.4 Example implementation

An example implementation of a fetcher process can be found in `curl-stream.c`. This implements a `fz_stream` using the popular 'curl' http fetching library.

The structure of this process broadly behaves as follows:

- We consider the file as an (initially empty) buffer which we are filling by making requests. In order to ensure that we make maximum use of our download link, we ensure that whenever one request finishes, we immediately launch another. Further, to avoid the overheads for the request/response headers being too large, we may want to divide the file into ‘chunks’, perhaps 4 or 32k in size.
- We have a receiver thread that sits there in a loop requesting chunks to fill this buffer. In the absence of any other impetus the receiver should request the next chunk of data from the file that it does not yet have, following the last fill point. Initially we start the fill point at the beginning of the file, but this will move around based on the requests made of the progressive stream.
- Whenever MuPDF attempts to read from the stream, we check to see if we have data for this area of the file already. If we do, we can return it. If not, we remember this as the next ‘fill point’ for our receiver process and throw a `FZ_ERROR_TRYLATER` error.
- The caller process is responsible for implementing the fetcher, hence it can know when more data has arrived. This can trigger retries of renderings intelligently, thus avoiding retrying renders when the incoming data is stalled.