MuPDF Explored

Robin Watts

May 24, 2021

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 1.13) 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.

Acknowledgements

Many thanks are due to Tor Andersson for creating MuPDF, to everyone who has contributed to it over the years, and to all my colleagues at Artifex Software for providing an environment in which it could grow, and nursing it through to maturity. Particular thanks are due to Sebastian Rasmussen for patiently proof-reading the book through its many revisions, and suggesting numerous improvements.

Finally, many thanks are due to Helen Rogers, for putting up with me.

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.

Often the first place that people will encounter MuPDF is as a Linux or Android desktop viewer, but these are merely simple examples of applications built using some of the features of the library.

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 Affero General Public 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 (or a compatible license[1]).
- 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 (or make available as "Software as a service").

[1] The question of whether a given open source license is compatible with the GNU AGPL is a complex one - a good list can be found here: https://www.gnu.org/licenses/license-list.en.html#GPLCompatibleLicenses

1.3 Dependencies

The core MuPDF library makes use of various software libraries.

Extract Document content extraction library.

FreeGLUT A free-software/open-source alternative to the OpenGL Utility Toolkit (GLUT) library.

Freetype Renderer for various font types.

Gumbo A HTML5 parser, designed to serve as a building block for other tools.

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

JBig2dec Image decoder for JBIG2 images.

JpegLib Image decoder for JPEG images.

LCMS2 ICC Color Management engine.

Leptonica A library broadly useful for image processing and image analysis applications.

libpng Image decoder for PNG images.

libtiff Image decoder for TIFF images.

MuJS Javascript engine used for PDF files.

OpenJPEG Image decoder for JPEG2000 images.

tesseract A neural net (LSTM) based OCR engine which is focused on line recognition.

ZLib Compression library.

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

OpenSSL Encryption library, required for Digital Signatures support.

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.

The exception to this is LCMS2. The version of LCMS2 included with MuPDF has API changes to make it incompatible with vanilla LCMS2. As such it is not merely a recommendation that you use the supplied version, but a requirement. The reasons for this incompatibility are discussed in the documentation with our version of the library.

Chapter 2

About this book

This book is divided into 3 parts.

The first part describes the MuPDF C API, the concepts behind it, and how to call it. If you wish to build an application from MuPDF, the information you need should be here.

The second part describes some of the modules used to build MuPDF. If you wish to extend MuPDF, perhaps to open new formats, or to offer new operations on documents once open, this is the part to refer to.

The third part describes some details of the actual language interpreters. This will primarily be of interest to people wanting to do low level operations on document formats (in particular PDF), but might be of interest to authors of new document format handlers to see how common problems have been addressed.

The fourth part describes some of the tools, libraries, and 'helper' routines provided with MuPDF. These helper routines are not strictly part of the MuPDF library, but can nonetheless be very useful when implementing applications based on it.

Finally, we have a part dedicated to platform specifics and the different language bindings available.

Part I

The MuPDF C API

Chapter 3

Quick Start

3.1 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`.

The concepts you meet in this example are explained and expanded upon in the following chapters. It may be useful to have this example to hand as you read on, to give you concrete illustrations of the ideas discussed.

Chapter 4

Naming Conventions

The function and variable names within MuPDF have been carefully chosen to follow standard conventions. By consistently using the same terms, we hope that it will become easy to remember things like the reference counting behaviour of functions, and minimise the need to look things up in the documentation.

We require any code submitted to MuPDF to follow the same conventions, and would encourage people to follow the same style in their own code that interfaces with MuPDF.

4.1 Prefixes

Historically, the graphics library upon which MuPDF relies was known as 'fitz'. Accordingly, all MuPDF's API calls and most types start with '`fz_`'.

The exception to this is where we have APIs and types provided by particular format handlers (e.g. PDF or XPS). These functions/types are prefixed with the format handlers name, for example '`pdf_`' or '`xps_`'.

All exported functions and types should be prefixed in this way to avoid the possibility of clashing with symbols in calling applications. Internal functions are not required to be prefixed, but we encourage it for clarity.

4.2 Naming

All functions are named according to one of the following schemes:

- verb_noun
- verb_noun_with_noun
- noun_attribute

- set_noun_attribute
- noun_from_noun – convert from one type to another (avoid noun_to_noun)

The sole exceptions to this are where we have MuPDF specific functions that emulate (or extend) 'well known' functions, where we parrot those functions. For example '`fz_printf`' or '`fz_strdup`'.

In addition, we avoid using 'get' in function names as this is generally redundant. In contrast, however, we do use 'set' where required. Consider for instance `fz_aa_level` (to retrieve the current anti-aliasing level), and `fz_set_aa_level` (to update it).

MuPDF makes extensive use of reference counting (see section 21.3 Reference Counting for more details). We reserve various words to indicate that reference counting is being used:

`new` Indicates that this call creates a new object and returns a reference to it. For example, `fz_new_pixmap`.

`find` Indicates that this call locates an object from somewhere (typically either from a cache, or from a set of standard objects), and returns a new reference to it. For example, `fz_find_color_converter`.

`load` Indicates that this call creates a new object and returns a reference to it. This is akin to 'new', but carries the implication that the operation will read some data and require some (possibly significant) amount of data processing before the object is created. For example, `fz_load_outline` or `fz_load_jpeg`.

`open` Indicates that this call will create a stream object, and return a reference to it. For example, `fz_open_document`.

`keep` Indicates that this call will create a new reference to an existing referenced object. For example, `fz_keep_colorspace`.

All of these calls return an object reference. It is the callers responsibility to store this reference safely somewhere for the duration of the required lifespan of the object, and to destroy that reference when the caller no longer requires it.

No function should return ownership of a reference without being named with one of the reserved words above.

Once all the references to a given object are released, the system will remove the object itself.

`drop` Indicates that this call will relinquish ownership of the object passed in. For example, `fz_drop_font`.

Failure to drop references will result in memory leaks. Dropping references too early may result in crashes due to objects being accessed after they have been destroyed.

API functions to destroy objects that are not subject to reference counted can be destroyed by calling functions using the words '`drop`', '`close`' or '`free`'.

In contrast to '`find`' described above, we have one other reserved word regarding searching:

`lookup` Indicates that the call will return a borrowed pointer (or a value). For example, `fz_lookup_pixmap_converter`.

When we have a structure already allocated, and we wish to initialise some or all of its internal details, we use '`init`'. The matching pair to this should be named '`fin`'. For example, `fz_cmm_init_profile` is matched by `fz_cmm_fin_profile`.

Some objects are created using functions using the verb '`create`'. Sometimes these can be reference counted objects (e.g. '`pdf_create_document`'), in which case they should be '`drop`'ped as usual. Non reference counted objects should be '`destroy`ed' (e.g. `fz_destroy_mutex`.

4.3 Types

Various different integer types are used throughout MuPDF.

In general:

- `int` is assumed to be at least 32 bits.
- `short` is assumed to be exactly 16 bits.
- `char` is assumed to be exactly 8 bits.
- array sizes, string lengths, and allocations are measured using `size_t`. `size_t` is 32bit in 32bit builds, and 64bit on all 64bit builds.
- buffers of data use `unsigned char`s (or `uint8_t`).
- Offsets within files/streams are represented using `int64_t` on all builds for simplicity.

Previously MuPDF used a `fz_off_t` type for file offsets. This changed size according to build type or `FZ_LARGEFILE` being defined, but this has been abandoned due to potential pitfalls. Having the API change between builds according to different symbol definitions is poor form as it can lead to unexpected crashes.

In addition, we use floats (and doubles internally, though we avoid these in the API where possible). These are assumed to be IEEE compliant.

Chapter 5

The Context

5.1 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 default levels of anti-aliasing 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 6 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 5.4 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 7 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 7 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 5.4 Multi-threading for more information.

5.2 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);
```

5.3 Custom Allocators

In some circumstances it can be desirable to force all allocations through a set of 'custom' allocators. These are defined as a `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 exception that they take an additional initial argument - that of the `user` value specified in the `fz_alloc_context`.

5.4 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 5.5 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.

5.5 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 stack. 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, are 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.

5.6 Destruction

Once you have finished with a `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);
```

5.7 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);
```

The purpose of allowing larger areas to be decoded than are immediately required, is so that these larger areas can be placed into the cache. This may mean that future requests can be satisfied from the cache rather than requiring complete new decodes. An example of such a situation might be where MuPDF is powering a viewer application, and a page is slowly panned onto screen revealing more and more of an image. These tuning functions put control over such decisions back into the hands of the application author.

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);
```

Essentially this routine allows the application author to exercise control over whether images are displayed with interpolation or not. Rather than simple linear interpolation, MuPDF uses the 'Mitchell' sampling function. This provides subjectively better quality.

The default is to use the Mitchell scaler only when downscaling, to avoid details 'dropping out' of images, but by providing a tuning function, the application author can choose to use it in more (or fewer) cases as desired.

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);
```

5.8 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 6

Error handling

6.1 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 a `fz_try`/`fz_always`/`fz_catch` construct.

Specific functions that never throw exceptions include all those named `fz_keep_...`, `fz_drop_...` and `fz_free`. This, coupled with the fact that all such 'destructor' functions will silently accept a `NULL` argument, makes the `fz_always` block an excellent place to clean up resources used throughout processing.

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. Try to avoid calling functions
     * that can themselves throw exceptions here, or the rest
     * of the fz_always block will be skipped - this is rarely
     * what is wanted! NEVER return from here. 'break' can be
     * used to continue execution in, or after the catch block
     * as appropriate. */
}
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. Unless the exception
     * is rethrown (or a fresh exception thrown), execution
     * continues after this block. */
}
```

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).

Similarly, you can `break` out of a `fz_always` block, and execution will correctly pass into or after the `fz_catch` block as appropriate, but this is less useful in practise.

The second one, is more convoluted. If you do not wish to understand the long and complex reasons behind this, skip the following subsection, and just read the corrected example that follows. As long as you follow the rules given in the summary at the end, you will be fine.

6.1.1 Why is fz_var necessary?

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)
    {
        /* Handle the error somehow. If we are nested within another
         * layer of fz_try/fz_catch, we can simply fz_rethrow. If
         * not, handle it in a way appropriate for this application,
         * perhaps by simply returning NULL. */
        return NULL;
    }
    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`. As part of this process, the house would take references to the walls and roof as required. Next we tidy up our local references to 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`? 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 the values of local variables made between a `setjmp` and a `longjmp` can be lost. (In fact, the C specification goes further than this, and says that such variables become 'undefined').

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);` before the `fz_try` 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 calls 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, and is guaranteed not to lose its value, whether an exception is thrown or not.

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.

6.1.2 Example: How to protect local variables with `fz_var`

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)
    {
        /* Handle the error somehow. If we are nested within another
         * layer of fz_try/fz_catch, we can simply fz_rethrow. If
         * not, handle it in a way appropriate for this application,
         * perhaps by simply returning NULL. */
        return NULL;
    }
    return h;
}
```

Note the calls to `fz_var`. These warn the compiler that it should take care not to lose updates to `w` or `r` if an exception is thrown in the `fz_try`. See Rule 5 in section 6.4 Summary below.

6.2 Throwing exceptions

Most client code need never worry about anything more than catching exceptions thrown by the 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_MEMORY = 1,
    FZ_ERROR_GENERIC = 2,
    FZ_ERROR_SYNTAX = 3,
    FZ_ERROR_TRYLATER = 4,
    FZ_ERROR_ABORT = 5,
```

```
    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);
```

`FZ_ERROR_MEMORY` is reserved for exceptions thrown due to a memory allocation failing. This will rarely be thrown by application code; the typical generators of such exceptions are `fz_malloc`/`fz_calloc`/`fz_realloc` etc.

`FZ_ERROR_SYNTAX` is reserved for exceptions thrown during interpretation of document files due to syntax errors. This enables the interpreter code to keep track of how many syntax errors have been found in a file, and to abort interpretation after a reasonable number have been passed.

`FZ_ERROR_TRYLATER` is reserved for exceptions thrown due to lack of data in progressive mode (see chapter 16 Progressive Mode for more details). Catching an error of this type can trigger different handling, whereby the operation is retried when more data has arrived.

`FZ_ERROR_ABORT` is reserved for exceptions that should stop any ongoing operations; for instance, while looping over the annotations on a page to render them, most exceptions are caught at the top level and ignored, ensuring that a single broken annotation doesn't cause subsequent annotations to be skipped. `FZ_ERROR_ABORT` can be used to override this behaviour and cause the annotation rendering process to end as swiftly as possible.

6.3 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 (from inside the `fz_catch` block) 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 6.2 Throwing exceptions for a list of the possible exception types.

For example, if an exception was thrown whilst attempting to render a page to a full page bitmap, it is entirely possible that this might be due to running out of memory. An application might reasonably decide to retry the render doing a strip at a time.

If, however, the render failed because of a corrupt file we'd gain nothing by retrying - hence the application should check the type of the exception to ensure it was a `FZ_ERROR_MEMORY` before trying the alternative technique.

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);
```

6.4 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 a `fz_try`/`fz_catch` construct.
2. A `fz_try` block must be paired with a `fz_catch` block, and optionally a `fz_always` block can appear between them.
3. Never return from a `fz_try` block.
4. A `fz_try` block will terminate when control reaches the end of the block, or when `break` is called.
5. Any local variable that is changed within a `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 7

Memory Management and The Store

7.1 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 re-decode 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 infeasible 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, but not so much that we start to run out for other needs. 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 (loading it, calculating it, or decoding it etc), 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.

7.2 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` is used to indicate that no amount of memory is too much.

7.3 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_MEMORY` type, enabling callers to easily distinguish them using `fz_caught` and to react accordingly.

7.3.1 Implementation

Further information on The Store can be found in chapter 21 Store Internals in Part II.

Chapter 8

The Document interface

8.1 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.

8.2 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 a fz_stream. See chapter 12 The Stream interface for more details on fz_streams.

```
/*
    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 a fz_stream; see chapter 12 The Stream interface for more details.

In common with most other objects in MuPDF, fz_documents 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.

8.3 Handling password protected documents

Some document types (such as PDF) can require passwords to allow the file to be opened. After you have obtained a 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.

    Returns 0 for failure to authenticate, non-zero for success.

    For PDF documents, further information can be given by examining
    the bits in the return code.

        Bit 0 => No password required
        Bit 1 => User password authenticated
        Bit 2 => Owner password authenticated

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

8.4 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 re-

quest.

The results of the layout will depend both upon a target width and height, a given font size, and the CSS styles in effect. MuPDF has an inbuilt set of default CSS styles that will be used if a document does not provide its own. In addition, the user can provide a final set that will override any rules found in the default sets. In this way, the appearance of the rendered document can be changed (perhaps by changing document colours or font styles/sizes).

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);
```

8.5 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.

8.6 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.

```
/*
    fz_keep_annot: Take a new reference to an annotation.
*/
fz_annot *fz_keep_annot(fz_context *ctx, fz_annot *annot);

/*
    fz_drop_annot: Drop a reference to an annotation. If the
    reference count reaches zero, annot will be destroyed.
*/
void fz_drop_annot(fz_context *ctx, fz_annot *annot);
```

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.

8.7 Color Considerations

Some formats, notably PDF, contain significant extra information to enable a high quality color managed workflow. The document interface (and the related page interface) have some methods to enable this.

Documents can have a defined 'output intent' that governs the color space (and profile) used for rendered output:

```
/*
    Find the output intent colorspace if the document has defined one.
*/
fz_colorspace *fz_document_output_intent(fz_context *ctx, fz_document
    *doc);
```

Callers will typically interrogate this before creating their output pixmaps if they want to honour it.

Each page in PDF can be authored with specific spot colors (inks) in mind. Details of these can be obtained from:

```
/*
    fz_page_separations: Get the separations details for a page.
    This will be NULL, unless the format specifically supports
    separations (such as PDF files). May be NULL even
    so, if there are no separations on a page.

    Returns a reference that must be dropped.
*/
fz_separations *fz_page_separations(fz_context *ctx, fz_page *page);
```

The returned object will be NULL for all document formats that do not support spot colors (at the time of writing, all but PDF). For PDF, the object will be NULL for all pages that do not make use of separations.

More information about using these objects can be found in section 9.5.2 Advanced Rendering - Overprint and Spots.

8.8 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 a `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 9 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);
```

These functions enable viewer applications to generate separate display lists for page contents and annotations. This can be useful if annotations are frequently changed, as it allows regeneration/redraw to happen on a per-annotation rather than per-page level.

All three of these functions (`fz_run_page`, `fz_run_page_contents`, `fz_run_annot`) take a `fz_cookie` pointer. The Cookie is a lightweight way of controlling the processing of the page. For more details, see section 9.3 Cookie. For most simple cases this can be `NULL`.

8.9 Presentations

Some file formats, such as PDF can be used as 'presentations', where pages are displayed as a slideshows - a form of poor man's PowerPoint if you will. Essentially each page contains a record that says how long it should be displayed for before transitioning to the next page with a given graphical effect.

The core MuPDF library is never responsible for actually presenting a page to the user, so it is therefore not possible to expect it to cope with handling all the work required by such transitions.

What it can do is to help in 2 particular areas. Firstly, it can provide some functions to aid the caller in the task of querying the transitions required. Secondly, it can help in providing some helper functions to generate bitmaps of various stages of common transitions.

8.9.1 Querying

We define a structure type to hold the details of arbitrary transitions, together with some opaque state:

```
enum {
    FZ_TRANSITION_NONE = 0, /* aka 'R' or 'REPLACE' */
    FZ_TRANSITION_SPLIT,
    FZ_TRANSITION_BLINDS,
    FZ_TRANSITION_BOX,
    FZ_TRANSITION_WIPE,
    FZ_TRANSITION_DISSOLVE,
    FZ_TRANSITION_GLITTER,
    FZ_TRANSITION_FLY,
    FZ_TRANSITION_PUSH,
    FZ_TRANSITION_COVER,
    FZ_TRANSITION_UNCOVER,
    FZ_TRANSITION_FADE
};

typedef struct fz_transition_s
{
    int type;
    float duration; /* Effect duration (seconds) */

    /* Parameters controlling the effect */
    int vertical; /* 0 or 1 */
    int outwards; /* 0 or 1 */
    int direction; /* Degrees */
    /* Potentially more to come */

    /* State variables for use of the transition code */
    int state0;
    int state1;
} fz_transition;
```

Armed with such a structure, we can call a function to get it filled out:

```
/*
    fz_page_presentation: Get the presentation details for a given page.

    transition: A pointer to a transition struct to fill out.

    duration: A pointer to a place to set the page duration in seconds.
    Will be set to 0 if no transition is specified for the page.

    Returns: a pointer to the transition structure, or NULL if there is
        no
    transition specified for the page.
*/
fz_transition *fz_page_presentation(fz_context *ctx, fz_page *page,
    fz_transition *transition, float *duration);
```

This structure is defined to be sufficient to encapsulate the currently defined

PDF transition types; it may be extended in future if other formats require more expressiveness.

Callers are free to directly implement their transitions using the information herein, or else they can make use of a helper function.

8.9.2 Helper functions

Details of a helpful routine for displaying some of these transitions can be found in chapter 37 Transitions.

Chapter 9

The Device interface

9.1 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 operations.

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.

9.2 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 will not describe these device functions here, but rather defer them to chapter 22 Device Internals in Part 2. While it is perfectly permissible for callers to call the device convenience functions themselves, the vast majority of application authors will never do so, and will simply treat each `fz_device` as a 'black box' to be passed to the interpretation functions (see section 8.8 Rendering Pages).

9.3 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.

9.3.1 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.

9.3.2 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.

9.3.3 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.

9.4 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.
*/
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.
*/
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_DONT_INTERPOLATE_IMAGES = 1,
```

```
    FZ_MAINTAIN_CONTAINER_STACK = 2,
    FZ_NO_CACHE = 4,
};
```

FZ_DONT_INTERPOLATE_IMAGES being enabled prevents the draw device performing interpolation. MuTool Draw uses this to inhibit interpolation when anti-aliasing is disabled. Finer control over this can now be given using the Tuning Context (see section 5.7 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.

9.5 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.

9.5.1 BBox Device

The BBox device is a simple device that calculates the bbox of all the marking operations[1] 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 when the device is closed with the bounding box for the contents.

[1] A marking operation is any graphical operation that causes a mark to appear on the page.

9.5.2 Draw Device

The Draw device is the core renderer for MuPDF. Every draw device instance is constructed with a destination Pixmap (see section 10.3 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.

Advanced Rendering - Overprint and Spots

Most formats define pages in terms of some fairly simple 'well known' colorspaces, like RGB and CMYK. Some formats (notably PDF) are much more powerful, and allow pages to be constructed with a range of non-standard 'spot' inks.

When combined with advanced features such as overprinting, care needs to be taken to ensure that the rendering is exactly as expected.

For example, if a PDF page is constructed to render a page using overprinting it only makes strict sense to render this to a CMYK (or a CMYK + Spots) pixmap. With (say) an RGB pixmap, CMYK colors would be mapped down to RGB as they are plotted, losing the information required to correctly overprint later graphical objects.

Nonetheless, while we might want to get a 'true' rendition of the page, we might require it ultimately to appear as an RGB pixmap. As such what we really want is to get a 'simulation' of how the overprint would work.

One way to work would be to call the draw device and request a CMYK + Spots rendering, and then to require the caller to convert this to their desired target colorspace manually. This is not in keeping with the general desire in MuPDF to encapsulate functionality in a friendly way.

Therefore, the draw device examines the 'separations' field of the pixmap that it is called with to decide how to render.

If there is no separations value supplied (i.e. it is NULL), then the draw device assumes that no form of overprint (or overprint simulation) is required.

If there is a separations value, and there is at least one separation that is not entirely disabled, then the draw device will draw internally to a CMYK + Spots pixmap (where the spots are the non-disabled separations from the separations value). This rendering can safely proceed with overprint processing enabled.

At the end of the render, the draw device will convert down from the CMYK + Spots pixmap to the colorspace of the initial pixmap. Any spot colorants present in the initial pixmap will be populated from the rendered one; any spots that aren't will be converted down to process colors.

Thus by creating the initial pixmap passed into the draw device using a separations object with the colorants correctly set to be composite/spots/disabled as required, overprint or overprint simulation can be controlled as required.

9.5.3 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 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);
```

For more details of the uses of Display Lists, see chapter 11 Display Lists.

9.5.4 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 chapter 15 The Document Writer interface) 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);
```

9.5.5 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.

9.5.6 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 chapter 15 The Document Writer interface) 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.

9.5.7 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. This can save a considerable amount of time, as it avoids the interpreter needing to run through an entire page when observation of one of the very first graphical operations is enough to know that colour is being used.

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.

9.5.8 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 10

Building Blocks

10.1 Overview

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

10.2 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 (normally 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 21.3 Reference Counting).

10.2.1 Basic Colorspaces

MuPDF contains a set of inbuilt colorspaces that cover most simple requirements. These are the 'device' colorspaces:

```
/*
    fz_device_gray: Get colorspace representing device specific gray.
*/
fz_colorspace *fz_device_gray(fz_context *ctx);

/*
    fz_device_rgb: Get colorspace representing device specific rgb.
*/
fz_colorspace *fz_device_rgb(fz_context *ctx);
```

```
/*
    fz_device_bgr: Get colorspace representing device specific bgr.
*/
fz_colorspace *fz_device_bgr(fz_context *ctx);

/*
    fz_device_cmyk: Get colorspace representing device specific CMYK.
*/
fz_colorspace *fz_device_cmyk(fz_context *ctx);

/*
    fz_device_lab: Get colorspace representing device specific LAB.
*/
fz_colorspace *fz_device_lab(fz_context *ctx);
```

10.2.2 Indexed Colorspaces

MuPDF allows for indexed colorspaces - those where a palette is used to select color values from a (normally) larger colorspace.

These are created using the `fz_new_indexed_colorspace` call:

```
fz_colorspace *fz_new_indexed_colorspace(fz_context *ctx, fz_colorspace
    *base, int high, unsigned char *lookup);
```

10.2.3 Separation and DeviceN Colorspaces

MuPDF Colorspaces are extensible, so specific document handlers can implement their own new spaces. A good example of this is how PDF implements Separation and DeviceN colorspaces.

These are special spaces which represent arbitrary sets of 1 or more colorants. These can either be mapped down to 'equivalent' colors in a more standard space, or (depending on the capabilities of the underlying device) processed in their raw form.

10.2.4 Further information

Further information on Colorspaces can be found within chapter 29 Colorspace Internals.

10.3 Pixmaps

10.3.1 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.

Pixmaps contain a set of n values per pixel, where $n = c+s+a$. c is the number of color components in the colorspace of a pixmap (or 0, if the colorspace is NULL). s is the number of spot colors in a pixmap (frequently 0). a is 0 if there is no alpha plane, and 1 otherwise.

The initial c entries are referred to as the 'process' color components. These can be either additive or subtractive dependent on the colorspace of the pixmap. Additive spaces (such as Gray, or RGB) have value 0 as dark, 255 as light. Subtractive spaces (such as CMYK) have value 0 as light (no ink), 255 as dark (full ink).

The next s entries are the spot colors represented by a pixmap. These are always in subtractive form.

The final entry (if $a = 1$) is the alpha value. This is 0 for completely transparent, 255 for completely opaque.

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...

A CMYK pixmap with spots for Orange and Green would have data of the form: CMYKOGCMYKOGCMYKOG...

To allow greater flexibility in the layout of the underlying memory blocks used by pixmaps, they 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 some cases (notably when we have pixmaps that represent a sub-rectangle of larger pixmaps) these 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.

10.3.2 Premultiplied alpha

By convention MuPDF holds pixmaps in 'premultipled alpha' form. This means that when an alpha plane is present, the values for the process and spot colors are stored scaled by the alpha value.

So for a pixel with R=G=B=1, with solid alpha, we'd have values of 255, but with an alpha value of 0.5 we'd have values of 127 stored.

This format is used because it simplifies many of the plotting and compositing operations used within MuPDF.

10.3.3 Saving

For information on saving pixmaps, see chapter 14 Rendered Output Formats.

10.4 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 10.5 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.

10.5 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.

10.6 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 a `fz_pixmap`) and (often) the original source data can be retrieved.

Further discussion of `fz_image`s is deferred to chapter 24 Image Internals in Part 2. While it is perfectly permissible for a caller to create images, in most cases they will treat them as 'black boxes' to just be passed around.

10.7 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.

10.8 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 'flip' the rectangle (i.e. that a minimum coordinate may end up as a maximum one after translation, and vice versa):

```
/*
    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);
```

10.9 Paths

Postscript (or equivalently 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".

Further discussion of paths is postponed to Part 2 in chapter 23 Path Internals. While is is entirely permissible for application callers to want to create their own paths (for passing to device functions) it's far more typical for callers to simply treat them as 'black boxes' to be passed around.

10.10 Text

MuPDFs central text type is a `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 extract 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.

While the idea of a single representation being enough to encapsulate each glyph from the text on the page in turn is attractive, this is not the case. Indeed, it's not even possible to trivially define the order in which glyphs will be sent!

It would be nice to think that text would be held in the source file in the order in which it should be displayed on the page, but this is frequently not the case.

The 'logical order' of text can be thought of as being the order in which text would be read out loud, if you were reading from the page. In many cases (such as for EPUB files), this is the order in which the information is stored within the file itself. Sadly, for other formats this is not always the case.

PDF files in particular have no particular defined ordering in which text is sent - as each glyph is individually positioned on the page, files can (and do) send them in any order they feel like. While most PDF files containing European languages will tend to send text in the expected logical ordering, there is no guarantee that this will always be the case. This likelihood gets even more remote as we start to deal with right-to-left text, top-to-bottom text, far eastern scripts, or texts in multiple different scripts or languages.

The classic case where logical order may differ noticeably from rendered order, is for 'bidirectional' text[1]. Even if the internal document representation is in logical order, the order in which the text will actually be displayed can be quite different. Consider, for instance, some source text in Hebrew. If the individual glyphs are A,B,C,D etc, then the right-to-left nature of Hebrew means that these will be displayed in the order 'DCBA' on the page.

[1] Text which has a mixture of left-to-right and right-to-left blocks

If, however, we have conventional western (arabic) numerals on the page, interspersed within the Hebrew text, this is still written left-to-right. So A,B,C,D,1,2,3,4,E,F,G would appear as 'GFE12DCBA'.

The algorithm dealing with such strings is fairly complex, and so further discussion of this for the interested reader is best redirected to the 'Unicode Bidirectional Algorithm' as defined in Technical Report 9 at http://unicode.org/reports/tr9.

The final dose of complexity comes from scripts that require 'shaping'. While simple western scripts (broadly) have a direct relationship between the character sent (e.g. the letter 'A') and the shape used to represent it on the page (e.g. the glyph 'A'), this does not hold true for all scripts.

The simplest example for this is that of a ligature. A piece of source text might contain the letter 'f' followed by the letter 'i' (perhaps in the word 'file'). When typeset onto a page, rather than displaying the glyphs individual, a combined glyph is generally used, 'fi'.

This concept of their being a 'transformation' step from the input text to the output rendered form is extended massively when dealing with non-western scripts. For Arabic and Indic scripts in particular (and Eastern scripts in general), groups of characters are frequently combined together to give increasingly complex glyphs. This process is referred to as shaping, and it is generally applied after the bidirectional algorithm has been run.

Different source formats cope with this in different ways. The text strings within a PDF file have already had the layout and shaping process applied - they are literally a list of positioned glyphs to be displayed on the page. Each glyph is identified by a 'glyph-id' - a simple index of the glyph within a font, with no meaning other than that. The Unicode values for the original text are frequently not there at all (and when they are they require specific work to derive).

Other formats, such as EPUB, take the opposite approach, by specifying the Unicode values directly, and leaving the displaying application (i.e. MuPDF) to do the conversion to glyphs (including the 'shaping' operation).

To cope with these different input requirements, and to allow us to translate one format into another, we require `fz_text` objects to encapsulate both forms of data at the same time.

Accordingly, our `fz_text` object represents a block of text, including font style and position, together with both Unicode and glyph data (subject to the availability of the information in the original file). Where possible we try to provide this information in logical order, though no guarantee can be made of this.

If more information is required, then details of the current implementation are included in chapter 25 Text Internals in Part 2, otherwise just use it as a simple black box.

10.11 Shadings

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.

Further discussion is deferred to chapter 26 Shading Internals in Part 2, as it would be unusable (though not inconceivable) for applications authors to want to author their own shadings.

Chapter 11

Display Lists

11.1 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.

11.2 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.

11.3 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);
```

11.4 Reference counting

In common with most other objects in MuPDF, `fz_display_list`s 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.

11.5 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);
```

Note that this is the bounding box of the page that was written to the display list, not the bounding box of the contents of the list; the latter will typically (but not always) be smaller than the former due to page borders etc.

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 12

The Stream interface

12.1 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.

12.2 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.

12.3 Usage

12.3.1 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);
```

12.3.2 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);
```

12.3.3 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);
```

This function skips as many bits as as required to align with a byte boundary.

12.3.4 Reading whole streams

As a convenience function, MuPDF provides a mechanism for reading the entire contents of a stream into a `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);
```

12.3.5 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
*/
int64_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, int64_t offset, int
    whence);
```

In the event that a stream does not support seeking, an error will be thrown.

As `fz_seek` and `fz_tell` work at byte granularity, care should be exercised when reading streams bitwise. Always `fz_sync_bits` before expecting `fz_tell` to give you a value that you can safely `fz_seek` back to.

12.3.6 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);
```

12.3.7 Destruction

In common with most other MuPDF objects, `fz_streams` 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.

Chapter 13

The Output interface

13.1 Overview

In the same way as `fz_streams` abstracts input streams, MuPDF uses a reusable class, `fz_output`, to abstract output streams.

13.2 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);
```

13.3 Usage

13.3.1 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);
```

13.3.2 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);
```

13.3.3 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.
*/
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);
```

13.3.4 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);
```

Chapter 14

Rendered Output Formats

14.1 Overview

MuPDFs built in renderer (see subsection 9.5.2 Draw Device) 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 a `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 use of a `fz_output` allows for writing to memory buffers, or even potentially to encrypt or compress further as the write progresses.

The third function returns a `fz_band_writer` to do the same thing.

14.2 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.

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);
```

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 spots and alphas).

    alpha: Number of alpha components.

    xres, yres: X and Y resolutions in dpi.

    pagenum: Page number

    cs: Colorspace (NULL for bitmaps)

    seps: Separation details (or NULL).

    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, const
     fz_colorspace *cs, fz_separations *seps);
```

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);
```

14.3 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);
```

14.4 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);
```

14.5 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);
```

14.6 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);
```

14.7 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 a `fz_image` or a `fz_pixmap` and produce a `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);
```

14.8 PSD

The PSD format is used by Photoshop. It is especially useful as it allows the inclusion of large numbers of spot colors, together with their 'equivalent color' information. This means it is the format of choice for outputting color correct renders with spot colors and correct overprint processing. PSD files can be written with grayscale, RGB or CMYK pixmaps, plus optional spots, with or without alpha.

```
/*
    fz_save_pixmap_as_psd: Save a pixmap as a PSD image file.
*/
void fz_save_pixmap_as_psd(fz_context *ctx, fz_pixmap *pixmap, const
     char *filename);

/*
    Write a pixmap to an output stream in PSD format.
*/
void fz_write_pixmap_as_psd(fz_context *ctx, fz_output *out, const
     fz_pixmap *pixmap);

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

14.9 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_pixmap`s (for contone output) and those that take `fz_bitmap`s (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);
```

14.9.1 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.

14.9.2 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.

14.10 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);
```

14.11 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);
```

14.11.1 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.

14.11.2 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);
```

14.12 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);
```

These are not generated by the band writer itself to allow a single output stream to be generated containing many pages, but a single file header and trailer.

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 9 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 entry point `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 a `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 a `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.

Chapter 17

Fonts

17.1 Overview

Fonts are represented in MuPDF by the abstract `fz_font` type. This reference counted structure, encapsulates the basic information about a font, specifically:

Glyph list Each font consists of a list of glyphs.

Glyph data How to draw each glyph. In traditional fonts this information is known as the 'Outline data' (or 'Outlines'), but some font types (such as Type 3 fonts from PDF) can encapsulate other data, such as images and colors too.

Unicode map Most (but not all) fonts contain information that enables glyphs to be mapped to/from the Unicode code points they represent. Without such information, it can be impossible to meaningfully extract text information from a document (such as for cut and paste).

Font BBox All fonts include information for a bounding box that covers all the glyphs within a font. Sadly this can frequently be inaccurate or incorrect, so should be treated with distrust.

Glyph advances All fonts contain simple Glyph advance information - how far to move the text cursor after having drawn a given glyph. This information ensures that successive characters are properly spaced w.r.t. each other.

Kerning data Most fonts contain simple kerning data; this allows for the glyph advance between any 2 glyphs to be adjusted based upon particular glyph values. The classic example of kerning is noting that the spacing between A and the left hand edge of its following letter is typically different between AV and AN.

Shape data Some fonts allow for the automatic 'shaping' of glyph sequences.

> The trivial example of this in western fonts is that the letters 'f' and 'i' can be combined into a single ligature glyph 'fi'. For many non Latin scripts (especially Indic and south east Asian scripts), this procedure happens to a far greater extent. This can be as simple as the incorporation of diacritical marks, or as complex as the complete rearrangement or replacement of glyph sequences to give different appearances on the final rendered page. This process is know as 'font shaping' and the data required to perform this is font specific, and is optionally encapsulated within fonts themselves.

The `fz_font` does not include information about the particular size that a font is used at on the page, nor the basic colour used to render a font. It is therefore typical to see `fz_font`s passed around the system paired with both a size (and/or transformation matrix), a colorspace, and color definition.

MuPDF uses Freetype to handle most of its font rendering. For Type 3 PDF fonts, it renders them itself. Font shaping is done using the HarfBuzz library.

17.2 Inbuilt Fonts

The exact set of fonts built in to any version of MuPDF is configurable (See chapter 18 Build configuration).

For PDF, we build in a basic set of fonts licensed from URW. Originally the PDF specification suggested that 14 standard fonts should be available on all readers, and that any other fonts should be embedded within files. This recommendation has since been updated to suggest that all fonts (or at least the required subset thereof) are embedded in all files. Nonetheless it is still common to find many PDF files that ask for fonts that are not embedded.

In order to cope with the widest range of scripts possible, MuPDF is supplied with (and can optionally include) a selection of the 'Noto' fonts from Google.

If this would take too much space on your target system, an alternative is to use the supplied DroidSansFallback font (also from Google) which is a good trade-off between size and coverage.

MuPDF does not, by default, make use of fonts present on the underlying system (for instance, on Windows, MuPDF will not look for fonts in C:/Windows/). Should you wish to implement this kind of 'system font' loading, however, MuPDF does provide hooks for this to be done. The `fz_install_load_system_font_funcs` call takes a set of function pointers that can be used for this purpose.

17.3 Implementation

The implementation details of `fz_font`s are still in flux, so it would be inappropriate to document them in any more detail at the moment.

Chapter 18

Build configuration

18.1 Overview

By default, MuPDF builds almost the most capable version of itself that it can. This can result in larger library sizes than are actually required for any given application. Some care has been taken in the design of MuPDF to try to allow linkers to intelligently drop sections of the code that are not actually required, but nonetheless it can still be worthwhile tuning builds.

18.2 Configuration file

To simply the task of tuning builds, all the configuration options have been gathered into a single header file, `include/mupdf/fitz/config.h`.

This file is in two halves; the top half consists of a sequence of pairs of comments, and commented out `#define`s, the second half consists of `#ifdef` logic to make sense of the `#define` values set.

The configuration options available at build time are therefore described by the pairs in the first half of the file.

Each one of these pairs has first a comment that describes what the following configuration options do, and then commented out `#define`s that sets the configuration options to their defaults.

If an integrator wants to change a build option from the default, they have two options. Firstly, and most simply, they can edit the first half of the file to set the required `#define`s by uncommenting the relevant line and editing it. Alternatively, they can arrange for these values to be predefined by the compiler, normally by editing the `CFLAGS` defined in the Makefile. This latter method has

the advantage of not requiring any edits to the source itself, and of allowing different configurations to be built from the same source tree.

The second half of the file should never need to be edited.

18.3 Plotter selection

The first section of the file deals with the plotters that are to be built into MuPDF.

```
/*
   Enable the following for spot (and hence overprint/overprint
   simulation) capable rendering. This forces FZ_PLOTTERS_N on.
*/
#define FZ_ENABLE_SPOT_RENDERING
```

If this is enabled (i.e. if spot rendering capabilities are required) then all the plotters in the following section are implicitly enabled. For people that do not need spot rendering (i.e. anyone dealing with on screen display where overprint simulation is not required) can turn this off, and further configure the next options:

```
/*
   Choose which plotters we need.
   By default we build the greyscale, RGB and CMYK plotters in,
   but omit the arbitrary plotters. To avoid building
   plotters in that aren't needed, define the unwanted
   FZ_PLOTTERS_... define to 0.
*/
/* #define FZ_PLOTTERS_G 1 */
/* #define FZ_PLOTTERS_RGB 1 */
/* #define FZ_PLOTTERS_CMYK 1 */
/* #define FZ_PLOTTERS_N 0 */
```

The plotter selection built into MuPDF will determine which formats can be rendered too. If, for instance, you are using MuPDF to target a greyscale only device (say, an e-ink screen for a greyscale ebook reader) then there is no need to ever render in color, and the RGB and CMYK options can be disabled.

Similarly, if you know you are targeting screen display, then the greyscale and CMYK plotters can be omitted.

The N plotters are generic plotters capable of coping with any color depth, but are not as optimised for rendering to G, RGB or CMYK as the specific sets of plotters for those depths. You may find that you can trade some speed for space by enabling the N plotters and disabling the other plotters.

If you wish to render to colorspaces other than G, RGB or CMYK, you must

enable the N plotters.

At least 1 plotter family must be defined in any build. If all the plotters are specifically disabled, then the N plotters will be enabled.

18.4 Document handlers

Support for the different document types supported by MuPDF is given by a set of document handlers (see chapter 20 The Document Handler interface). Each handler type can be enabled or disabled as required:

```
/*
    Choose which document handlers to include.
    By default all are enabled. To avoid building unwanted
    ones, define FZ_ENABLE_... to 0.
*/
/* #define FZ_ENABLE_PDF 1 */
/* #define FZ_ENABLE_XPS 1 */
/* #define FZ_ENABLE_SVG 1 */
/* #define FZ_ENABLE_CBZ 1 */
/* #define FZ_ENABLE_IMG 1 */
/* #define FZ_ENABLE_TIFF 1 */
/* #define FZ_ENABLE_HTML 1 */
/* #define FZ_ENABLE_EPUB 1 */
```

For instance to omit support for HTML, we would define `FZ_ENABLE_HTML` to be 0. Note that some document handlers share substantial parts of their code with other ones; for example the HTML and EPUB document ages share a large amount of code, so disabling just one of them will not make much difference to the overall code size.

18.5 JPEG 2000 support

Some security minded applications choose to disable JPEG2000 decoding. While MuPDF ships with the latest version of the JPEG2000 decoder we use, with all known security patches applied, it is possible to disable this entirely if required.

```
/*
    Choose whether to enable JPEG2000 decoding.
    By default, it is enabled, but due to frequent security
    issues with the third party libraries we support disabling
    it with this flag.
*/
/* #define FZ_ENABLE_JPX 1 */
```

Note that if this is disabled, MuPDF will no longer be able to decode all PDF files. If you are authoring your own PDF files (or you know that they do not

make use of JPEG2000, then disabling this will have no effect).

18.6 Javascript

PDF files can make use of Javascript for form data validation. If such interactive features are not required then the inclusion of an (albeit tiny) JavaScript engine serves no purpose. Accordingly, it can be disabled by setting `FZ_ENABLE_JS` to 0.

```
/*
    Choose whether to enable JavaScript.
    By default JavaScript is enabled both for mutool and PDF
        interactivity.
*/
/* #define FZ_ENABLE_JS 1 */
```

18.7 Fonts

By far the largest impact on MuPDFs size is given by choosing which fonts to include. Accordingly, there are a range of different options:

```
/*
    Choose which fonts to include.
    By default we include the base 14 PDF fonts,
    DroidSansFallback from Android for CJK, and
    Charis SIL from SIL for epub/html.
    Enable the following defines to AVOID including
    unwanted fonts.
*/
/* To avoid all noto fonts except CJK, enable: */
/* #define TOFU */

/* To skip the CJK font, enable: (this implicitly enables TOFU_CJK_EXT
    and TOFU_CJK_LANG) */
/* #define TOFU_CJK */

/* To skip CJK Extension A, enable: (this implicitly enables
    TOFU_CJK_LANG) */
/* #define TOFU_CJK_EXT */

/* To skip CJK language specific fonts, enable: */
/* #define TOFU_CJK_LANG */

/* To skip the Emoji font, enable: */
/* #define TOFU_EMOJI */

/* To skip the ancient/historic scripts, enable: */
```

```
/* #define TOFU_HISTORIC */

/* To skip the symbol font, enable: */
/* #define TOFU_SYMBOL */

/* To skip the SIL fonts, enable: */
/* #define TOFU_SIL */

/* To skip the Base14 fonts, enable: */
/* #define TOFU_BASE14 */
/* (You probably really don't want to do that except for measurement
    purposes!) */
```

If documents require a font that is not present, systems will try to 'fallback' to alternative ones. When this is not always successful (or indeed possible) unknown glyphs are often rendered as empty boxes, known informally in the typographic world as 'Tofu'.

By default MuPDF includes all the fonts it knows about. The configuration options are therefore a matter of choosing which scripts should instead be rendered as tofu.

Accordingly, to drop support for rendering emoji characters, you'd define `TOFU_EMOJI`.

The largest set of fonts are those for the wide range of worldwide scripts given by the Google Noto fonts. These can be omitted by defining `TOFU`.

Part II

MuPDF Internals

Chapter 19

The Image interface

19.1 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);
```

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 7 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 a `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_image`s 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.

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_image`s 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.

19.2 Standard Image Types

19.2.1 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 a `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 a `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.

19.2.2 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.

19.2.3 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.

19.3 Creating Images

To create an image from a standard type, simply call the appropriate function. For example, if you have a `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 a `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] (repeated 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);
```

19.4 Implementing an Image Type

Should it be necessary, support for new types of image can be implemented fairly simply, by defining a structure derived from a `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] (repeated 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`).

19.5 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 7 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 20

The Document Handler interface

20.1 Overview

MuPDF is written as an extensible framework for handling different document types. Each different document format provides a `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);
```

20.2 Implementing a Document Handler

20.2.1 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 a `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.

20.2.2 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.

20.2.3 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);
```

20.3 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. See chapter 18 Build configuration for more information.

20.3.1 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 2.0) 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.

20.3.2 XPS

Support for XPS (Open XML Paper Specification) is provided by `xps_document_handler`. All current versions at the time of writing are supported.

20.3.3 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.

20.3.4 HTML

Support for basic HTML + simple CSS is provided by `htdoc_document_handler`. Tables are not currently supported, but is planned.

20.3.5 SVG

Support for SVG (Scalable Vector Graphics) is provided by `svg_document_handler`. Support is incomplete, but sufficient for many files.

20.3.6 Image

Support for a range of common image types (including PNG, JPEG, TIFF, JPEG2000, BMP and GIF) is provided by `image_document_handler`.

20.3.7 CBZ

Support for CBZ (Comic Book Archive) format is provided by `cbz_document_handler`. This supports files in .zip or .tar format.

Chapter 21

Store Internals

21.1 Overview

In chapter 7 Memory Management and The Store we introduced the concept of the Store, and its use in getting the most out of the available memory of a system. Here we explain the implementation, so document handler authors (and application programmers) can make use of the same mechanism.

21.2 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, a `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.

21.3 Reference Counting

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

In the event that a `fz_new_...` call fails (perhaps due to running out of memory), then it will tidy up any partially constructed object(s) before throwing an exception.

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.

It is a matter of design that no `fz_drop_...` (or `fz_free`) call ever throws an exception. Furthermore, all such 'destructor' calls must accept a NULL pointer (and do nothing). This vastly simplifies error handling in most cases.

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

It is up to the developer to choose which size of storage to use for the reference count, remembering that the MuPDF counting routines do not detect overflow. If someone takes more than 127 references to an object built upon an `int8_t`, for example, the reference counting routines will believe that it is a static object, and it will never be freed.

21.4 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 5 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.

21.5 Using the Store

21.5.1 Overview

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

MuPDF uses reference counting for most of its data structures (see section 21.3 Reference Counting), and `fz_storable`s are no exception.

The objects in the Store are held in a chain according to when they were last used. Whenever an object is 'used', it is moved to the head of the chain. Whenever we need to evict an object from the Store to make room, we therefore discard objects from the tail of the chain. In this way frequently used objects are kept around, while rarely used ones are discarded in preference.

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 (i.e. it is moved to the front of the usage chain), 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.

When the `fz_context` (or, more accurately, the last of a set of cloned contexts) is finally destroyed, the Store is destroyed too. This results in every object in the Store being released. Unless something has gone wrong with reference counting, this will result in all our objects being freed.

21.5.2 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 a `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. Keys can be an amalgam of several reference counted objects, so a single call to the keep or drop functions provided here will take or release references for all these objects in one operation.

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.

21.5.3 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; /* 8 or 12 bytes */
        struct
        {
            const void *ptr;
            int i;
            fz_irect r;
        } pir; /* 24 or 28 bytes */
        struct
        {
            int id;
```

```
            float m[4];
        } im; /* 20 bytes */
        struct
        {
            unsigned char src_md5[16];
            unsigned char dst_md5[16];
            unsigned int ri:2;
            unsigned int bp:1;
            unsigned int bpp16:1;
            unsigned int proof:1;
            unsigned int src_extras:5;
            unsigned int dst_extras:5;
            unsigned int copy_spots:1;
        } link; /* 36 bytes */
    } u;
} fz_store_hash; /* 40 or 44 bytes */
```

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 hash table as required.

21.5.4 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.

The pixmaps can never be found by a search of the Store, because to find them, we'd have to search for them using the appropriate `fz_image`. They are therefore, to all intents and purposes 'dead', and just taking up useless space.

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 searching the Store.

21.5.5 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.

The need for a reap is detected as part of normal operations in the core code, and such passes are then triggered automatically as required. The user need never (and indeed cannot) trigger such passes manually. The user can, however, exercise some control over when such operations take place.

If an application is about to perform an operation that may drop many objects (say dropping a collection of cached display lists), then it should call `fz_defer_reap_start` beforehand, and match that with a `fz_defer_reap_end` afterwards. Any reap passes triggered by the dropping of objects within the display lists would be deferred until the end - resulting in at most one pass rather than potentially many.

Chapter 22

Device Internals

In chapter 9 The Device interface, we introduced the central concept of a `fz_device`, and described the normal ones found in a standard MuPDF build. We skipped over the actual implementation details of how to call, or implement such devices. We rectify that here.

As mentioned before, although each device offers a set of function pointers, we prefer people not to call these directly, but rather to call some convenience functions that map down onto these. This protects us against API changes in future, and copes automatically with NULL pointers (in the case when a device doesn't care about a particular type of call).

We describe these 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.

22.1 Line Art

Line Art is handled by the device functions to plot paths. See chapter 23 Path Internals 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, const fz_color_params
    *color_params);
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, const
    fz_color_params *color_params);
```

```
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);
```

22.2 Text

Text is handled by the device functions to plot text. See chapter 25 Text Internals 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, const fz_color_params *color_params);
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, const
    fz_color_params *color_params);
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 a `fz_pop_clip` is seen. See section 22.5 Clipping and Masking for more details.

Sometimes formats (such as PDF) can send text that has no rendered appearance. We refer to this as 'ignored' text. This serves a variety of purposes, the most usual of which is to allow text to be copy/paste out of a document when the actual appearance of that text is given in the form of an image or line art.

For example, a product logo may be rendered using vector graphics, which would ordinarily not have any textual meaning. By including some ignored text, then a user can copy the content out as text, or a text-to-speech engine can correctly enunciate the page contents.

An alternative common example would be where a document has been scanned, and then the text within it has been run through an OCR (Optical Character Recognition) process. The OCR engine would typically include its results as ignored text. The final document would look identical to the scans, but would copy/paste as expected.

22.3 Images

Images are handled by the device functions to plot images. See chapter 24 Image Internals for more information.

```
void fz_fill_image(fz_context *ctx, fz_device *dev, fz_image *image,
    const fz_matrix *ctm, float alpha, const fz_color_params
    *color_params);
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, const fz_color_params *color_params);
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 a `fz_pop_clip` is seen. See section 22.5 Clipping and Masking for more details.

22.4 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 chapter 26 Shading Internals for more details.

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

22.5 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 section 22.1 Line Art, section 22.2 Text, and section 22.3 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, const
    fz_color_params *color_params);
void fz_end_mask(fz_context *ctx, fz_device *dev);
```

22.6 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.

22.7 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 7 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.

The id is set by the caller - i.e. the interpreter for the document format in use. Care should be taken by the caller to make these unique.

```
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);
```

22.8 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 reason for using Render Flags rather than Device Hints (see section 9.4 Device Hints) is that Render Flags can be carried forward though display lists.

22.9 Device Color Spaces

Some pages redefine the basic default colorspaces (for color management purposes). This is handled at the device level using the `fz_set_default_colorspaces` call:

```
void fz_set_default_colorspaces(fz_context *ctx, fz_device *dev,
    fz_default_colorspaces *default_cs);
```

The device typically takes a reference to the `default_cs` object and refers to it as required.

22.10 Layers

Some file formats (such as PDF) have the concept of 'tagged content'. Graphical objects within the page stream can be tagged with information regarding how they correspond to each other on the page.

PDF files are rarely authored directly, but are typically 'distilled' from documents created in other programs (such as illustrator or similar packages). These other programs frequently have the concept of 'layers', and this layer information is generally carried over into the created PDF file using this tagging mechanism.

In fact, layers are such a ubiquitous source of tag information, that the PDF processing community (and some applications) frequently refers to such information as 'layer' information, even though this is an abuse of the terminology. MuPDF follows this convention.

Other interpreters may use layer information to explicitly encode genuine layers of course!

There is no render-time special treatment of layered content, so content is simply composited blindly onto the same output image. Indeed, layers do not necessarily even nest nicely with other content, and can cross groups etc in unpredictable ways.

As a document is interpreted, when a layer (or a tag) start or end is found, the device is signalled using the following calls:

```
void (*begin_layer)(fz_context *, fz_device *, const char *layer_name);
void (*end_layer)(fz_context *, fz_device *);
```

Not all content on a given layer may necessarily be sent at once - there may be several start/end pairs with the same tag.

As an example, consider a newspaper laid out in a desktop processor. Multiple 'stories' on the same page may be laid out each in multiple columns of content. The PDF file produced from this may tag each column with an identifier for the story from which it came, and the stories may be interleaved with one another.

By collating only the content tagged with a given identifier (layer name), individual stories may be able to be extracted from a PDF page.

The use of tagged content will vary from application to application, following convention, but no hard and fast rules.

Chapter 23

Path Internals

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.

23.1 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 Bézier) and curves (cubic Bézier).

```
/*
    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 Bézier) 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.

For details of exactly what each of these path segment types means, see "The PDF Reference Manual" or "The Postscript Language Reference Manual".

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);
```

23.2 Reference counting

As stated before, `fz_path`s 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);
```

23.3 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 section 23.7 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 11 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
    interchangeably 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.

23.4 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.

23.5 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 section 23.6 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);
```

23.6 Stroking

Where filling a path simply requires details of the fill to be used, stroking a path requires far more information; varying the thickness of the line, or the dash pattern, or linecaps/joins used can can radically alter its appearance. The details of these 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);
```

23.7 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 9.5.4 PDF Output Device or subsection 9.5.6 SVG Output Device) to convert paths to a new representation, and in the draw device (see subsection 9.5.2 Draw Device) for rendering.

To isolate callers from the implementation specifics of paths, MuPDF offers a mechanism to 'walk' a `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 the appropriate function pointer 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.

Chapter 24

Image Internals

The primary use of a `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 7 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 a `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_image`s 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.

24.1 Compressed Images

The `fz_compressed_image` structure is a specialisation of `fz_image`, that holds the source data for an image in a `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 a `fz_compressed_image` then it will return `NULL`.

24.2 Pixmap Images

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

Chapter 25

Text Internals

As described in Part 1 (section 10.10 Text), the central text object in MuPDF is `fz_text`. It represents blocks of bidirectional text, carrying (potentially) both details of the underlying Unicode characters, and the specific glyphs to be used to render them.

```
typedef struct fz_text_s fz_text;
```

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

25.1 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);
```

25.2 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);
```

25.3 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);
```

25.4 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);
```

25.5 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);
```

25.6 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_items`, 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_items` 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;
};
```

Chapter 26

Shading Internals

As described in section 10.11 Shadings, `fz_shade` is an encapsulation of the information required to define a PDF shading. This is essentially a superset of all the shading types provided by our supported document handlers. If we ever meet a format that requires features not provided in PDF, then `fz_shade` will be extended to cope.

```
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;
```

26.1 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:

26.2 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);
```

26.3 Painting

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

26.4 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 27

Stream Internals

The concepts embodied by a `fz_stream` object, and details of how to use them were given in chapter 12 The Stream interface. 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 a `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 28

Output Internals

The concepts embodied by a `fz_output` object, and details of how to use them were given in chapter 13 The Output interface. 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_streams`, 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.Many `fz_output` implementations rely on close being called to ensure the output is correctly flushed, and no data lost.

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 size_t (fz_output_tell_fn)(fz_context *ctx, void *state);
```

Chapter 29

Colorspace Internals

In section 10.2 Colorspaces, we were introduced to the basic Colorspaces available within MuPDF. Here we describe how they work internally, and how new colorspaces can be implemented by document handler authors.

Colorspaces are complicated slightly by the need to cope with both ICC-enabled and ICC-disabled workflows.

29.1 Non ICC-based Colorspaces

The first and most basic colorspace creation method is for creating a non-ICC based colorspace, by calling:

```
fz_colorspace *fz_new_colorspace(
                        fz_context *ctx,
                        const char *name,
                        enum fz_colorspace_type type,
                        int flags,
                        int n,
                        fz_colorspace_convert_fn *to_ccs,
                        fz_colorspace_convert_fn *from_ccs,
                        fz_colorspace_base_fn *base,
                        fz_colorspace_clamp_fn *clamp,
                        fz_colorspace_destruct_fn *destruct,
                        void *data,
                        size_t size);
```

The `name` parameter is a pointer to a (short) ASCII string describing the colorspace, and `n` is the number of colorants in the space.

The `type` of the colorspace should be chosen as one of the following types (which should be consistent with the value of `n` chosen):

```
enum fz_colorspace_type
{
    FZ_COLORSPACE_NONE,
    FZ_COLORSPACE_GRAY,
    FZ_COLORSPACE_RGB,
    FZ_COLORSPACE_BGR,
    FZ_COLORSPACE_CMYK,
    FZ_COLORSPACE_LAB,
    FZ_COLORSPACE_INDEXED,
    FZ_COLORSPACE_SEPARATION,
};
```

The `flags` value for is the logical or of a selection of values from the following enum:

```
enum
{
    FZ_COLORSPACE_IS_DEVICE = 1,
    FZ_COLORSPACE_IS_ICC = 2,
    FZ_COLORSPACE_IS_CAL = 4,
    FZ_COLORSPACE_LAST_PUBLIC_FLAG = 4,
};
```

MuPDF uses some extra bits internally, so unknown bits should be considered private.

If the colorspace requires any private data (perhaps a palette for an indexed space), then an opaque pointer can be passed as `data`. A function to destroy this data when the colorspace reference count reaches zero should be passed as `destruct`.

Colorspaces are placed into the Store, so some measure of the size of their data is required - the size in bytes of the colorspaces extra data should be passed as `size`.

If this colorspace is based on another one then a function to return a borrowed reference to the underlying space should be supplied as `base`. For instance, an indexed space with an RGB palette would pass a function that returns `fz_device_rgb`.

This leaves the 3 functions that do the heavy lifting.

The `to_ccs` parameter should be a function that takes `n` colorant values in the colorspace, and returns them as RGB values.

The `from_ccs` parameter should be a function that takes RGB colorant values, and returns them as `n` colorant values in the colorspace.

Finally, the `clamp` field should be a function that takes `n` colorant values in the colorspace, and clamps them into the appropriate range.

29.2 ICC-based colorspaces

If you have an ICC profile for your colorspace, then you can call one of these functions:

```
fz_colorspace *fz_new_icc_colorspace(
                              fz_context *ctx,
                              const char *name,
                              int num,
                              fz_buffer *buf);
fz_colorspace *fz_new_icc_colorspace_from_file(
                              fz_context *ctx,
                              const char *name,
                              const char *path);
fz_colorspace *fz_new_icc_colorspace_from_stream(
                              fz_context *ctx,
                              const char *name,
                              fz_stream *in);
```

They all load the ICC profile from the specified source, and create the ICC based `fz_colorspace`.

Unfortunately, in non-ICC based workflows, ICC profiles can't be loaded. This means that document handlers have to be prepared to 'fall back' to a non-ICC based approximation. This has to be done at the document handler level; for example the PDF agent calls `fz_get_cmm_engine`, and if it returns NULL, drops back to the alternate specified space.

29.3 Calibrated Colorspaces

One final route to create colorspaces exists, that of creating them from given calibration settings. This builds upon the ICC workflow, so again, should only be used in the ICC-workflow case.

```
fz_colorspace *fz_new_cal_colorspace(
                              fz_context *ctx,
                              const char *name,
                              float *wp,
                              float *bp,
                              float *gamma,
                              float *matrix);
```

Chapter 30

Color Management

30.1 Overview

MuPDF can optionally make use of a color management engine to offer a fully color managed workflow.

Its use of the engine is encapsulated within the `fz_cmm_engine` structure. Currently we provide an implementation of this structure using a modified version of LCMS2 (known as LCMS2MT), but systems with other CMM engines in already can use those instead by reimplementing the functions therein.

By default, on start-up, MuPDF has no color management engine enabled. This keeps the library size down (and performance up!) for people who do not wish to use it.

The relevant functions are:

```
/*
    fz_set_cmm_engine: Set the color management engine to
    be used. This should only ever be called on the "base"
    context before cloning it, and before opening any files.

    Attempting to change the engine in use once a file has
    been opened, or to use different color management engine
    for the same file in different threads will lead to
    undefined behaviour, including crashing.

    Using different ICC engines for different files using
    different sets of fz_contexts should theoretically be
    possible.
*/
void fz_set_cmm_engine(fz_context *ctx, const fz_cmm_engine *engine);
```

```
/*
    Currently we only provide a single color management
    engine, based on a (modified) LCMS2.

    An unmodified LCMS2 should work too, but only when restricted
    to a single thread.
*/
extern fz_cmm_engine fz_cmm_engine_lcms;
```

Thus to enable the color managed workflow using LCMS, call:

```
fz_set_cmm_engine(ctx, &fz_cmm_engine_lcms);
```

It is best to do this immediately after creating the base context.

It is possible to switch between ICC and non-ICC workflows in the same instance of MuPDF (and even to use the two simultaneously). It is, however, not generally possible to swap a given context once operations on that context have started. This is because any outstanding `fz_colorspace`s (such as those found within the store) will still refer to the wrong color management implementation!

Part III

The MuPDF Interpreters

Chapter 31

PDF Interpreter Details

31.1 Overview

The PDF document handler is built upon a large corpus of code within MuPDF that deals specifically with the objects, structures and operators found within a document. This code is collectively known as the PDF interpreter.

While it is perfectly possible to use MuPDF to open documents and render pages from a PDF file without understanding anything at all about the PDF interpreter, there are many situations where deeper access to the interpreter can be advantageous.

For example, in order for PDF documents to have their pages rearranged, or files embedded/extracted from them, simple access to the underlying PDF document structure is required.

In fact, the access given to a PDF document structure is such that almost any operation can be coded for.

31.2 PDF Document

The first step in dealing with a PDF document is to get a handle to it. This is done by opening it as normal using `fz_open_document` to get a `fz_document` pointer. To 'promote' this to a `pdf_document`, we use the `pdf_specifics` call:

```
/*
    pdf_specifics: down-cast a fz_document to a pdf_document.
    Returns NULL if underlying document is not PDF
*/
pdf_document *pdf_specifics(fz_context *ctx, fz_document *doc);
```

If `pdf_specifics` returns non-NULL, then you know that you are indeed dealing with a PDF format document.

Having a `pdf_document` pointer allows a series of new APIs to be called (see `include/mupdf/pdf/document.h`).

In terms of handling a PDF file via its constituent objects, one of the most useful is:

```
pdf_obj *pdf_trailer(fz_context *ctx, pdf_document *doc);
```

This obtains a pointer to a representation of the trailer dictionary object.

31.3 PDF Objects

PDF files are made up of a series of objects. These objects can be in many different types, including dictionaries, streams, numbers, booleans, names, strings etc. For full details, see 'The PDF Reference Manual'.

MuPDF represents all of these as a `pdf_obj` pointer. Such pointers are reference counted in the usual way:

```
pdf_obj *pdf_keep_obj(fz_context *ctx, pdf_obj *obj);
void pdf_drop_obj(fz_context *ctx, pdf_obj *obj);
```

Given such a pointer, the actual type of the object can be obtained using:

```
int pdf_is_null(fz_context *ctx, pdf_obj *obj);
int pdf_is_bool(fz_context *ctx, pdf_obj *obj);
int pdf_is_int(fz_context *ctx, pdf_obj *obj);
int pdf_is_real(fz_context *ctx, pdf_obj *obj);
int pdf_is_number(fz_context *ctx, pdf_obj *obj);
int pdf_is_name(fz_context *ctx, pdf_obj *obj);
int pdf_is_string(fz_context *ctx, pdf_obj *obj);
int pdf_is_array(fz_context *ctx, pdf_obj *obj);
int pdf_is_dict(fz_context *ctx, pdf_obj *obj);
int pdf_is_indirect(fz_context *ctx, pdf_obj *obj);
int pdf_is_stream(fz_context *ctx, pdf_obj *obj);
```

These all return non-zero if the object is of the tested type, and zero otherwise.

To extract the data from a PDF object, you can use one of the following functions:

```
/* safe, silent failure, no error reporting on type mismatches */
int pdf_to_bool(fz_context *ctx, pdf_obj *obj);
int pdf_to_int(fz_context *ctx, pdf_obj *obj);
fz_off_t pdf_to_offset(fz_context *ctx, pdf_obj *obj);
float pdf_to_real(fz_context *ctx, pdf_obj *obj);
```

```
char *pdf_to_name(fz_context *ctx, pdf_obj *obj);
char *pdf_to_str_buf(fz_context *ctx, pdf_obj *obj);
int pdf_to_str_len(fz_context *ctx, pdf_obj *obj);
```

It is, in fact, safe to call any of these functions on any `pdf_obj` pointer. If the object is not of the expected type, a 'safe' default will be returned.

31.3.1 Arrays

Array objects consist of lists of other objects, each of which can potentially be of a different type. Accordingly, we have a function to enquire how long a list we have:

```
int pdf_array_len(fz_context *ctx, pdf_obj *array);
```

Armed with this knowledge we can then fetch any object we want from within the array.

```
pdf_obj *pdf_array_get(fz_context *ctx, pdf_obj *array, int i);
```

Ideally `i` should be between 0 and length-1 (though the function will just return NULL if an out of range element is requested).

Note that the `pdf_obj` reference returned by this function is merely borrowed. That is to say, if you wish to keep the object pointer around for more than the immediate lifespan of the call, you should manually call `pdf_keep_obj` to keep it, and later `pdf_drop_obj` to dispose of it.

An object can be inserted into an array at a given index, using:

```
void pdf_array_insert(fz_context *ctx, pdf_obj *array, pdf_obj *obj, int
    index);
```

Any objects after this point are shuffled up the array. Alternatively an object can be put into an array at a given point, overwriting any object that is there already:

```
void pdf_array_put(fz_context *ctx, pdf_obj *array, int i, pdf_obj *obj);
```

If the array needs to be extended it will be, and any intervening objects will be created as 'null'. Alternatively objects can be appended to an array using:

```
void pdf_array_push(fz_context *ctx, pdf_obj *array, pdf_obj *obj);
```

In all these cases, the array will take new references to the object passed in - that is, after the call, both the array and the caller will hold references to the object. In cases where the object to be inserted is a 'borrowed' reference, this is ideal.

In other cases, where the ownership of the object reference should be passed down into the array, we have alternative formulations of those functions:

```
void pdf_array_insert_drop(fz_context *ctx, pdf_obj *array, pdf_obj
    *obj, int index);
void pdf_array_put_drop(fz_context *ctx, pdf_obj *array, int i, pdf_obj
    *obj);
void pdf_array_push_drop(fz_context *ctx, pdf_obj *array, pdf_obj *obj);
```

These functions are so named because they are equivalent to first inserting/putting/pushing the object, and then dropping it, with the nice side effect that any errors encountered during the push still result in the object being correctly dropped, often saving the caller from having to wrap the call in a `fz_try`/`fz_catch` clause.

31.4 PDF Operator Processors

Graphical content within a PDF file is given as streams of PDF "operators". These operators describe marking operations on a conceptual page. In order to display a PDF file the interpreter needs to run through these operators processing each in turn.

In addition, certain manipulations of PDF operations (like redaction, sanitisation and appending, for example) are best done by operating directly on these operators streams. The alternative scheme, of first converting the operators to graphical objects, then resynthesising an operator stream from that leads to problems with round trip conversions, and the potential loss of structure.

For this reason, the PDF interpreter within MuPDF is structured around an extensible class of `pdf_processors`. A `pdf_processor` is a set of functions, one for each operator. The interpreter runs through the operators and handles them by calling the appropriate functions.

By changing the `pdf_processor` in use, we can therefore change what the effect of interpreting the page is.

MuPDF contains three different `pdf_processor` implementations, though the system is deliberately open ended, and more can be supplied by any user of the library. Some can even be chained together in powerful ways.

31.4.1 Run processor

The first, and most commonly used processor is the `pdf_run_processor`. This processor has the effect of interpreting the incoming operators and turning them into device calls (i.e. graphical objects rendered on a page).

When using the standard `fz_run_page` (and similar) function(s) this is the `pdf_processor` that is used automatically. It can still be useful to create these

manually, especially when coupling them with a `pdf_filter_processor` (or similar).

Such processors can be created using:

```
/*
    pdf_new_run_processor: Create a new "run" processor. This maps
    from PDF operators to fz_device level calls.

    dev: The device to which the resulting device calls are to be
    sent.

    ctm: The initial transformation matrix to use.

    usage: A NULL terminated string that describes the 'usage' of
    this interpretation. Typically 'View', though 'Print' is also
    defined within the PDF reference manual, and others are possible.

    gstate: The initial graphics state.

    nested: The nested depth of this interpreter. This should be
    0 for an initial call, and will be incremented in nested calls
    due to Type 3 fonts.
*/
pdf_processor *pdf_new_run_processor(fz_context *ctx, fz_device *dev,
    const fz_matrix *ctm, const char *usage, pdf_gstate *gstate, int
    nested);
```

The component parts of this processor are generally functions named `pdf_run_...`, and frequently call back into the main pdf interpreter (to handle nested content streams as found in XObjects etc).

31.4.2 Filter processor

The `pdf_filter_processor` is an example of a processor that allows chaining. PDF operators are fed into the processor, which then 'filters' them and passes them out to another processor.

```
/*
    pdf_new_filter_processor: Create a filter processor. This
    filters the PDF operators it is fed, and passes them down
    (with some changes) to the child filter.

    The changes made by the filter are:

    * No operations are allowed to change the top level gstate.
    Additional q/Q operators are inserted to prevent this.

    * Repeated/unnecessary colour operators are removed (so,
```

```
    for example, "0 0 0 rg 0 1 rg 0.5 g" would be sanitised to
    "0.5 g")

    The intention of these changes is to provide a simpler,
    but equivalent stream, repairing problems with mismatched
    operators, maintaining structure (such as BMC, EMC calls)
    and leaving the graphics state in an known (default) state
    so that subsequent operations (such as synthesising new
    operators to be appended to the stream) are easier.

    The net graphical effect of the filtered operator stream
    should be identical to the incoming operator stream.

    chain: The child processor to which the filtered operators
    will be fed.

    old_res: The incoming resource dictionary.

    new_res: An (initially empty) resource dictionary that will
    be populated by copying entries from the old dictionary to
    the new one as they are used. At the end therefore, this
    contains exactly those resource objects actually required.

*/
pdf_processor *pdf_new_filter_processor(fz_context *ctx, pdf_processor
    *chain, pdf_obj *old_res, pdf_obj *new_res);
```

Similar filtering processors could be written for other tasks, such as discarding all the text from a page, changing all occurrences of a particular font for another, or converting all the objects on a page to a given colorspace.

The component parts of this processor are generally functions named `pdf_filter_...`.

31.4.3 Buffer processor

The `fz_buffer_processor` is designed to produce a `fz_buffer` from an input stream of operators. This is frequently found coupled with a `fz_filter_processor`, to gather up the filtered version of the operator stream ready for reinsertion into the document.

```
/*
    pdf_new_buffer_processor: Create a buffer processor. This
    collects the incoming PDF operator stream into an fz_buffer.

    buffer: The (possibly empty) buffer to which operators will be
    appended.

    ahxencode: If 0, then image streams will be send as binary,
```

```
    otherwise they will be asciihexencoded.
*/
pdf_processor *pdf_new_buffer_processor(fz_context *ctx, fz_buffer
    *buffer, int ahxencode);
```

This is built using a `fz_output_processor`.

31.4.4 Output processor

The `fz_output_processor` is designed to produce an output stream from an input stream of operators. This is frequently found coupled with a `fz_filter_processor`, to gather up the filtered version of the operator stream ready for reinsertion into the document.

```
/*
    pdf_new_output_processor: Create an output processor. This
    sends the incoming PDF operator stream to an fz_output stream.

    out: The output stream to which operators will be sent.

    ahxencode: If 0, then image streams will be send as binary,
    otherwise they will be asciihexencoded.
*/
pdf_processor *pdf_new_output_processor(fz_context *ctx, fz_output *out,
    int ahxencode);
```

The component parts of this processor are generally functions named `pdf_out_...`.

31.5 Copying objects between PDF documents

PDF objects vary in complexity from simple values (booleans, integers, floats, names, etc) to more complex entities (arrays, dictionaries, streams, indirect references etc). While the simplest object types are independent of any particular document, the more complex types are implicitly bound to the document in which they appear.

Most of the time this is all taken care of automatically by the MuPDF core, but special care must be taken when trying to copy objects from one PDF file to another.

31.5.1 The problem

To illustrate this, imagine that you have 2 PDF documents open, docA and docB. Imagine that we want to lookup an object from docA, and insert into docB. A naive code fragment to do this might be:

```
pdf_dict_putp_drop(ctx,
                   pdf_trailer(ctx, docB),
                   "Root/Example",
                   pdf_dict_getp(ctx,
                                 pdf_trailer(ctx, docA),
                                 "Root/Example"));
```

This may actually work in limited cases, such as:

```
1 0 obj
<<
  /Type /Catalog
  /Pages 3 0 R
  /Metadata 9 0 R
  /Example true
>>
endobj
...
trailer
<<
  /Root 1 0 R
>>
```

The value of `Root/Example` is read as `true`, which can safely be written into another file.

This can easily fall down though, as can be seen in more complex cases:

```
2 0 obj
/Complex
endobj
1 0 obj
<<
  /Type /Catalog
  /Pages 3 0 R
  /Metadata 9 0 R
  /Example [ (More) 2 0 R ]
>>
endobj
...
trailer
<<
  /Root 1 0 R
>>
```

In this case the value of `Root/Example` is read as an array of 2 elements; the first element being the string "More", and the second being a reference to object 2 in the file.

If this was to be written directly into the new file, we'd still have an array of 2 elements, with the first element being the string "More". The second would refer to whatever object 2 in the new file happens to be.

The solution to this requires us to walk the directed (possibly cyclic) graph of child objects within the object to be copied from one file to another, and to 'deep copy' the contents.

We refer to this process as 'grafting' objects from one tree into another.

31.5.2 Grafting objects

To move a single object to a new tree, use `pdf_graft_object`:

```
/*
    pdf_graft_object: Return a deep copied object equivalent to the
    supplied object, suitable for use within the given document.

    dst: The document in which the returned object is to be used.

    obj: The object deep copy.

    Note: If grafting multiple objects, you should use a pdf_graft_map
    to avoid potential duplication of target objects.
*/
pdf_obj *pdf_graft_object(fz_context *ctx, pdf_document *dst, pdf_obj
    *obj);
```

This takes an object in one document, and returns an equivalent object that can safely be written into document `dst`. Any indirect references within the original object will have been copied across as new objects within `dst` as a side effect of this call.

The 'safe' version of the code given above would therefore be:

```
pdf_dict_putp_drop(ctx,
                   pdf_trailer(ctx, docB),
                   "Root/Example",
                   pdf_graft_object(ctx, docB,
                                    pdf_dict_getp(ctx,
                                                  pdf_trailer(ctx, docA,
                                                  "Root/Example"));
```

31.5.3 A further problem

Even this is not perfect. Consider the example:

```
2 0 obj
/Complex
```

```
endobj
1 0 obj
<<
  /Type /Catalog
  /Pages 3 0 R
  /Metadata 9 0 R
  /Example [ (More) 2 0 R ]
  /Example2 [ (Even more) 2 0 R ]
>>
endobj
...
trailer
<<
  /Root 1 0 R
>>
```

Suppose we want to copy both `Root/Example` and `Root/Example2` between files. If we read the first of these, and write it, it will cause object 2 to be copied to the new file (as a new object, 99 say). When we read the second one, and write that, it will cause object 2 to be copied into the second file again (as object 100 perhaps).

In the example above, with the object consisting of a single name this duplication may not matter, but when you consider that objects might be dictionaries with lots of contents, or even streams with many megabytes of data attached, the problem becomes clear.

The solution to this is to use a `pdf_graft_map`.

31.5.4 Graft maps

A `pdf_graft_map` is a mapping from one `pdf_document` to another that ensures objects in the source document are only ever copied into the target document at most once.

```
/*
    pdf_new_graft_map: Prepare a graft map object to allow objects
    to be deep copied from one document to the given one, avoiding
    problems with duplicated child objects.

    dst: The document to copy objects to.

    Note: all the source objects must come from the same document.
*/
pdf_graft_map *pdf_new_graft_map(fz_context *ctx, pdf_document *dst);

/*
    pdf_drop_graft_map: Drop a graft map.
*/
```

```
void pdf_drop_graft_map(fz_context *ctx, pdf_graft_map *map);

/*
    pdf_graft_mapped_object: Return a deep copied object equivalent
    to the supplied object, suitable for use within the target
    document of the map.

    map: A map targeted at the document in which the returned
    object is to be used.

    obj: The object deep copy.

    Note: Copying multiple objects via the same graft map ensures
    that any shared child are not duplicated more than once.
*/
pdf_obj *pdf_graft_mapped_object(fz_context *ctx, pdf_graft_map *map,
    pdf_obj *obj);
```

A 'safe' version of the example given earlier that copies both `Root/Example` and `Root/Example2` would therefore be:

```
pdf_graft_map *map = pdf_new_graft_map(ctx, docB);

pdf_dict_putp_drop(ctx,
                   pdf_trailer(ctx, docB),
                   "Root/Example",
                   pdf_graft_mapped_object(
                                   ctx, map,
                                   pdf_dict_getp(ctx,
                                                 pdf_trailer(ctx, docA,
                                                 "Root/Example"));
pdf_dict_putp_drop(ctx,
                   pdf_trailer(ctx, docB),
                   "Root/Example2",
                   pdf_graft_mapped_object(
                                   ctx, map,
                                   pdf_dict_getp(ctx,
                                                 pdf_trailer(ctx, docA,
                                                 "Root/Example2"));
pdf_drop_graft_map(ctx, map);
```

Chapter 32

XPS Interpreter Details

32.1 Overview

The XPS document handler differs from the PDF document handler in that its sole interface is that exposed through the `fz_document` class.

XPS files are zip format archives with a specific layout of files. As well as opening xps (and .oxps) files (archives) directly, MuPDF will also open unpacked files by pointing `fz_open_document` at the "`rels/.rels`" file in the unpacked directory tree.

Chapter 33

EPub/HTML Interpreter Details

The EPUB and HTML document handlers are based upon the same layout code. With the exception of a few global configuration settings, their sole interfaces are those exposed through the `fz_document` class.

33.1 CSS rules

The layout follows simple CSS rules. On opening both EPUB and HTML documents CSS rules are read from a series of locations in defined order.

Firstly, an inbuilt set of CSS (see `html_default_css`) is read for all files with the exception of those identified as FictionBook 2 format which have their own special case rules (see `fb2_default_css`).

Next, the CSS rules from the document itself are read. If you would rather avoid this, it can be suppressed using:

```
/*
    fz_set_use_document_css: Toggle whether to respect document styles
        in HTML and EPUB.
*/
void fz_set_use_document_css(fz_context *ctx, int use);
```

Finally a set of 'user' CSS is read. This defaults to empty, but can be supplied using:

```
/*
    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);
```

Thus the user CSS is the last to be read and can potentially override all the settings made by the defaults and document CSS.

33.2 Shaped text

The text read from the document is held as Unicode, and displayed using the built in fonts. By default these are the Google Noto fonts, which are in OpenType format. One of the features of OpenType fonts is the ability to offer excellent typographical output for a wide range of scripts due to the inbuilt automated tables to control font shaping.

Font shaping allows a font to choose a different set of output glyphs (with highly customised positioning) based on the context within which an input Unicode character (or series of characters) are used.

Some languages use this to add diacritical marks (in particular Vietnamese). Others (such as Arabic) may use it to ensure that characters join smoothly. Still others (such as Indic languages) completely change the appearance of groups of input characters by combining them into single shapes that represent multiple characters at once.

The complex rules that control this are encoded as tables within the OpenType format fonts. The interpretation and application of these tables/rules is handled for us using the HarfBuzz library.

33.3 Bidirectional text

While Western languages are written left to right, others are written (broadly) right to left. Even in right to left languages, specific regions of text (such as numbers) are written left to right. The behaviour of 'enclosing' operators brackets adds additional complexity.

The rules governing exactly how layout should proceed when faced with a combination of left to right and right to left text are complex to say the least, but fortunately an algorithm has been published by the Unicode consortium to specify exactly how layout should proceed in any given circumstance (as 'Technical Recommendation 9').

MuPDF contains an implementation of this algorithm derived from the example code provided with this recommendation.

Chapter 34

SVG Interpreter Details

The SVG interpreter offers very little API other than that exposed through the `fz_document` interface.

The sole extras are defined to facilitate the use of SVGs as illustrations within EPUB files.

```
/*
    Parse an SVG document into a display-list.
*/
fz_display_list *fz_new_display_list_from_svg(fz_context *ctx, fz_buffer
    *buf, float *w, float *h);

/*
    Create a scalable image from an SVG document.
*/
fz_image *fz_new_image_from_svg(fz_context *ctx, fz_buffer *buf);
```

These enable the HTML agent to easily create a `fz_image` out of an SVG. This `fz_image` has the property that it remains scalable, and hence will not appear pixellated if the document is reflowed to different dimensions.

Part IV

Tools, Libraries, and Helper Routines

Chapter 35

MuTool

35.1 Overview

Mutool is a collection of useful command line utilities rolled into a single executable.

As explained earlier MuPDF is a C library that encapsulates all the smarts required to open/render/manipulate document files of a range of a formats. This means that most utilities using it are reduced to very thin shells that just call down to the library.

When you consider the additional factor of the size of the resources built into MuPDF (fonts, CMAPs etc), this means it makes a lot of sense to build a single executable with multiple utilities sharing a single copy of the library.

If you run `mutool` at the command line with no arguments, a list of the possible options will be displayed:

```
$ mutool
usage: mutool <command> [options]
       clean     -- rewrite pdf file
       convert   -- convert document
       create    -- create pdf document
       draw      -- convert document
       extract   -- extract font and image resources
       info      -- show information about pdf resources
       merge     -- merge pages from multiple pdf sources into a new pdf
       pages     -- show information about pdf pages
       portfolio -- manipulate PDF portfolios
       poster    -- split large page into many tiles
       run       -- run javascript
       show      -- show internal pdf objects
```

35.2 Clean

The `clean` utility will produce a cleaned version of an input PDF. It can apply a range of different options, a full list of which can be obtained by running `mutool clean` with no options:

```
$ mutool clean
usage: mutool clean [options] input.pdf [output.pdf] [pages]
	-p -	password
	-g	garbage collect unused objects
	-gg	in addition to -g compact xref table
	-ggg	in addition to -gg merge duplicate objects
	-gggg	in addition to -ggg check streams for duplication
	-l	linearize PDF
	-a	ascii hex encode binary streams
	-d	decompress streams
	-z	deflate uncompressed streams
	-f	compress font streams
	-i	compress image streams
	-s	clean content streams
	pages	comma separated list of page numbers and ranges
```

The arguments here are fairly self explanatory, and usage is best explained with a few examples.

Firstly, and most simply, `clean` can be used to try to repair broken files. Many PDF files found in the wild are broken - sometimes because of having been corrupted, either by transmission/archiving problems, but a disappointing amount by just having been created by bad PDF writing software. Running a clean pass will attempt to repair the files:

```
mutool clean in.pdf out.pdf
```

Individual pages (or page ranges) can be extracted from a PDF. For example:

```
mutool clean -gggg in.pdf out.pdf 1-10,12
```

That will extract the pages 1 to 10, and page 12 of `in.pdf` and output it into a new `out.pdf`. The `-gggg` options ensure that unused objects will be dropped from the PDF.

An 8 page PDF might be rearranged into booklet form using:

```
mutool clean -gggg in.pdf out.pdf 8,1,7,2,6,3,5,4
```

Finally, a more exotic, but very common example; if someone reports a problem seen on page 4 of a given PDF, the following command will extract that page, and expand the content streams, without decompressing the images or the fonts:

```
mutool clean -difgggg in.pdf out.pdf 4
```

If this file still exhibits the same problem, it is generally far easier to debug through it than the original one was.

35.3 Convert

The `convert` utility performs a similar task to the `draw` utility, using a different internal mechanism (the document writer interface). Which is better for any given task is often a matter of taste.

```
$ mutool convert
mutool convert version 1.11
Usage: mutool convert [options] file [pages]
	-p -	password

	-A -	number of bits of antialiasing (0 to 8)
	-W -	page width for EPUB layout
	-H -	page height for EPUB layout
	-S -	font size for EPUB layout
	-U -	file name of user stylesheet for EPUB layout
	-X	disable document styles for EPUB layout

	-o -	output file name (%d for page number)
	-F -	output format (default inferred from output file name)
			png, pnm, pgm, ppm, pam, tga, pbm, pkm,
			pdf, svg, cbz
	-O -	comma separated list of options for output format

	pages	comma separated list of page ranges (N=last page)

Common raster format output options:
	rotate=N: rotate rendered pages N degrees counterclockwise
	resolution=N: set both X and Y resolution in pixels per inch
	x-resolution=N: X resolution of rendered pages in pixels per inch
	y-resolution=N: Y resolution of rendered pages in pixels per inch
	width=N: render pages to fit N pixels wide (ignore resolution
	    option)
	height=N: render pages to fit N pixels tall (ignore resolution
	    option)
	colorspace=(gray|rgb|cmyk): render using specified colorspace
	alpha: render pages with alpha channel and transparent background

Structured text output options:
	preserve-ligatures: do not expand all ligatures into constituent
	    characters
	preserve-whitespace: do not convert all whitespace characters
	    into spaces
```

```
PDF output options:
	decompress: decompress all streams (except compress-fonts/images)
	compress: compress all streams
	compress-fonts: compress embedded fonts
	compress-images: compress images
	ascii: ASCII hex encode binary streams
	pretty: pretty-print objects with indentation
	linearize: optimize for web browsers
	sanitize: clean up graphics commands in content streams
	garbage: garbage collect unused objects
	or garbage=compact: ... and compact cross reference table
	or garbage=deduplicate: ... and remove duplicate objects

SVG output options:
	text=text: Emit text as <text> elements (inaccurate fonts).
	text=path: Emit text as <path> elements (accurate fonts).
	no-reuse-images: Do not reuse images using <symbol> definitions.
```

35.4 Create

The `create` tool allows PDFs to generated from simple text files full of pdf operator streams plus formatted comments.

```
$ mutool create
usage: mutool create [-o output.pdf] [-O options] page.txt [page2.txt
    ...]
	-o -	name of PDF file to create
	-O -	comma separated list of output options
	page.txt content stream with annotations for creating resources

Content stream special commands:
	%%MediaBox LLX LLY URX URY
	%%Rotate Angle
	%%Font Name Filename (or base 14 font name)
	%%Image Name Filename

PDF output options:
	decompress: decompress all streams (except compress-fonts/images)
	compress: compress all streams
	compress-fonts: compress embedded fonts
	compress-images: compress images
	ascii: ASCII hex encode binary streams
	pretty: pretty-print objects with indentation
	linearize: optimize for web browsers
	sanitize: clean up graphics commands in content streams
	garbage: garbage collect unused objects
	or garbage=compact: ... and compact cross reference table
```

```
       or garbage=deduplicate: ... and remove duplicate objects
```

35.5 Draw

The `draw` utility is the most commonly used tool, capable of converting/rendering documents to a range of bitmap and vector formats. It performs a similar task to the `convert` utility, using a different set of internal mechanisms. Which is better for any given task is often a matter of taste.

```
$ mutool draw
mudraw version 1.11
Usage: mudraw [options] file [pages]
       -p -   password

       -o -   output file name (%d for page number)
       -F -   output format (default inferred from output file name)
              raster: png, tga, pnm, pam, pbm, pkm, pwg, pcl, ps
              vector: svg, pdf, trace
              text: txt, html, stext

       -s -   show extra information:
              m - show memory use
              t - show timings
              f - show page features
              5 - show md5 checksum of rendered image

       -R -   rotate clockwise (default: 0 degrees)
       -r -   resolution in dpi (default: 72)
       -w -   width (in pixels) (maximum width if -r is specified)
       -h -   height (in pixels) (maximum height if -r is specified)
       -f -   fit width and/or height exactly; ignore original aspect
              ratio
       -B -   maximum band_height (pgm, ppm, pam, png output only)

       -W -   page width for EPUB layout
       -H -   page height for EPUB layout
       -S -   font size for EPUB layout
       -U -   file name of user stylesheet for EPUB layout
       -X     disable document styles for EPUB layout

       -c -   colorspace (mono, gray, grayalpha, rgb, rgba, cmyk,
              cmykalpha)
       -G -   apply gamma correction
       -I     invert colors

       -A -   number of bits of antialiasing (0 to 8)
       -A -/- number of bits of antialiasing (0 to 8) (graphics, text)
       -l -   minimum stroked line width (in pixels)
```

```
        -D      disable use of display list
        -i      ignore errors
        -L      low memory mode (avoid caching, clear objects after each
                page)
        -P      parallel interpretation/rendering
        -N      disable ICC workflow ("N"o color management)
        -O      Control spot rendering
                  0 = No spot rendering
                  1 = Overprint simulation
                  2 = Full spot rendering

        -y l    List the layer configs to stderr
        -y -    Select layer config (by number)
        -y -{,-}* Select layer config (by number), and toggle the
                  listed entries

        pages   comma separated list of page numbers and ranges
```

35.6 Extract

The `extract` utility extracts objects from PDF files. This can be used to extract resources such as fonts, ICC profiles, images etc.

```
$ mutool extract
usage: mutool extract [options] file.pdf [object numbers]
        -p      password
        -r      convert images to rgb
```

Run without any object numbers listed, it will extract all the resources it can find within the file to your local filesystem. It is best to run this in a clean directory!

Alternatively specific resources can be extracted by specifying the PDF object numbers. These can be obtained by using the `info` utility.

35.7 Info

The `info` utility displays various different sets of information on the contents of a PDF file.

```
$ mutool info
usage: mutool info [options] file.pdf [pages]
        -p -    password for decryption
        -F      list fonts
        -I      list images
        -M      list dimensions
        -P      list patterns
```

```
       -S     list shadings
       -X     list form and postscript xobjects
       pages  comma separated list of page numbers and ranges
```

35.8 Merge

The `merge` utility allows a new output PDF to be created by combining pages (or ranges of pages) from a set of input pdfs.

```
$ mutool merge
usage: mutool merge [-o output.pdf] [-O options] input.pdf [pages]
    [input2.pdf] [pages2] ...
       -o -      name of PDF file to create
       -O -      comma separated list of output options
       input.pdf name of input file from which to copy pages
       pages     comma separated list of page numbers and ranges

PDF output options:
       decompress: decompress all streams (except compress-fonts/images)
       compress: compress all streams
       compress-fonts: compress embedded fonts
       compress-images: compress images
       ascii: ASCII hex encode binary streams
       pretty: pretty-print objects with indentation
       linearize: optimize for web browsers
       sanitize: clean up graphics commands in content streams
       garbage: garbage collect unused objects
       or garbage=compact: ... and compact cross reference table
       or garbage=deduplicate: ... and remove duplicate objects
```

35.9 Pages

The `pages` utility displays information on the different pages (such as MediaBoxes, CropBoxes etc) of a PDF file.

```
$ mutool pages
usage: mutool pages [options] file.pdf [pages]
       -p -   password for decryption
       pages  comma separated list of page numbers and ranges
```

35.10 Portfolio

The `portfolio` utility is used to create or manipulate PDF portfolios.

```
$ mutool portfolio
```

```
usage: mutool portfolio [options] portfolio.pdf [actions]

Options are:
       -p -   password
       -o -   output (defaults to input file)
       -O -   PDF output options (see mutool create)

Actions are:
       t      display a table listing the contents of the portfolio
       x N <file>
              extract Nth entry to <file>
       a <file> <name>
              add contents of <file> as an entry named <name>

For safety, only use ASCII characters in entry names for now.
```

35.11 Poster

The `poster` utility is used to subdivide pages within a PDF so that they can be printed to a small format printer and then pasted together to form a poster.

```
$ mutool poster
usage: mutool poster [options] input.pdf [output.pdf]
       -p -   password
       -x     x decimation factor
       -y     y decimation factor
```

35.12 Run

Unlike the other utilities, invoking `run` with no arguments will not get you a list of arguments, but will instead start an interpreter waiting for your input. This interpreter expects to be fed Javascript commands to be executed using MuJS.

The PDF bindings present within the Javascript environment allow many powerful operations to be scripted on PDF and other files.

35.13 Show

The `show` command will display various sections of the PDF file in printable format. This will primarily be of use to people working with the internals of PDF files, but the `outline` option provides a way to get the outlines (a.k.a. bookmarks) from a PDF file at the command line.

```
$ mutool show
```

```
usage: mutool show [options] file.pdf [grep] [xref] [trailer] [pagetree]
    [outline] [object numbers]
       -p -   password
       -o -   output file
       -b     print streams as binary data
       -e     print encoded streams (don't decode)
```

Chapter 36

MuOfficeLib

The MuPDF API is designed to be a set of interlocking pieces that can be assembled together in many different ways to offer a powerful range of functionality. The cost of this versatility is that a certain amount of assembly is required.

For those people who would prefer a more encapsulated solution, we have a helper library, MuOfficeLib, that handles much of the

Chapter 37

Transitions

For a basic description of how to query a document for the specified presentation details, see section 8.9 Presentations. Given these details, the caller is still responsible for displaying (animating) the transition.

To help with this task, MuPDF provides a helper function that generate the required frame of the transition into the supplied target pixmap from supplied rendered pixmaps of the start/end of the transition:

```
/*
    fz_generate_transition: Generate a frame of a transition.

    tpix: Target pixmap
    opix: Old pixmap
    npix: New pixmap
    time: Position within the transition (0 to 256)
    trans: Transition details

    Returns 1 if successfully generated a frame.
*/
int fz_generate_transition(fz_context *ctx, fz_pixmap *tpix, fz_pixmap
    *opix, fz_pixmap *npix, int time, fz_transition *trans);
```

This helper function is largely independent of the rest of MuPDF, and will be dropped from the library at link time if it is not used.

It is still the callers responsibility to drive this, to display the updated pixmaps, and to generate the time field as appropriate from the actual real clock values.

Chapter 38

MuThreads

In order to implement tools such as MuTool, and helpers such as mu-office-lib, we needed a simple cross platform threading library. MuThreads is that library.

It is a very simple interface layer that gives a consistent implementation of Mutexes, Semaphores and Threads that builds both on pthreads and on Windows threads. More platforms may be added in the future.

The interface is described in a single header, '`mupdf/helpers/mu-threads.h`', and should be self-evident.

Part V

Platform specifics and Language Bindings

Chapter 39

Platform specifics

39.1 Overview

MuPDF is designed to compile and run on almost any platform that supports a standard C runtime library. In general, we try to restrict our use of C features later than C89, and to rely only on POSIX standard APIs.

Our primary development platforms are Linux (various versions, 32 and 64bit using both Clang and GCC compilers) and Windows (32 and 64bit, using MSVC 2005).

The choice of MSVC 2005 is deliberate; executables produced by this version run on every Windows version from XP (Service Pack 3) up to the latest (Windows 10 Creators Update, at the time of writing). More importantly, VS2005 solutions/projects can be imported into later versions, whereas projects from later versions cannot be used in earlier ones.

MSVC 2005 is not truly standards compliant, but it's close enough for our purposes. Neither is Windows POSIX compliant, but in the cases where a POSIX API is required and not available, we provide windows specific code, often by implementing an equivalent API using Windows platform calls.

We do not currently actively use MacOS for developing MuPDF, though we have in the past. MuPDF compiles and runs fine on MacOS, and we intend to keep it so. There is no MacOS specific code in the MuPDF codebase; the MuPDF viewer for MacOS is just the standard Linux X11 one. Similarly GSView 6.0 for Mac is a QT based application that uses the same code for Linux and Mac.

MuPDF has language bindings for Android, iOS, Java and JavaScript.

39.2 C library

On all platforms, MuPDF builds a C library that offers the standard C level API as defined within this book. By default we build a static library. Dynamic libraries can be built (and indeed are used on platforms such as Android), but we do not guarantee ABI compatibility between releases, so we do not provide this on all platforms by default.

The standard Unix Makefile requires GNU Make, and detects the presence of the third party libraries within the `thirdparty` directory. If present, these are used in preference to any system libraries.

Our libraries frequently contain bug fixes to the standard ones, and we attempt to pass such fixes upstream to the main package maintainers. Given that all our regression and quality control testing happens with the versions that we supply, we prefer people to use our versions, and will frequently ask for bug reports to be reproduced using such a build before investing too much time in trying to solve them.

39.3 Android viewer

39.4 Desktop Java

39.4.1 Language bindings

39.4.2 Viewer

39.5 GSView

39.6 Javascript

39.7 Linux/Windows viewer

39.7.1 Non-GL version

39.7.2 GLFW version

Appendix A

How to contribute to MuPDF

CLA. Style. Philosophy.