libdicom

Markus D. Herrmann

Mar 28, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

C library and executable tools for reading and writing DICOM data sets.

1.1 Supported parts of the standard

- Part 5 - Data Structures and Encoding
- Part 6 - Data Dictionary
- Part 10 - Media Storage and File Format for Media Interchange

Note that the library does not read the Pixel Data element at once, but instead provides an interface to randomly access individual frame items of Pixel Data elements. However, the library does not concern itself with decoding the values of frame items.

1.2 Design goals

The library aims to:

- Provide a stable application binary interface (ABI)
- Be highly portable and run on Linux, Unix, macOS, and Windows operating systems with different architectures
- Be dead simple and free of surprises
- Have no external build or runtime dependencies
- Be easily callable from other languages via a C foreign function interface (FFI)
- Be fast to compile and produce small binaries
- Be easty to compile to WebAssembly using Emscripten

CHAPTER

TWO

INSTALLATION

2.1 Building

The library and executables can be built using Meson:

```
meson setup --buildtype release builddir
meson compile -C builddir
meson install -C builddir
```

2.2 Build dependencies

To install build dependencies:

On Debian-based Linux distributions:

```
sudo apt install build-essential git meson pkg-config
```

On macOS:

```
brew install git meson pkg-config
```

On Windows:

```
# Pin to Meson 0.60.1 for https://github.com/mesonbuild/meson/issues/10022
pip install meson==0.60.1
```

2.3 Build configuration

Build for development and debugging:

```
meson setup builddir --buildtype debug
meson compile -C builddir
```

2.4 Optional dependencies

This package uses check for unit testing and uthash for data structures. It will automatically download and build both libraries, or can use system copies.

To install system copies:

On Debian-based Linux distributions:

```
sudo apt install check uthash-dev
```

On macOS:

```
brew install check uthash
```

CHAPTER

THREE

USAGE

3.1 API overview

A Filehandle (*DcmFilehandle*) enables access of a DICOM file, which contains an encoded Data Set representing a SOP Instance. A Filehandle can be created via *dcm_filehandle_create_from_file()* or *dcm_filehandle_create_from_memory()* , and destroyed via *dcm_filehandle_destroy()*. You can make your own load functions to load from other IO sources, see *dcm_filehandle_create()*.

The content of a Part10 file can be read using various functions.

The File Meta Information can be accessed via *dcm_filehandle_get_file_meta()*.

The principal metadata of the Data Set can be accessed via *dcm_filehandle_get_metadata_subset()*. This function will stop read on tags which are likely to take a long time to process.

You can read all metadata and control read stop using a sequence of calls to *dcm_filehandle_read_metadata()*.

In case the Data Set contained in a Part10 file represents an Image instance, individual frames may be read out with *dcm_filehandle_read_frame()*.

Use *dcm_filehandle_read_frame_position()* to read the frame at a certain (column, row) position. This will return NULL and set the error code *DCM_ERROR_CODE_MISSING_FRAME* if there is no frame at that position.

A Data Element (*DcmElement*) is an immutable data container for storing values.

Every data element has a tag indicating its purpose. Tags are 32-bit unsigned ints with the top 16 bits indicating the group and the bottom 16 the element. They are usually written in hexadecimal, perhaps `0x00400554`, meaning element `0x554` of group `0x40`, or as keywords, in this case `"SpecimenUID"`. You can get the tag from its corresponding keyword with *dcm_dict_tag_from_keyword()*, or find the keyword from a tag with *dcm_dict_keyword_from_tag()*.

Every Data Element has a Value Representation (VR), which specifies the data type and format of the contained value. VRs can be conceptually grouped into numbers (integers or floating-point values), numeric strings (strings of characters encoding numbers using the decimal or scientific notation), character strings (text of restriction length and character repertoire), or byte strings (unicode). Each VR is represented using a standard C type (e.g,. VR `"US"` has type `uint16_t` and VR `"UI"` has type `char *`) and additional value constraints may be checked at runtime (e.g., the maximal capacity of a character string).

The VR must be appropriate for the tag. Use *dcm_vr_from_tag()* to find the set of allowed VRs for a tag. Use *dcm_is_valid_vr_for_tag()* to check if a VR is allowed for a tag.

Depending on the VR, an individual Data Element may have a Value Multiplicity (VM) greater than one, i.e., contain more than one value. Under the hood, a Data Element may thus contain an array of values.

A Data Element can be created with *dcm_element_create()*, it can have a value assigned to it with eg. *dcm_element_set_value_integer()*, and it can be destroyed with *dcm_element_destroy()*. See *Memory management* below for details on pointer ownership.

An individual value can be retrieved via the getter functions like (e.g., *dcm_element_get_value_integer()*). Note that in case of character string or binary values, the getter function returns the pointer to the stored character array (`const char *`) and that pointer is only valid for the lifetime of the Data Element. When a Data Element is destroyed, the memory allocated for contained values is freed and any pointers to the freed memory area become dangling pointers.

A Data Set (*DcmDataSet*) is an ordered collection of Data Elements (*DcmElement*). A Data Set can be created via *dcm_dataset_create()* and destroyed via *dcm_dataset_destroy()*. Data Elements can be added to a Data Set via *dcm_dataset_insert()*, removed from a Data Set via *dcm_dataset_remove()*, and retrieved from a Data Set via *dcm_dataset_get()* or *dcm_dataset_get_clone()*.

When a Data Element is added to a Data Set, the Data Set takes over ownership of the memory allocated for contained Data Elements. When a Data Element is retrieved from a Data Set, it may either be borrowed with ownership of the memory allocated for the Data Element remaining with the Data Set in case of *dcm_dataset_get()* or copied with the caller taking on ownership of the memory newly allocated for the Data Element in case of *dcm_dataset_get_clone()*.

An individual Data Element can only be part of only one Data Set. When a Data Element is removed from a Data Set, the memory allocated for the Data Element is freed. When a Data Set is destroyed, all contained Data Elements are also automatically destroyed.

A Sequence (`DcmSequence`) is an ordered collection of Items, each containing one Data Set. A Sequence can be created via *dcm_sequence_create()* and destroyed via *dcm_sequence_destroy()*. Data Sets can be added to a Sequence via *dcm_sequence_append()*, removed from a Sequence via *dcm_sequence_remove()*, and retrieved from a Sequence via *dcm_sequence_get()*.

When a Data Set is added to a sequence, the sequence takes over ownership of the memory allocated for the Data Set (and consequently of each contained Data Element). When a Data Set is retrieved from a sequence, it is only borrowed and ownership of the memory allocated for the Data Set remains with the sequence. Retrieved Data Sets are immutable (locked). When a Data Set is removed from a sequence, the Data Set is destroyed (i.e., the allocated memory is freed). When a Sequence is destroyed, all contained Data Sets are also automatically destroyed.

3.2 Thread safety

By design, libdicom has no dependencies, not even on a threading library. This means it can't be threadsafe, since it can't lock any internal datastructures. However, there are no global structures, so as long as you don't share a *DcmFilehandle* between threads, you're fine.

You can share *DcmFilehandle* between threads if you lock around calls into libdicom. The lock only needs to be per-*DcmFilehandle*, you don't need a global lock.

3.3 Error handling

Library functions which can return an error take a double pointer to a *DcmError* struct as a first argument. If an error is detected, this pointer will be updated to refer to an error object. You can extract a `DcmErrorCode` with *dcm_error_get_code()*, an error summary with *dcm_error_get_summary()*, and a detailed error message with *dcm_error_get_message()*. After presenting the error to the user, call *dcm_error_clear()* to clear the error pointer and free any allocated memory.

You can pass *NULL* instead of an error pointer if you are not interested in error messages. In this case, any errors will be logged to debug instead, see *dcm_log_debug()*.

For example:

```c
#include <stdlib.h>
#include <dicom/dicom.h>

int main() {
    const char *file_path = "bad-file";
    DcmError *error = NULL;

    DcmFilehandle *filehandle = dcm_filehandle_create_from_file(&error, file_path);
    if (filehandle == NULL) {
        printf("error detected: %s\n", dcm_error_code_str(dcm_error_get_code(error)));
        printf("summary: %s\n", dcm_error_get_summary(error));
        printf("message: %s\n", dcm_error_get_message(error));
        dcm_error_clear(&error);
        return 1;
    }

    dcm_filehandle_destroy(filehandle);

    return 0;
}
```

3.4 Memory management

libdicom objects (Data Element, Data Set, Sequence, Frame Item, etc.) can contain references to other libdicom objects. For example, you can set a sequence as the value of an element like this:

```c
if (!dcm_element_set_value_sequence(error, element, sequence)) {
    handle error;
}
```

If this function succeeds, ownership of the sequence object passes to the element, i.e., when the element is destroyed, the sequence will also be destroyed.

If this function fails, ownership does not transfer.

libdicom objects can also contain references to data structures allocated by other programs, for example, arrays of numeric values.

```c
int *values = pointer to array of integers;
uint32_t vm = number of ints in array;
if( !dcm_element_set_value_numeric_multi(error, element, values, vm, true)) {
  handle error;
}
```

The final parameter, *steal*, sets whether ownership of the pointer to the array should be "stolen" by libdicom. If it is true, then libdicom will use `free()` to free the array when the element is freed. If it is false, libdicom will make a copy of the array.

3.5 Getting started

Below is an example for reading metadata from a DICOM Part10 file and printing an element to standard output:

```c
#include <stdlib.h>
#include <dicom/dicom.h>

int main() {
    const char *file_path = "/path/to/file.dcm";
    DcmError *error = NULL;

    DcmFilehandle *filehandle = dcm_filehandle_create_from_file(&error, file_path);
    if (filehandle == NULL) {
        dcm_error_log(error);
        dcm_error_clear(&error);
        return 1;
    }

    const DcmDataSet *metadata =
        dcm_filehandle_get_metadata_subset(&error, filehandle);
    if (metadata == NULL) {
        dcm_error_log(error);
        dcm_error_clear(&error);
        dcm_filehandle_destroy(filehandle);
        return 1;
    }

    const char *num_frames;
    uint32_t tag = dcm_dict_tag_from_keyword("NumberOfFrames");
    DcmElement *element = dcm_dataset_get(&error, metadata, tag);
    if (element == NULL ||
        !dcm_element_get_value_string(&error, element, 0, &num_frames)) {
        dcm_error_log(error);
        dcm_error_clear(&error);
        dcm_filehandle_destroy(filehandle);
        return 1;
    }

    printf("NumerOfFrames == %s\n", num_frames);

    dcm_filehandle_destroy(filehandle);

    return 0;
}
```

CHAPTER

FOUR

API DOCUMENTATION

DCM_CAPACITY_AE
: Maximum number of characters in values with Value Representation AE.

DCM_CAPACITY_AS
: Maximum number of characters in values with Value Representation AS.

DCM_CAPACITY_AT
: Maximum number of characters in values with Value Representation AT.

DCM_CAPACITY_CS
: Maximum number of characters in values with Value Representation CS.

DCM_CAPACITY_DA
: Maximum number of characters in values with Value Representation DA.

DCM_CAPACITY_DS
: Maximum number of characters in values with Value Representation DS.

DCM_CAPACITY_DT
: Maximum number of characters in values with Value Representation DT.

DCM_CAPACITY_IS
: Maximum number of characters in values with Value Representation IS.

DCM_CAPACITY_LO
: Maximum number of characters in values with Value Representation LO.

DCM_CAPACITY_LT
: Maximum number of characters in values with Value Representation LT.

DCM_CAPACITY_PN
: Maximum number of characters in values with Value Representation PN.

DCM_CAPACITY_SH
: Maximum number of characters in values with Value Representation SH.

DCM_CAPACITY_ST
: Maximum number of characters in values with Value Representation ST.

DCM_CAPACITY_TM
: Maximum number of characters in values with Value Representation TM.

DCM_CAPACITY_UC
: Maximum number of characters in values with Value Representation UC.

DCM_CAPACITY_UI

Maximum number of characters in values with Value Representation UI.

DCM_CAPACITY_UR

Maximum number of characters in values with Value Representation UR.

DCM_CAPACITY_UT

Maximum number of characters in values with Value Representation UT.

void **dcm_init**(void)

Start up libdicom.

Call this from the main thread during program startup.

This function can be called many times.

Deprecated since version 1.1.0: Calling this function is no longer necessary.

type **DcmError**

Error return object.

enum **_DcmErrorCode**

Enumeration of error codes.

enumerator **DCM_ERROR_CODE_NOMEM** = 1

Out of memory

enumerator **DCM_ERROR_CODE_INVALID** = 2

Invalid parameter

enumerator **DCM_ERROR_CODE_PARSE** = 3

Parse error

enumerator **DCM_ERROR_CODE_IO** = 4

IO error

enumerator **DCM_ERROR_CODE_MISSING_FRAME** = 5

Missing frame

const char ***dcm_error_code_str**(DcmErrorCode code)

Convert an error code to a human-readable string.

Parameters

- **code** – The error code

Returns

A string that can be displayed to users

const char ***dcm_error_code_name**(DcmErrorCode code)

Get a symbolic name for a DcmErrorCode.

Parameters

- **code** – The error code

Returns

A symbolic name for the code.

void **dcm_error_set**(*DcmError* **error, DcmErrorCode code, const char *summary, const char *format, ...)

Set an error.

Create a new DcmError object and store the pointer in error.

You can't set error twice – always check the error state and return immediately if set.

Parameters

- **error** – Pointer to store the new error object in
- **code** – Numeric error code
- **summary** – Summary of error
- **format** – printf-style format string
- **...** – Format string arguments

void **dcm_error_clear**(*DcmError* **error)

Clear an error, if set.

Parameters

- **error** – Pointer holding the error object

const char ***dcm_error_get_summary**(*DcmError* *error)

Get a summary of the error.

Do not free this result. The pointer will be valid as long as error is valid.

Parameters

- **error** – DcmError to read the error from

Returns

Short description of the error

const char ***dcm_error_get_message**(*DcmError* *error)

Get the error message.

Do not free this result. The pointer will be valid as long as error is valid.

Parameters

- **error** – Error object

Returns

Message stored in a error object

DcmErrorCode **dcm_error_get_code**(*DcmError* *error)

Get the error code.

Parameters

- **error** – Error object

Returns

Error code

void **dcm_error_log**(*DcmError* *error)

Log an error message using information stored on the error object.

Parameters

- **error** – Error object

void **dcm_error_print**(*DcmError* *error)

Print an error message to stderr.

Parameters

- **error** – Error object

void **dcm_free**(void *pointer)

Free an allocated memory area.

Any memory allocated by libdicom and returned to the calling program should be freed with this.

Parameters

- **pointer** – Memory area to free

void ***dcm_calloc**(*DcmError* **error, uint64_t n, uint64_t size)

Allocate and zero an area of memory.

Any memory which you pass to libdicom and which you ask libdicom to manage with a "steal" flag should be allcoatyed with one of the libdicom memory allocators.

Parameters

- **error** – Pointer to error object
- **n** – Number of items to allocate
- **size** – Size of each item in bytes

Returns

Pointer to memory area

enum **_DcmLogLevel**

Enumeration of log levels

enumerator **DCM_LOG_CRITICAL** = 50

Critical

enumerator **DCM_LOG_ERROR** = 40

Error

enumerator **DCM_LOG_WARNING** = 30

Warning

enumerator **DCM_LOG_INFO** = 20

Info

enumerator **DCM_LOG_DEBUG** = 10

Debug

enumerator **DCM_LOG_NOTSET** = 0

Not set (no logging)

DcmLogLevel **dcm_log_set_level**(DcmLogLevel log_level)

Set the log level.

Parameters

- **log_level** – New log level.

Returns

previous log level

type **DcmLogf**

Log function. See dcm_log_set_logf().

DcmLogf **dcm_log_set_logf**(*DcmLogf* logf)

Set the log function.

This function will be used to log any error or warning messages from the library. The default DcmLogf function prints messages to stderr. Set to NULL to disable all logging.

Parameters

- **logf** – New log function.

Returns

previous log function

void **dcm_log_critical**(const char *format, ...)

Write critical log message to stderr stream.

Parameters

- **format** – printf-style format string
- **...** – Format string arguments

void **dcm_log_error**(const char *format, ...)

Write error log message to stderr stream.

Parameters

- **format** – printf-style format string
- **...** – Format string arguments

void **dcm_log_warning**(const char *format, ...)

Write warning log message to stderr stream.

Parameters

- **format** – printf-style format string
- **...** – Format string arguments

void **dcm_log_info**(const char *format, ...)

Write info log message to stderr stream.

Parameters

- **format** – printf-style format string
- **...** – Format string arguments

void **dcm_log_debug**(const char *format, ...)

Write debug log message to stderr stream.

Parameters

- **format** – printf-style format string
- **...** – Format string arguments

const char ***dcm_get_version**(void)

Get the version of the library.

Returns

semantic version string

enum **_DcmVR**

An enum of Value Representations.

Value Representations which are not known to libdicom will be coded as DCM_VR_ERROR (unknown Value Representation).

Note to maintainers: this enum must match the table in dicom-dict.c, and the DcmVRTag enum. As the DICOM standard evolves, numbering must be maintained for ABI compatibility.

enum **_DcmVRClass**

The general class of the value associated with a Value Representation.

DCM_VR_CLASS_STRING_MULTI – one or more null-terminated strings, cannot contain backslash

DCM_VR_CLASS_STRING_SINGLE – a single null-terminated string, backslash allowed

DCM_VR_CLASS_NUMERIC_DECIMAL – one or more binary floating point numeric values, other fields give sizeof(type)

DCM_VR_CLASS_NUMERIC_INTEGER – one or more binary integer numeric values, other fields give sizeof(type)

DCM_VR_CLASS_BINARY – an uninterpreted array of bytes, length in the element header

DCM_VR_CLASS_SEQUENCE – Value Representation is a sequence

DcmVRClass **dcm_dict_vr_class**(DcmVR vr)

Find the general class for a particular Value Representation.

Parameters

- **vr** – The Value Representation

Returns

The general class of that Value Representation

DcmVR **dcm_dict_vr_from_str**(const char *vr)

Turn a string Value Representation into an enum value.

Parameters

- **vr** – The Value Representation as a two character string.

Returns

the enum for that Value Representation

const char ***dcm_dict_str_from_vr**(DcmVR vr)

Turn an enum Value Representation into a character string.

Parameters

- **vr** – The Value Representation as an enum value.

Returns

the string representation of that Value Representation, or NULL

const char ***dcm_dict_keyword_from_tag**(uint32_t tag)

Look up the Keyword of an Attribute in the Dictionary.

Returns NULL if the tag is not recognised.

Parameters

- **tag** – Attribute Tag

Returns
attribute Keyword

uint32_t **dcm_dict_tag_from_keyword**(const char *keyword)

Look up the tag of an Attribute in the Dictionary.

Returns `0xffffffff` if the keyword is not recognised.

Parameters

- **tag** – Attribute keyword

Returns
attribute tag

DcmVR **dcm_vr_from_tag**(uint32_t tag)

Find the Value Representation for a tag.

This will return DCM_VR_ERROR if the tag is unknown, or does not have a unique Value Representation.

Parameters

- **tag** – Attribute Tag

Returns
the unique Value Representation for this tag, or DCM_VR_ERROR

bool **dcm_is_public_tag**(uint32_t tag)

Determine whether a Tag is public.

A Tag is public if it is defined in the Dictionary.

Parameters

- **tag** – Attribute Tag

Returns
Yes/no answer

bool **dcm_is_private_tag**(uint32_t tag)

Determine whether a Tag is private.

Parameters

- **tag** – Attribute Tag

Returns
Yes/no answer

bool **dcm_is_valid_tag**(uint32_t tag)

Determine whether a Tag is valid.

Parameters

- **tag** – Attribute Tag

Returns
Yes/no answer

bool **dcm_is_valid_vr**(const char *vr)

Determine whether a Value Representation is valid.

Parameters

- **vr** – Attribute Value Representation

Returns
Yes/no answer

bool **dcm_is_valid_vr_for_tag**(DcmVR vr, uint32_t tag)

Determine whether a Value Representation is valid.

Parameters

- **vr** – Attribute Value Representation

Returns
Yes/no answer

bool **dcm_is_encapsulated_transfer_syntax**(const char *transfer_syntax_uid)

Determine whether a Transfer Syntax is encapsulated.

Parameters

- **transfer_syntax_uid** – Transfer Syntax UID

Returns
Yes/no answer

type **DcmElement**

Data Element.

DcmElement ***dcm_element_create**(*DcmError* **error, uint32_t tag, DcmVR vr)

Create a Data Element for a tag.

After creating a Data Element, you must attach an appropriate value using one of the setting functions. See for example *dcm_element_set_value_string()*.

Parameters

- **error** – Pointer to error object
- **tag** – Tag
- **vr** – The Value Representation for this Data Element

Returns
Pointer to Data Element

uint16_t **dcm_element_get_group_number**(const *DcmElement* *element)

Get group number (first part of Tag) of a Data Element.

Parameters

- **element** – Pointer to Data Element

Returns
Tag group number

uint16_t **dcm_element_get_element_number**(const *DcmElement* *element)

Get Element Number (second part of Tag) of a Data Element.

Parameters

- **element** – Pointer to Data Element

Returns
Tag Element Number

uint32_t **dcm_element_get_tag**(const *DcmElement* *element)

Get Tag of a Data Element.

Parameters

- **element** – Pointer to Data Element

Returns

Tag

DcmVR **dcm_element_get_vr**(const *DcmElement* *element)

Get the Value Representation of a Data Element.

Parameters

- **element** – Pointer to Data Element

Returns

Value Representation

uint32_t **dcm_element_get_length**(const *DcmElement* *element)

Get length of the entire value of a Data Element.

Parameters

- **element** – Pointer to Data Element

Returns

Length of value of Data Element

uint32_t **dcm_element_get_vm**(const *DcmElement* *element)

Get Value Multiplicity of a Data Element.

Parameters

- **element** – Pointer to Data Element

Returns

Value Multiplicity

bool **dcm_element_is_multivalued**(const *DcmElement* *element)

Determine whether a Data Element has a Value Multiplicity greater than one.

Parameters

- **element** – Pointer to Data Element

Returns

Yes/no answer

DcmElement ***dcm_element_clone**(*DcmError* **error, const *DcmElement* *element)

Clone (i.e., create a deep copy of) a Data Element.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element

Returns

Pointer to clone of Data Element

bool **dcm_element_get_value_string**(*DcmError* **error, const *DcmElement* *element, uint32_t index, const char **value)

Get a string from a string-valued Data Element.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **index** – Zero-based index of value within the Data Element
- **value** – Pointer to return location for value

Returns
true on success

bool **dcm_element_set_value_string**(*DcmError* **error, *DcmElement* *element, char *value, bool steal)

Set the value of a Data Element to a character string.

The Data Element must have a Tag that allows for a character string Value Representation. If that is not the case, the function will fail.

On success, if *steal* is true, ownership of *value* passes to *element*, i.e. it will be freed when *element* is destroyed. If *steal* is false, then a copy is made of *value* and ownership is not transferred.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **value** – String value
- **steal** – if true, ownership of value passes to element

Returns
true on success

bool **dcm_element_set_value_string_multi**(*DcmError* **error, *DcmElement* *element, char **values, uint32_t vm, bool steal)

Set the value of a Data Element to an array of character strings.

The Data Element must have a Tag that allows for a character string Value Representation and for a Value Multiplicity greater than one. If that is not the case, the function will fail.

On success, if *steal* is true, ownership of *value* passes to *element*, i.e. it will be freed when *element* is destroyed. If *steal* is false, then a copy is made of *value* and ownership is not transferred.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **values** – Pointer to memory location where values are written to
- **vm** – Number of values
- **steal** – if true, ownership of values passes to element

Returns
true on success

bool **dcm_element_get_value_integer**(*DcmError* **error, const *DcmElement* *element, uint32_t index, int64_t *value)

Get an integer from a 16, 32 or 64-bit integer-valued Data Element.

The integer held in the Element will be cast to int64_t for return.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **index** – Zero-based index of value within the Data Element
- **value** – Pointer to return location for value

Returns
true on success

bool **dcm_element_set_value_integer**(*DcmError* **error, *DcmElement* *element, int64_t value)

Set the value of a Data Element to an integer.

The Data Element must have a Tag that allows for a integer Value Representation. If that is not the case, the function will fail.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **value** – Integer value

Returns
true on success

bool **dcm_element_set_value_numeric_multi**(*DcmError* **error, *DcmElement* *element, void *values, uint32_t vm, bool steal)

Set the value of a Data Element to a number.

The Data Element must have a Tag that allows for a numeric Value Representation. If that is not the case, the function will fail.

Although the value passed is *void**, it should be a pointer to an array of 16- to 64-bit numeric values of the appropriate type for the Data Element Value Representation.

On success, if *steal* is true, ownership of *values* passes to *element*, i.e. it will be freed when *element* is destroyed. If *steal* is false, then a copy is made of *values* and ownership is not transferred.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **values** – Array of values
- **vm** – Number of values
- **steal** – if true, ownership of values passes to element

Returns
true on success

bool **dcm_element_get_value_decimal**(*DcmError* **error, const *DcmElement* *element, uint32_t index, double *value)

Get a floating-point value from a Data Element.

The Data Element Value Reepresentation may be either single- or double-precision floating point.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **index** – Zero-based index of value within the Data Element
- **value** – Pointer to return location for value

Returns
true on success

bool **dcm_element_set_value_decimal**(*DcmError* **error, *DcmElement* *element, double value)

Set the value of a Data Element to a floating-point.

The Data Element must have a Tag that allows for a floating-point Value Representation. If that is not the case, the function will fail.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **value** – Floating point value

Returns
true on success

bool **dcm_element_get_value_binary**(*DcmError* **error, const *DcmElement* *element, const void **value)

Get a binary value from a Data Element.

Use `dcm_element_length()` to get the length of the binary value.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **value** – Pointer to return location for value

Returns
true on success

bool **dcm_element_set_value_binary**(*DcmError* **error, *DcmElement* *element, void *value, uint32_t length, bool steal)

Set the value of a Data Element to binary data.

The Data Element must have a Tag that allows for a binary Value Representation. If that is not the case, the function will fail.

On success, if *steal* is true, ownership of *value* passes to *element*, i.e. it will be freed when *element* is destroyed. If *steal* is false, then a copy is made of *value* and ownership is not transferred.

Parameters

- **error** – Pointer to error object

- **element** – Pointer to Data Element
- **value** – Pointer to binary value
- **length** – Length in bytes of the binary value
- **steal** – if true, ownership of the value passes to element

Returns
true on success

bool **dcm_element_get_value_sequence**(*DcmError* **error, const *DcmElement* *element, DcmSequence **value)

Get a sequence value from a Data Element.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **value** – Pointer to return location for value

Returns
true on success

bool **dcm_element_set_value_sequence**(*DcmError* **error, *DcmElement* *element, DcmSequence *value)

Set the value of a Data Element to a Sequence.

The Data Element must have a Tag that allows for Value Representation "SQ". If that is not the case, the function will fail.

The Data Element takes ownership of the value pointer on success.

Parameters

- **error** – Pointer to error object
- **element** – Pointer to Data Element
- **value** – Pointer to Sequence

Returns
true on success

char ***dcm_element_value_to_string**(const *DcmElement* *element)

Make a string suitable for display to a user from the value of an element.

The return result must be freed with free(). The result may be NULL.

Returns
string to display

void **dcm_element_print**(const *DcmElement* *element, int indentation)

Print a Data Element.

Parameters

- **element** – Pointer to Data Element
- **indentation** – Number of white spaces before text

void **dcm_element_destroy**(*DcmElement* *element)

Destroy a Data Element.

Parameters

- **element** – Pointer to Data Element

type **DcmDataSet**

Data Set

DcmDataSet ***dcm_dataset_create**(*DcmError* **error)

Create an empty Data Set.

Parameters

- **error** – Pointer to error object

DcmDataSet ***dcm_dataset_clone**(*DcmError* **error, const *DcmDataSet* *dataset)

Clone (i.e., create a deep copy of) a Data Set.

Parameters

- **error** – Pointer to error object
- **dataset** – Pointer to Data Set

Returns

Pointer to clone of Data Set

bool **dcm_dataset_insert**(*DcmError* **error, *DcmDataSet* *dataset, *DcmElement* *element)

Insert a Data Element into a Data Set.

Parameters

- **error** – Pointer to error object
- **dataset** – Pointer to Data Set
- **element** – Pointer to Data Element

On success, the dataset takes over ownership of *element* and frees it when the dataset is destroyed.

If the insert operation fails, ownership does not pass and the caller is responsible for freeing *element*.

Returns

Whether insert operation was successful

bool **dcm_dataset_remove**(*DcmError* **error, *DcmDataSet* *dataset, uint32_t tag)

Remove a Data Element from a Data Set.

Parameters

- **error** – Pointer to error object
- **dataset** – Pointer to Data Set
- **tag** – Attribute Tag of a Data Element

Returns

Whether remove operation was successful

DcmElement ***dcm_dataset_get**(*DcmError* **error, const *DcmDataSet* *dataset, uint32_t tag)

Get a Data Element from a Data Set.

Parameters

- **error** – Pointer to error object
- **dataset** – Pointer to Data Set
- **tag** – Attribute Tag of a Data Element

Returns
Pointer to Data Element

DcmElement ***dcm_dataset_get_clone**(*DcmError* **error, const *DcmDataSet* *dataset, uint32_t tag)

Get a clone (deep copy) of a Data Element from a Data Set.

Parameters

- **error** – Pointer to error object
- **dataset** – Pointer to Data Set
- **tag** – Attribute Tag of a Data Element

Returns
Pointer to clone of Data Element

bool **dcm_dataset_foreach**(const *DcmDataSet* *dataset, bool (*fn)(const *DcmElement* *element, void *client), void *client)

Iterate over Data Elements in a Data Set.

The user function should return true to continue looping, or false to terminate the loop early.

The result is true if the whole Data Set returned true, or false if one call requested early termination.

The function must not modify the Data Set.

Parameters

- **seq** – Pointer to Data Set
- **fn** – Pointer to function that should be called for each Data Element
- **client** – Client data for function

Returns
true if all functions return true

DcmElement ***dcm_dataset_contains**(const *DcmDataSet* *dataset, uint32_t tag)

Fetch a Data Element from a Data Set, or NULL if not present.

Parameters

- **dataset** – Pointer to Data Set
- **tag** – Attribute Tag of a Data Element

Returns
Data Element, or NULL if not present

uint32_t **dcm_dataset_count**(const *DcmDataSet* *dataset)

Count the number of Data Elements in a Data Set.

Parameters

- **dataset** – Pointer to Data Set

Returns
Number of Data Elements

void **dcm_dataset_copy_tags**(const *DcmDataSet* *dataset, uint32_t *tags, uint32_t n)

Obtain a copy of the Tag of each Data Element in a Data Set.

The tags will be sorted in ascending order.

Parameters

- **dataset** – Pointer to Data Set
- **tags** – Pointer to memory location to of the array into which to copy tags. Number of items in the array must match the number of Data Elements in the Data Set as determined by *dcm_dataset_count()*.
- **n** – Number of items in the array.

Ownership of the memory allocated for *tags* remains with the caller. Specifically, the function does not free the memory allocated for *tags* if the copy operation fails.

void **dcm_dataset_lock**(*DcmDataSet* *dataset)

Lock a Data Set to prevent modification.

Parameters

- **dataset** – Pointer to Data Set

bool **dcm_dataset_is_locked**(const *DcmDataSet* *dataset)

Check whether a Data Set is locked.

Parameters

- **dataset** – Pointer to Data Set

Returns
Yes/no answer

void **dcm_dataset_print**(const *DcmDataSet* *dataset, int indentation)

Print a Data Set.

Parameters

- **error** – Pointer to error object
- **dataset** – Pointer to Data Set
- **indentation** – Number of white spaces before text

void **dcm_dataset_destroy**(*DcmDataSet* *dataset)

Destroy a Data Set.

Parameters

- **dataset** – Pointer to Data Set

Sequence

DcmSequence ***dcm_sequence_create**(*DcmError* **error)

Create a Sequence, i.e., an ordered list of Data Set items that represent the value of a Data Element with Value Representation SQ (Sequence).

Note that created object represents the value of a Data Element rather than a Data Element itself.

Parameters

- **error** – Pointer to error object

Returns
Pointer to Sequence

bool **dcm_sequence_append**(*DcmError* **error, DcmSequence *seq, *DcmDataSet* *item)

Append a Data Set item to a Sequence.

Parameters

- **error** – Pointer to error object
- **seq** – Pointer to Sequence
- **item** – Data Set item

On success, the sequence takes over ownership of *item* and frees it when the sequence is destroyed.

If the append fails, ownership does not pass and the caller is responsible for freeing *item*.

Returns
Whether append operation was successful

DcmDataSet ***dcm_sequence_get**(*DcmError* **error, const DcmSequence *seq, uint32_t index)

Get a Data Set item from a Sequence.

Parameters

- **error** – Pointer to error object
- **seq** – Pointer to Sequence
- **index** – Zero-based index of the Data Set item in the Sequence

Returns
Pointer to Data Set item

bool **dcm_sequence_foreach**(const DcmSequence *seq, bool (*fn)(const *DcmDataSet* *dataset, uint32_t index, void *client), void *client)

Iterate over Data Sets in a Sequence.

The user function should return true to continue looping, or false to terminate the loop early.

The result is true if the whole sequence returned true, or false if one call requested early termination.

The function must not modify the seqeucence.

Parameters

- **seq** – Pointer to Sequence
- **fn** – Pointer to function that should be called for each Data Set
- **client** – Client data for function

Returns
Pointer to Data Set item

bool **dcm_sequence_remove**(*DcmError* **error, DcmSequence *seq, uint32_t index)

Remove a Data Set item from a Sequence.

Parameters

- **error** – Pointer to error object
- **seq** – Pointer to Sequence
- **index** – Zero-based index of the Data Set item in the Sequence

Returns
Whether remove operation was successful

uint32_t **dcm_sequence_count**(const DcmSequence *seq)

Count the number of Data Set items in a Sequence.

Parameters

- **seq** – Pointer to Sequence

Returns
number of Data Set items

void **dcm_sequence_lock**(DcmSequence *seq)

Lock a Sequence to prevent modification.

Parameters

- **seq** – Pointer to Sequence

bool **dcm_sequence_is_locked**(const DcmSequence *seq)

Check whether a Sequence is locked.

Parameters

- **seq** – Pointer to Sequence

Returns
Yes/no answer

void **dcm_sequence_destroy**(DcmSequence *seq)

Destroy a Sequence.

Parameters

- **seq** – Pointer to Sequence

type **DcmFrame**

Frame Item of Pixel Data Element

Encoded pixels of an individual pixel matrix and associated descriptive metadata.

DcmFrame ***dcm_frame_create**(*DcmError* **error, uint32_t number, const char *data, uint32_t length, uint16_t rows, uint16_t columns, uint16_t samples_per_pixel, uint16_t bits_allocated, uint16_t bits_stored, uint16_t pixel_representation, uint16_t planar_configuration, const char *photometric_interpretation, const char *transfer_syntax_uid)

Create a Frame.

Parameters

- **error** – Pointer to error object
- **index** – Index of the Frame within the Pixel Data Element
- **data** – Pixel data of the Frame
- **length** – Size of the Frame (number of bytes)
- **rows** – Number of rows in pixel matrix
- **columns** – Number of columns in pixel matrix
- **samples_per_pixel** – Number of samples per pixel
- **bits_allocated** – Number of bits allocated per pixel
- **bits_stored** – Number of bits stored per pixel
- **pixel_representation** – Representation of pixels (unsigned integers or 2's complement)
- **planar_configuration** – Configuration of samples (color-by-plane or color-by-pixel)
- **photometric_interpretation** – Interpretation of pixels (monochrome, RGB, etc.)
- **transfer_syntax_uid** – UID of transfer syntax in which data is encoded

The object takes over ownership of the memory referenced by *data*, *photometric_interpretation*, and *transfer_syntax_uid* and frees it when the object is destroyed or if the creation fails.

Returns
Frame Item

uint32_t **dcm_frame_get_number**(const *DcmFrame* *frame)

Get number of a Frame Item within the Pixel Data Element.

Parameters

- **frame** – Frame

Returns
number (one-based index)

uint32_t **dcm_frame_get_length**(const *DcmFrame* *frame)

Get length of a Frame Item.

Parameters

- **frame** – Frame

Returns
number of bytes

uint16_t **dcm_frame_get_rows**(const *DcmFrame* *frame)

Get Rows of a Frame.

Parameters

- **frame** – Frame

Returns
number of rows in pixel matrix

uint16_t **dcm_frame_get_columns**(const *DcmFrame* *frame)

Get Columns of a Frame.

Parameters

- **frame** – Frame

Returns
number of columns in pixel matrix

uint16_t **dcm_frame_get_samples_per_pixel**(const *DcmFrame* *frame)

Get Samples per Pixel of a Frame.

Parameters

- **frame** – Frame

Returns
number of samples (color channels) per pixel

uint16_t **dcm_frame_get_bits_allocated**(const *DcmFrame* *frame)

Get Bits Allocated of a Frame.

Parameters

- **frame** – Frame

Returns
number of bits allocated per pixel

uint16_t **dcm_frame_get_bits_stored**(const *DcmFrame* *frame)

Get Bits Stored of a Frame.

Parameters

- **frame** – Frame

Returns

number of bits stored per pixel

uint16_t **dcm_frame_get_high_bit**(const *DcmFrame* *frame)

Get High Bit of a Frame.

Parameters

- **frame** – Frame

Returns

most significant bit of pixels

uint16_t **dcm_frame_get_pixel_representation**(const *DcmFrame* *frame)

Get Pixel Representation of a Frame.

Parameters

- **frame** – Frame

Returns

representation of pixels (unsigned integers or 2's complement)

uint16_t **dcm_frame_get_planar_configuration**(const *DcmFrame* *frame)

Get Planar Configuration of a Frame.

Parameters

- **frame** – Frame

Returns

configuration of samples (color-by-plane or color-by-pixel)

const char ***dcm_frame_get_photometric_interpretation**(const *DcmFrame* *frame)

Get Photometric Interpretation of a Frame.

Parameters

- **frame** – Frame

Returns

interpretation of pixels (monochrome, RGB, etc.)

const char ***dcm_frame_get_transfer_syntax_uid**(const *DcmFrame* *frame)

Get Transfer Syntax UID for a Frame.

Parameters

- **frame** – Frame

Returns

UID of the transfer syntax in which frame is encoded

const char ***dcm_frame_get_value**(const *DcmFrame* *frame)

Get pixel data of a Frame.

Parameters

- **frame** – Frame

Returns
pixel data

void **dcm_frame_destroy**(*DcmFrame* *frame)

Destroy a Frame.

Parameters

- **frame** – Frame

type **DcmFilehandle**

Part 10 File

struct **_DcmIO**

An object we can read from.

struct **_DcmIOMethods**

A set of IO methods, see dcm_io_create().

DcmIO *(***open**)(*DcmError* **error, void *client)

Open an IO object

void (***close**)(DcmIO *io)

Close an IO object

int64_t (***read**)(*DcmError* **error, DcmIO *io, char *buffer, int64_t length)

Read from an IO object, semantics as POSIX read()

int64_t (***seek**)(*DcmError* **error, DcmIO *io, int64_t offset, int whence)

Seek an IO object, semantics as POSIX seek()

DcmIO ***dcm_io_create**(*DcmError* **error, const DcmIOMethods *methods, void *client)

Create an IO object using a set of IO methods.

Parameters

- **error** – Error structure pointer
- **io** – Set of read methods
- **client** – Client data for read methods

Returns
IO object

DcmIO ***dcm_io_create_from_file**(*DcmError* **error, const char *filename)

Open a file on disk for IO.

Parameters

- **error** – Error structure pointer
- **filename** – Path to the file on disk

Returns
IO object

DcmIO ***dcm_io_create_from_memory**(*DcmError* **error, const char *buffer, int64_t length)

Open an area of memory for IO.

Parameters

- **error** – Error structure pointer
- **buffer** – Pointer to memory area
- **length** – Length of memory area in bytes

Returns
IO object

void **dcm_io_close**(DcmIO *io)

Close an IO object.

Parameters

- **io** – Pointer to IO object

int64_t **dcm_io_read**(*DcmError* **error, DcmIO *io, char *buffer, int64_t length)

Read from an IO object.

Read up to length bytes from the IO object. Returns the number of bytes read, or -1 for an error. A return of 0 indicates end of file.

Parameters

- **error** – Pointer to error object
- **io** – Pointer to IO object
- **buffer** – Memory area to read to
- **length** – Size of memory area

Returns
Number of bytes read

int64_t **dcm_io_seek**(*DcmError* **error, DcmIO *io, int64_t offset, int whence)

Seek an IO object.

Set whence to *SEEK_CUR* to seek relative to the current file position, *SEEK_END* to seek relative to the end of the file, or *SEEK_SET* to seek relative to the start.

Returns the new absolute read position, or -1 for IO error.

Parameters

- **error** – Error structure pointer
- **io** – Pointer to IO object
- **offset** – Seek offset
- **whence** – Seek mode

Returns
New read position

DcmFilehandle ***dcm_filehandle_create**(*DcmError* **error, DcmIO *io)

Create a representation of a DICOM File using an IO object.

The File object tracks information like the transfer syntax and the byte ordering.

Parameters

- **error** – Error structure pointer
- **io** – IO object to read from

Returns
filehandle

DcmFilehandle ***dcm_filehandle_create_from_file**(*DcmError* **error, const char *filepath)

Open a file on disk as a DcmFilehandle.

Parameters

- **error** – Error structure pointer
- **filepath** – Path to the file on disk

Returns
filehandle

DcmFilehandle ***dcm_filehandle_create_from_memory**(*DcmError* **error, const char *buffer, int64_t length)

Open an area of memory as a DcmFilehandle.

Parameters

- **error** – Error structure pointer
- **buffer** – Pointer to memory area
- **length** – Length of memory area in bytes

Returns
filehandle

void **dcm_filehandle_destroy**(*DcmFilehandle* *filehandle)

Destroy a Filehandle.

Parameters

- **filehandle** – File

const *DcmDataSet* ***dcm_filehandle_get_file_meta**(*DcmError* **error, *DcmFilehandle* *filehandle)

Get File Meta Information from a File.

Reads the File Meta Information and saves it in the File handle. Returns a reference to this internal copy of the File Meta Information.

The return result must not be destroyed. Make a clone of it with *dcm_dataset_clone()* if you need it to remain valid after closing the File handle.

After calling this function, the filehandle read point is always positioned at the start of the File metadata.

It is safe to call this function many times.

Parameters

- **error** – Pointer to error object
- **filehandle** – Pointer to file handle

Returns
File Meta Information

const char ***dcm_filehandle_get_transfer_syntax_uid**(const *DcmFilehandle* *filehandle)

Get Transfer Syntax UID for a fileahndle.

Parameters

- **filehandle** – File

Returns
UID of the transfer syntax for this File.

DcmDataSet ***dcm_filehandle_read_metadata**(*DcmError* **error, *DcmFilehandle* *filehandle, const uint32_t *stop_tags)

Read metadata from a File.

Read slide metadata, stopping when one of the tags in the stop list is seen. If the stop list pointer is NULL, it will stop on any of the pixel data tags.

The return result must be destroyed with *dcm_dataset_destroy()*.

After calling this function, the filehandle read point is always positioned at the tag that stopped the read. You can call this function again with a different stop set to read more of the metadata.

Parameters

- **error** – Pointer to error object
- **filehandle** – File
- **stop_tags** – NULL, or Zero-terminated array of tags to stop on

Returns
metadata

const *DcmDataSet* ***dcm_filehandle_get_metadata_subset**(*DcmError* **error, *DcmFilehandle* *filehandle)

Get a fast subset of metadata from a File.

Gets a subset of the File's metadata and saves it in the File handle. Returns a reference to this internal copy of the File metadata.

The subset is the part of the DICOM metadata that can be read quickly. It is missing tags such as PerFrameFunctionalGroupSequence. Use dcm_filehandle_read_metadata() if you need all file metadata.

The return result must not be destroyed. Make a clone of it with *dcm_dataset_clone()* if you need it to remain valid after closing the File handle.

After calling this function, the filehandle read point is always positioned at the tag that stopped the read.

It is safe to call this function many times.

Parameters

- **error** – Pointer to error object
- **filehandle** – File

Returns
metadata

bool **dcm_filehandle_prepare_read_frame**(*DcmError* **error, *DcmFilehandle* *filehandle)

Read everything necessary to fetch frames from the file.

Scans the PixelData sequence and loads the PerFrameFunctionalGroupSequence, if present.

This function will be called automatically on the first call to *dcm_filehandle_read_frame_position()* or *dcm_filehandle_read_frame()*. It can take some time to execute, so it is available as a separate function call in case this delay needs to be managed.

After calling this function, the filehandle read point is always positioned at the PixelData tag.

It is safe to call this function many times.

Parameters

- **error** – Pointer to error object
- **filehandle** – File

Returns
true on success

DcmFrame ***dcm_filehandle_read_frame**(*DcmError* **error, *DcmFilehandle* *filehandle, uint32_t frame_number)

Read an individual Frame from a File.

Frames are numbered from 1 in the order they appear in the PixelData element.

Parameters

- **error** – Pointer to error object
- **filehandle** – File
- **index** – One-based frame number

Returns
Frame

DcmFrame ***dcm_filehandle_read_frame_position**(*DcmError* **error, *DcmFilehandle* *filehandle, uint32_t column, uint32_t row)

Read the frame at a position in a File.

Read a frame from a File at a specified (column, row), numbered from zero. This takes account of any frame positioning given in PerFrameFunctionalGroupSequence.

If the frame is missing, perhaps because this is a sparse file, this function returns NULL and sets the error *DCM_ERROR_CODE_MISSING_FRAME*. Applications can detect this and render a background image.

Parameters

- **error** – Pointer to error object
- **filehandle** – File
- **column** – Column number, from 0
- **row** – Row number, from 0

Returns
Frame

bool **dcm_filehandle_print**(*DcmError* **error, *DcmFilehandle* *filehandle)

Scan a file and print the entire structure to stdout.

Parameters

- **error** – Pointer to error object
- **filehandle** – File

Returns
true on successful parse, false otherwise.

CHAPTER

FIVE

COMMAND LINE TOOLS

5.1 dcm-dump

The `dcm-dump` command line tool reads the metadata of a DICOM Data Set stored in a DICOM Part10 file and prints the metadata to standard output:

```
dcm-dump /path/to/file.dcm | grep -e Modality -e ImageType
```

Refer to the man page of the tool for further instructions:

```
man dcm-dump
```

5.2 dcm-getframe

The `dcm-getframe` command line tool will read numbered frames from a DICOM file and print then to *stdout*. Use the `-o` flag to write to a file instead. Frames are numbered from 1 in the order they appear in the PixelData sequence.

```
dcm-getframe /path/to/file.dcm 12 > x.jpg
```

Refer to the man page of the tool for further instructions:

```
man dcm-getframe
```

CHAPTER

SIX

CONTRIBUTING

6.1 Coding style

Source code should be written following the K&R (Kernighan & Ritchie) style with a few modifications.

- Line length: max 80 characters
- Indentation: 4 spaces (no tabs)
- Braces:
  - Functions: opening brace at next line
  - Control statements (mandatory): opening brace at same line
- Spacing:
  - 2 lines between function definitions
  - 1 line between logical blocks within functions (and between variable declarations and definitions)
- Comments:
  - Documentation of functions and other symbols: balanced, multi-line `/** ... */` comments in reStructuredText format using field lists `:param:` and `:return:` to document function parameters and return values, respectively, and `:c:member:`, `:c:func:`, `:c:macro:`, `:c:struct:`, `:c:union:`, `:c:enum:`, `:c:enumerator:`, `:c:type:`, `:c:expr:`, `:c:var:` from the Sphinx C domain directive to cross-reference other C language constructs or to insert a C expression as inline code.
  - Inline comments in function body: single-line `//` C++ style comments
- Naming conventions:
  - Data structures (`struct` or `enum`) and types are named using upper camel case (e.g., `DcmDataSet`), while functions are named using all lower case with underscores (e.g., `dcm_dataset_create()`).
  - Names of `external` functions, data structures, and types that are declared in the `dicom.h` header file are prefixed with `dcm_` or `Dcm`. Names of `static` functions, types, or data structures declared in `*.c` files are never prefixed.

6.2 Interface

The library exposes an "object-oriented" application programming interface (API), which provides data structures and functions to store, access, and manipulate the data.

To facilitate portability, the `dicom.h` header file is restricted to

- C99 version of the standard (C89 + Boolean type from `stdbool.h` + fixed-width integer types from `stdint.h`/`inttypes.h`)
- Opaque data types
- Clear, exact-width integer types (`int16_t`, `int32_t`, `int64_t`, `uint16_t`, `uint32_t`, and `uint64_t`)
- Minimal use of enums

6.3 Implementation

The `dicom-data.c` (Part 5), `dicom-dict.c` (Part 6), and `dicom-file.c`
and `dicom-parse.c` (Part 10) are implemented based on the C11 version of the standard.

The Data Set and Sequence data structures are implemented using the battletested uthash headers.

6.4 Documentation

Documentation is written in reStructuredText format and HTML documents are autogenerated using Sphinx. API documentation is automatically extracted from the comments in the source code in the `dicom.h` header file via the Hawkmoth Sphinx C Autodoc extension, which relies on Clang to parse C code.

Documentation files are located under the `doc/source` directory of the repository. To build the documentation, install `libclang` development headers and the Python `venv` module, then build with `meson`:

```
meson compile -C builddir html
```

The generated documentation files will then be located under the `builddir/html` directory. The `builddir/html/index.html` HTML document can be rendered in the web browser.

6.5 Testing

Unit test cases are defined and run using check.

Test files are located under `/tests` and can be built and run using `meson`:

```
meson test -C builddir
```

6.6 Dynamic analysis

The source code can be analysed using valgrind.

For example:

```
valgrind --leak-check=full dcm-dump data/test_files/sm_image.dcm
```

Unit testing and dynamic analysis can also be performed using the provided *Dockerfile* (located in the root of the repository):

```
docker build -t dcm-testing .
docker run dcm-testing
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

D