Doubango AI

State of the art **MRZ/MRTD (Machine Readable Zone/Travel Documents)** implementation for embedded devices (ARM) and desktops (x86) using deep learning
https://github.com/DoubangoTelecom/ultimateMRZ-SDK

Table of Contents

This is a short technical guide to help developers and integrators take the best from our **MRZ / MRTD** (**M**achine **R**eadable **Z**one / **T**ravel **D**ocuments) SDK. You don't need to be a developer or expert in deep learning to understand and follow the recommendations defined in this guide.

1 Intro

This is state-of-the-art **Machine Readable Zone / Travel Documents (MRZ / MRTD)** dectector and recognizer using deep learning.

Unlike other solutions you can find on the web, you don't need to adjust the camera/image to define a **R**egion **O**f **I**nterest (ROI). We also don't try to use small ROI to decrease the processing time or false-positives. The whole image (up to 4K supported) is processed and every pixel is checked. No matter if the MRZ lines are **small, far away, blurred, partially occluded, skewed** or **slanted**, our implementation can accurately detect and recognize every character.

The detector is agnostic and doesn't decode (recognize/OCR) the text to check it against some pre-defined rules (regular expressions) which means **we support all MRZ types** (Travel Documents 1/2/3, MRVA, MRVB...) regardless the font, content, shape or country. You can use our implementation to extract information from **local identity cards, passports, enhanced driver licenses, visas, resident cards...**

In addition to being highly accurate our implementation is very fast and can run at up to **50fps on ARM devices** (iPhone7) and 1**14fps on x86_64** (Core i7-4790K) using HD images (720p). See benchmark section for more information.

The Github repository contains the SDK binaries for **Android**, **iOS**, **Raspberry Pi** and **Windows**. You can contact us if you want binaries for other platforms.

Don't take our word for it, come check our implementation. **No registration, license key or internet connection is needed**, just clone the code from Github and start coding/testing: https://github.com/DoubangoTelecom/ultimateMRZ-SDK. Everything runs on the device, no data is leaving your computer. The code released on Github comes with many ready-to-use samples to help you get started easily. You can also check our online cloud-based implementation (no registration required) at https://www.doubango.org/webapps/mrz/ to check out the accuracy and precision before starting to play with the SDK.

2 Supported countries

As explained in the intro, we don't try to parse the MRZ zone to check the validity. Our neural network works like a human brain to detect the MRZ lines regardless the content. The big advantage with such method is that we don't need to define a list of supported countries or formats. **All countries and formats are supported.**

Like any implementation there could be errors when the provided image isn't clear enough. This is where you'll need to check the characters against some regular expressions. The next sections explain how to reach **100% accuracy** using regular expressions and check digits validation.

3 Architecture overview

3.1 Supported operating systems

We support any OS with a C++11 compiler. The code has been tested on **Android**, **iOS**, **Windows**, **Linux**, **Raspberry Pi 4** and many custom embedded devices (e.g. scanners).

The Github repository (https://github.com/DoubangoTelecom/ultimateMRZ-SDK) contains binaries for **Android**, **Raspberry Pi, Windows**, **Linux** and **iOS** as reference code to allow developers to test the implementation. These reference implementations come with **Java**, **Obj-C**, **Python** and **C++** APIs. The API is common to all operating systems which means you can develop and test your application on **Android**, **Raspberry Pi**, **Windows**, **Linux** or **iOS** and when you're ready to move forward we'll provide the binaries for your OS.

3.2 Supported CPUs

We officially support any ARM32 (**AArch32**), ARM64 (**AArch64**), **x86** and **x86_64** architecture. The SDK have been tested on all these CPUs.

MIPS32/64 may work but haven't been tested and would be horribly slow as there is no SIMD acceleration written for these architectures.

Almost all computer vision functions are written using assembler and accelerated with SIMD code (**NEON**, **SSE** and **AVX**). Some computer vision functions have been open sourced and shared in CompV project available at https://github.com/DoubangoTelecom/CompV.

3.3 Supported GPUs

We support any **OpenCL 1.2+** compatible GPU for the computer vision and OCR parts.

In addition to being GPGPU accelerated the implementation is SIMD accelerated. The GPU implementation requires support for 64-bit floating point math (**cl_khr_fp64** extension) which is not available on most of the **ARM Mali GPUs**. On such devices the code is massively multi-treaded and accelerated using **assembler code** and **NEON** instructions.

When you run the code on GPU devices without support for **cl_khr_fp64** extension then, you'll have the next message:

```
W org.doubango.compv: **[COMPV WARN]: function: "newObj()"
W org.doubango.compv: file: "..\source\ml\ultimate_base_ml_predict_rbf.cxx"
W org.doubango.compv: line: "153"
W org.doubango.compv: message: [UltBaseMachineLearningPredictRBF] GPGPU instance
requested but failed as double precision extension (cl_khr_fp64) is missing
```

You can safely ignore the warning.

3.4 Supported programming languages

The code was developed using C++11 and assembler but the API (Application Programming Interface) has many bindings thanks to SWIG.

Bindings: **ANSI-C**, **C++**, **C#**, **Java**, **ObjC**, **Swift**, **Perl**, **Ruby** and **Python**.

3.5 Supported raw formats

We supports the following image/video formats: **RGBA32, BGRA32, RGB24**, **NV12**, **NV21**, **Y(Grayscale), YUV420P**, **YVU420P**, **YUV422P** and **YUV444P**. NV12 and NV21 are semi-planar formats also known as **YUV420SP**. Any modern camera will support at least one of these format.

The list of supported formats is wide enough to make sure any camera will work.

3.6 Optimizations

The SDK contains the following optimizations to make it run as fast as possible:

- Hand-written assembler (YASM for x86 and GNU ASM for ARM)
- SIMD (SSE, AVX, NEON) using intrinsics or assembler
- GPGPU (OpenCL 1.2+) acceleration
- Massively multithreaded
- Smart multithreading (minimal context switch, no forking, no false-sharing, no boundaries crossing...)
- Smart memory access (data alignment, cache pre-load, cache blocking, non-temporal load/store for minimal cache pollution, smart reference counting...)
- Fixed-point math
- 8-bit Quantization
- ... and many more

Many functions have been open sourced and included in CompV project: https://github.com/DoubangoTelecom/CompV. More functions from deep learning parts will be open sourced in the coming months. You can contact us to get some closed-source code we're planning to open.

3.7 Thread safety

All the functions in the SDK are thread safe which means you can invoke them in concurrent from multiple threads. But, you should not do it for many reasons:

- The SDK is already massively multithreaded in an efficient way (see the threading model section).
- You'll end up saturating the CPU and making everything run slower. The threading model makes sure the SDK will never use more threads than the number of virtual CPU cores (no forking). Calling the engine from different threads will break this rule as we cannot control the threads created outside the SDK.
- Unless you have access to the private API the engine uses a single context which means concurrent calls are locked when they try to write to a shared resource.

4 Configuration options

The configuration options are provided when the engine is initialized and **they are case-sensitive.**

Name	Type	values	Description

`debug_level`	`STRING`	`verbose`
		`info`
		`warn`
		`error`
		`fatal`
			Defines the debug level to output on the console.
			You should use `verbose` for diagnostic, `info` in development stage and `warn` in production.
			Default: `info`

`debug_write_input_image_enabled`	`BOOLEAN`	`true` `false`	Whether to write the transformed input image to the disk. This could be useful for debugging. Default: `false`
`debug_internal_data_path`	`STRING`	`Folder path`	Path to the folder where to write the transformed input image. Used only if `debug_write_input_image_enabled` is `true`. Default: `""`
`license_token_file`	`STRING`	`File path`	Path to the file containing the license token. First you need to generate a Runtime Key using `requestRuntimeLicenseKey()` function then activate the key to get a token. You should use `license_token_file` or `license_token_data` but not both.
`license_token_data`	`STRING`	`BASE64`	Base64 string representing the license token. First you need to generate a Runtime Key using `requestRuntimeLicenseKey()` function then activate the key to get a token. You should use `license_token_file` or `license_token_data` but not both.
`num_threads`	`INTEGER`	`Any`	Defines the maximum number of threads to use. You should not change this value unless you know what you're doing. Set to −1 to let the SDK choose the right value. The right value the SDK will choose will likely be equal to the number of virtual cores. For example, on an octa-core device the maximum number of threads will be #8, on quad-core it will be #4. Default: −1
`gpgpu_enabled`	`BOOLEAN`	`true`	Whether to enable GPGPU computing. This will enable or disable GPGPU computing on

		`false`	the computer vision and deep learning libraries. On ARM devices this flag will be ignored when fixed-point (integer) math implementation exist for a well-defined function. For example, this function will be disabled for the bilinear scaling as we have a fixed-point SIMD accelerated implementation: https://github.com/DoubangoTelecom/compv/blob/master/base/image/asm/arm/compv_image_scale_bilinear_arm64_neon.S . Same for many deep learning parts as we're using QINT8 quantized inference. This option could also be ignored when the memory transfer time is high compared to the computation time. Default: `true`
`gpgpu_workload_balancing_enabled`	`BOOLEAN`	`true` `false`	A device contains a CPU and a GPU. Both can be used for math operations. This option allows using both units. On some devices the CPU is faster and on other it's slower. When the application starts, the work (math operations to perform) is equally divided: 50% for the CPU and 50% for the GPU. Our code contains a profiler to determine which unit is faster and how fast (percentage) it is. The profiler will change how the work is divided based on the time each unit takes to complete. This is why this configuration entry is named "workload balancing". Default: `false` for x86 and `true` for ARM
`assets_folder`	`STRING`	`Folder path`	Path to the folder containing the configuration files and deep learning models. Default value is the current folder. The SDK will look for the models in "`$(assets_folder)/models`" folder and configuration files in "`$(assets_folder)`". Default: `.`
`roi`	`FLOAT[4]`	`Any`	Defines the Region Of Interest (ROI) for the detector. Any pixels outside the region of interest will be ignored by the detector. Defining an WxH region of interest instead of resizing the image at WxH is very important as you'll keep the same quality when you define a ROI while you'll lose in quality when using the later. Format: `[left, width, top, height]`

			Default: `[0.f, 0.f, 0.f, 0.f]`

`segmenter_accuracy`	`STRING`	`veryhigh`
		`high`
		`medium`
		`low`
		`verylow`
			Before calling the classifier to determine whether a zone contains a MRZ line we need to segment the text using multi-layer segmenter followed by clustering. The multi-layer segmenter uses **hysteresis** for the voting process using a [min, max] double thresholding values. This configuration entry defines how low the thresholding values should be. Lower the values are, higher the number of fragments will be and higher the recall will be. High number of fragments means more data to process which means more CPU usage and higher processing time.
			Default: `high`

`interpolation`	`STRING`	`nearest`
		`bilinear`
		`bicubic`
			Defines the interpolation method to use when pixels are scaled, deskewed or deslanted. `bicubic` offers the best quality but is slow as there is no SIMD or GPU acceleration yet. `bilinear` and `nearest` interpolations are multithreaded and SIMD accelerated. For most scenarios `bilinear` interpolation is good enough to provide high accuracy/precision results while the code still runs very fast.
			Default: `bilinear`

`min_num_lines`	`INTEGER`	`[1, inf]`	Defines the minimum number of MRZ lines needed to form a valid zone. For example, this value must be 2 for passports (TD3 format) and visas (MRVA and MRVB formats). Default: `2`
`min_score`	`FLOAT`	`[0.f, 1.f]`	Defines a threshold for the recognition score/confidence. Any recognition with a score below that threshold will be ignored/removed. This value could be used to filter the false-positives and improve the precision. Low value will lead to high recall and low precision while a high value means the opposite. Range: `[0.f, 1.f]` Default: `0.0f` `0.f` being poor confidence and `1.f` excellent confidence.

5 Sample applications

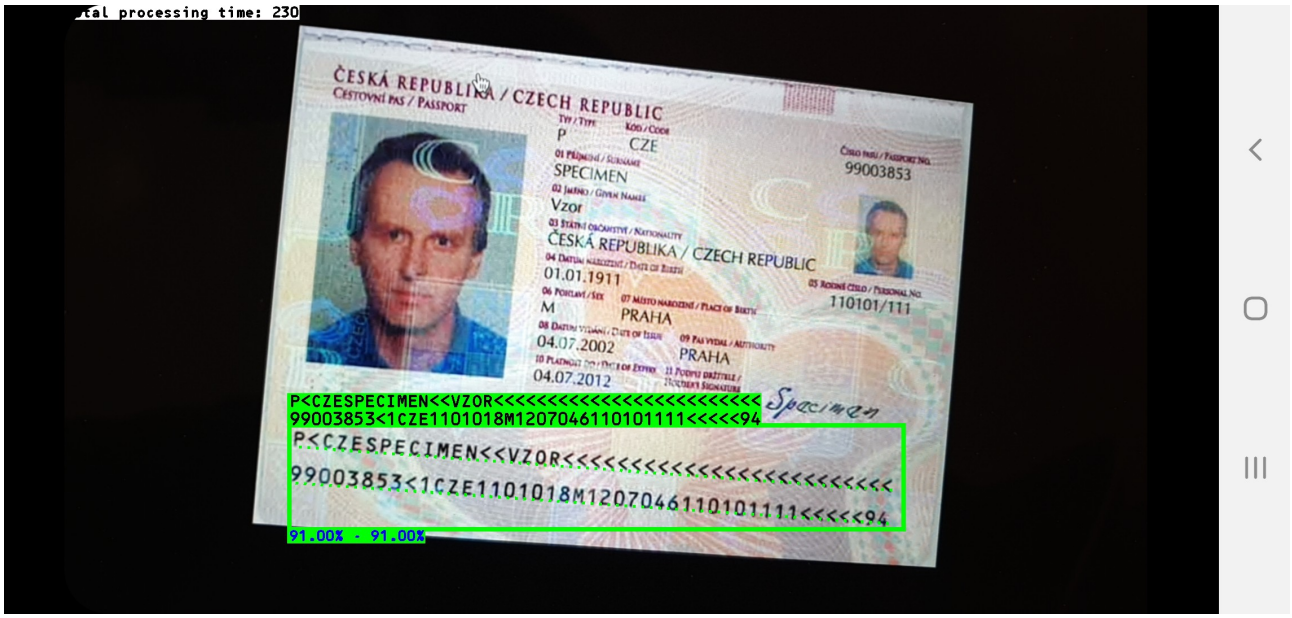
The source code comes with #4 sample applications: **Benchmark**, **VideoRecognizer**, **Parser** and **Validation**. All sample applications are open source and don't require registration or license key.

5.1 Benchmark

This application is used to check everything is ok and running as fast as expected. The information about the maximum frame rate (5**0fps**) on iPhone7 devices could be checked using this application.

5.2 VideoRecognizer

This application should be used as reference code by any developer trying to add ultimateMRZ to their products. It shows how to detect and recognize MRZ lines in realtime using live video stream from the camera.



ultimateMRZ running on Android

5.3 Parser

This is a C++ sample application using regular expressions to extract the MRZ fields. If you want to extract the expiry date, document number, holder's name or birth date then, this is the right application. See next sections about the regular expressions used to parse the MRZ lines.

5.4 Validation

MRZ lines contains check digits to validate the field. This C++ sample application shows how to validate the MRZ fields to make sure the OCR result from the recognizer is valid. Using the confidence score and validation process allows reaching **100% accuracy**.

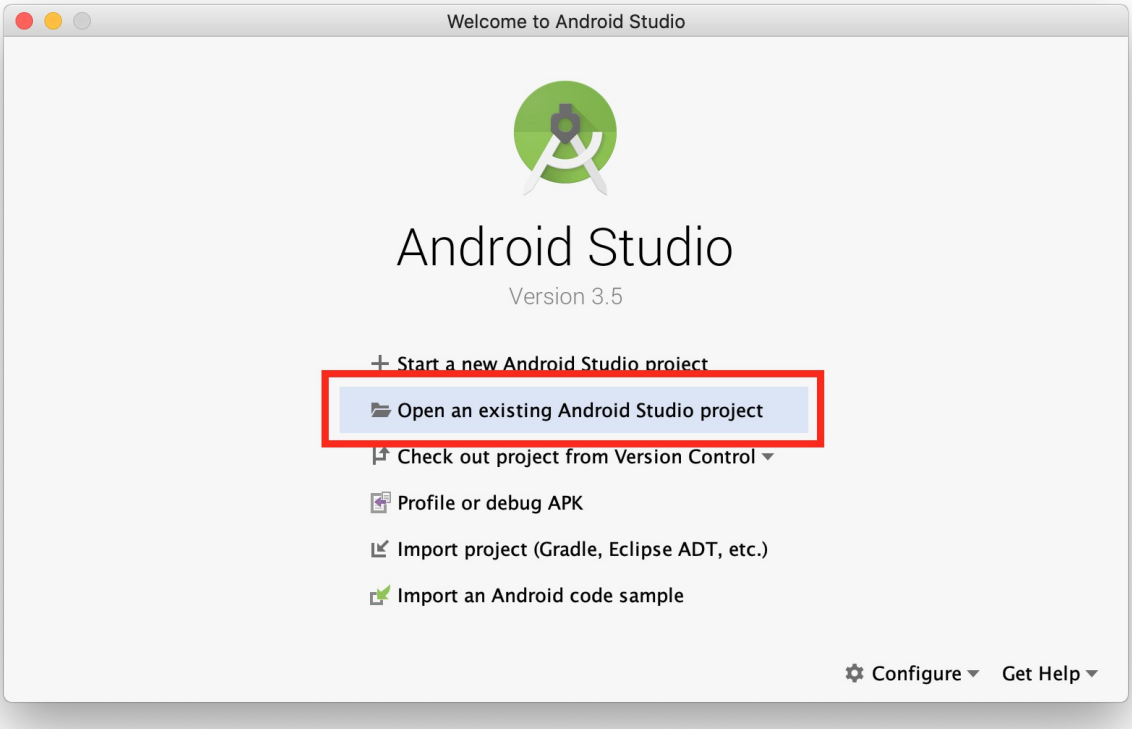
See next section about the validation process and how to use it with different languages.

5.5 Trying the samples

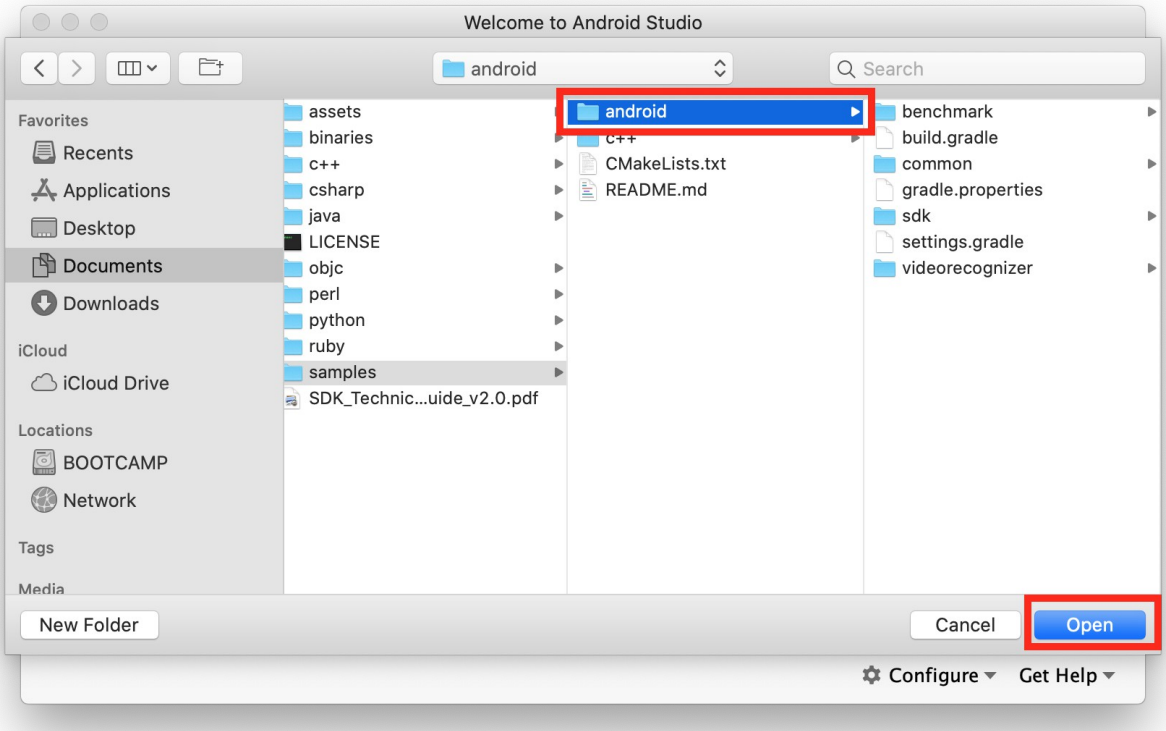
5.5.1 Android

To try the sample applications on Android:

1. Open Android Studio and select "Open an existing Android Studio project"



2. Navigate to "<ultimateMRZ-SDK>/samples", select "android" folder and click "Open"



3. Select the sample you want to try (e.g. "videorecognize"), the device (e.g. "samsung SM-G975F") and press "run".

5.5.2 iOS

To try the sample applications on iOS just open the corresponding Xcode project. For example, the Xcode project for the VideoRecognizer sample is at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/samples/ios/VideoRecognizer/VideoRecognizer.xcodeproj

5.5.3 Windows

To try the sample applications on iOS just open the corresponding Visual Studio project. For example, the VS project for the VideoRecognizer sample is at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/samples/windows/recognizer.sln.

5.5.4 Raspberry Pi, Linux and Others

For Raspberry Pi and other Linux systems you need to build the sample applications from source. More info at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/samples/c%2B%2B/README.md

6 Getting started

Please check the previously section for more information on how to use the sample applications.

6.1 Adding the SDK to your project

The Github repository contains binaries for Android, iOS, Raspberry Pi and Windows. The next sections explain how to add the SDK to an existing project.

6.1.1 Android

The SDK is distributed as an Android Studio module and you can add it as reference or you can also build it and add the AAR to your project. But, the easiest way to add the SDK to your project is by directly including the source.

In your ***build.gradle*** file add:

```
android {
        ....

   sourceSets {
      main {
         jniLibs.srcDirs += ['path-to-your-ultimateMRZ-SDK/binaries/android/jniLibs']
         java.srcDirs += ['path-to-your-ultimateMRZ-SDK/java/android']
         assets.srcDirs += ['path-to-your-ultimateMRZ-SDK/assets/models']
      }
   }
        ....
}
```

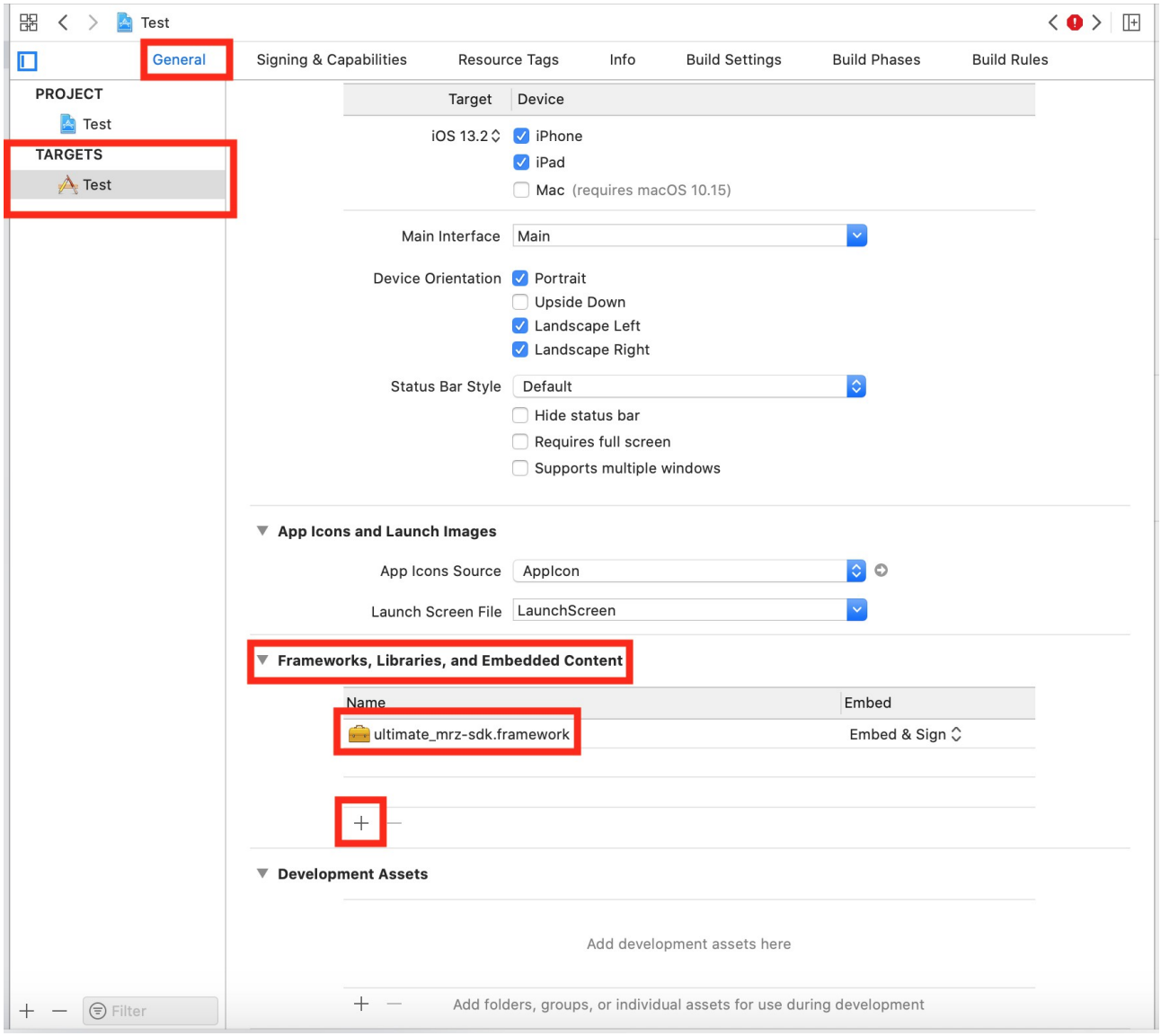
6.1.2 iOS

On iOS we provide a FAT C++ framework for Xcode. The framework has a single C++ header file which means you can easily write an Obj-C wrapper around it if you want to use Swift language. Please contact us if you want to use the framework with Swift.

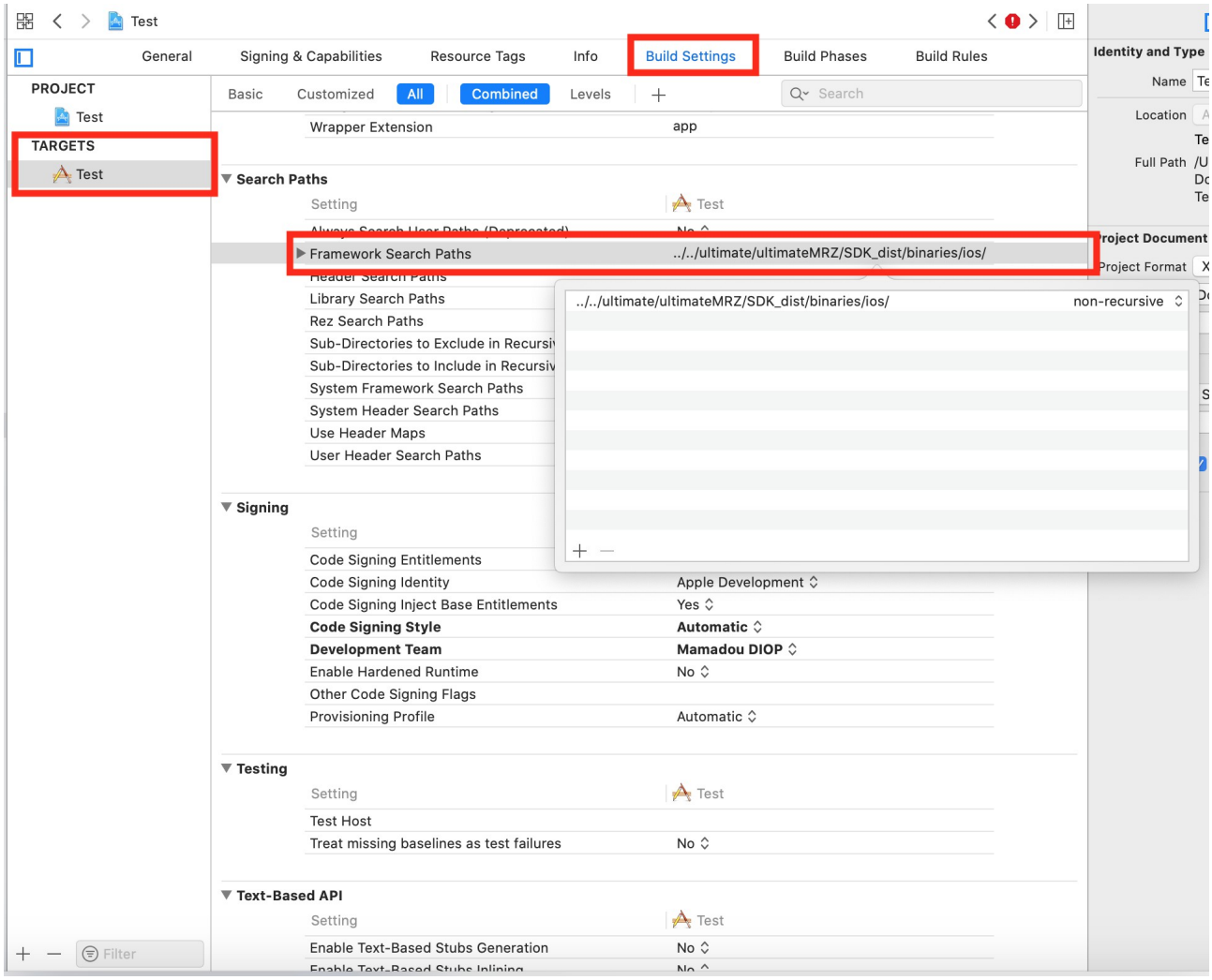
The framework is at ultimateMRZ-SDK/binaries/ios/ultimate_mrz-sdk.framework.

Make sure you're using latest Xcode version. In the next sections we're using **Xcode Version 11.3 (11C29).**

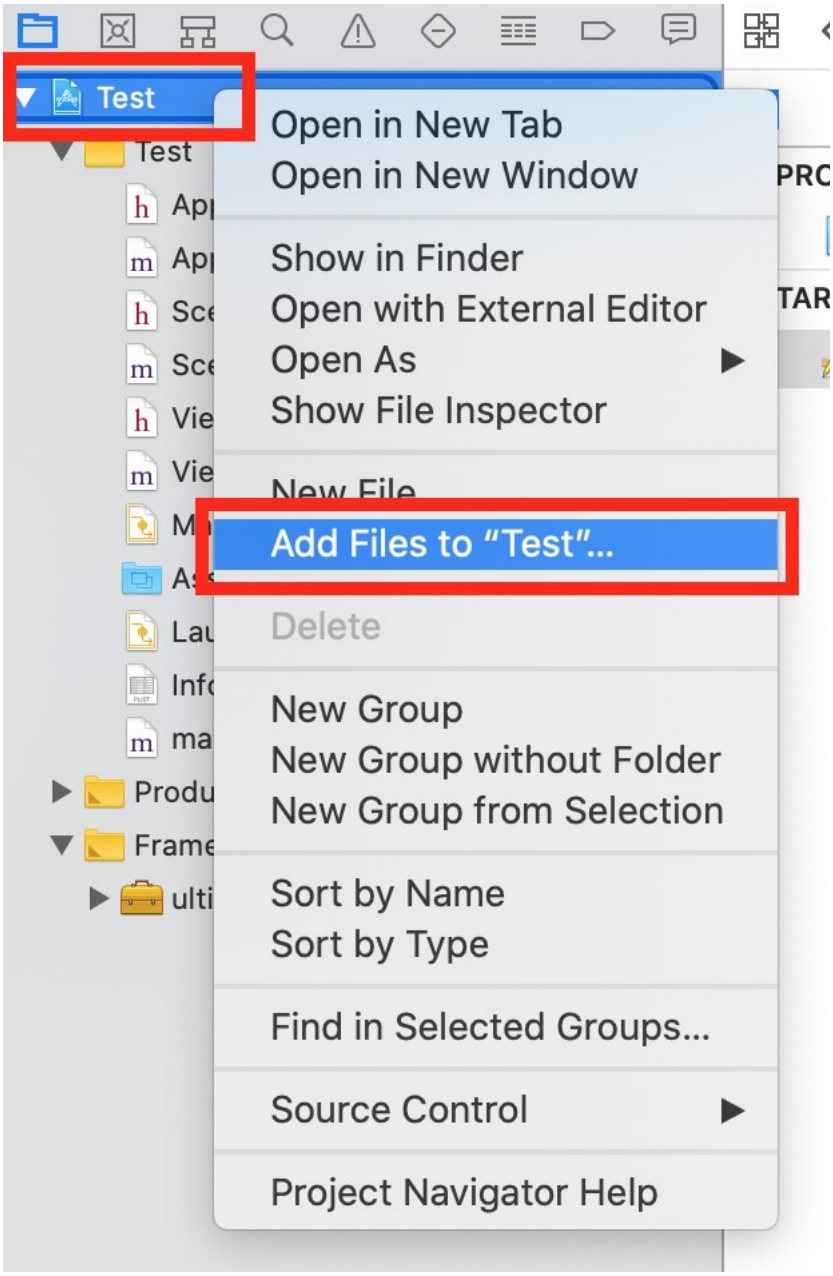
1. Select your target (let's say"Test"), select "General" section then, "Framework, Libraries, and Embedded Content" and press "+" to browse to https://github.com/DoubangoTelecom/ultimateMRZ-SDK/tree/master/binaries/ios/ultimate_mrz-sdk.framework to add the framework.



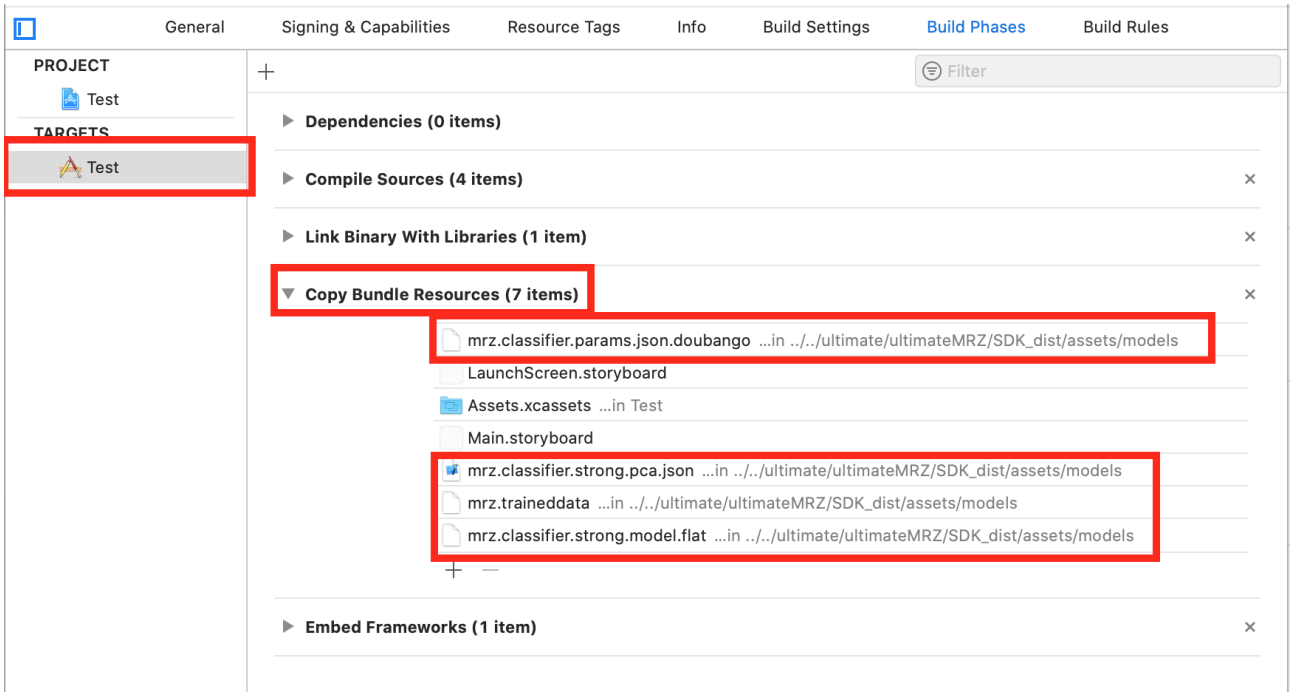
2. Select your target (let's say"Test"), select "Build Settings" section then, "Framework Search Paths" and press "+" to add path to the folder containing the framework (should be ultimateMRZ-SDK/binaries/ios)



3. Right click on your project, select "Add Files to..." and browse to "ultimateMRZ-SDK/assets/models" to select the **models** folder.



The previous action should add the models and configuration files to the bundle resources:



That's it.

6.1.3 Raspberry Pi, Windows, Linux and Others

The shared libraries are under "ultimateMRZ-SDK/binaries/**<platform>**". The header file at "ultimateMRZ-SDK/c++". You can use any C++ compiler.

6.2 Using the API

It's hard to be lost when you try to use the API as there are only 3 useful functions: init, process and deInit.

.... add sample code here

Again, please check the sample applications for more information on how to use the API.

7 Muti-threading design

No forking, minimal context switch. Doubango vs Others

8 Memory management design

This section is about the memory management design.

8.1 Memory pooling

The SDK will allocate at maximum 1/20th of the available RAM during the application lifetime and manage it using a pool. For example, if the device have 8G memory, then it will start allocating 3M memory and depending on the malloc/free requests this amount will be increased with 400M (1/20th of 8G) being the maximum. Most of the time the allocated memory will never be more than 5M.

Every memory allocation or deallocation operation (malloc, calloc, free, realloc...) is hooked which make it immediate (no delay). The application allocates and deallocates aligned memory hundreds of time every second and thanks to the pooling mechanism these operations don't add any latency.

We found it was interesting to add this section on the documentation so that the developers understand why the amount of allocated memory doesn't automatically decrease when freed. You may think there are leaks but it's probably not the case. Please also note that we track every allocated memory or object and can automatically detect leaks.

8.2 Minimal cache eviction

Thanks to the memory pooling when a block is freed it's not really deallocated but put on the top of the pool and reattributed at the next allocation request. This not only make the allocation faster but also minimize the cache eviction as the fakely freed memory is still hot in the cache.

8.3 Aligned on SIMD and cache line size

Any memory allocation done using the SDK will be aligned on 16bytes on ARM and 32bytes on x86. The data is also strided to make it cache-friendly. The 16bytes and 32bytes alignment values aren't arbitrary but chosen to make ARM NEON and AVX functions happy.

When the user provides non-aligned data as input to the SDK, then the data is unpacked and wrapped to make it SIMD-aligned. This introduce some latency. Try to provide aligned data and when choosing region of interest (ROI) for the detector try to use SIMD-aligned left bounds.

```
(left & 15) == 0; // means 16bytes aligned
(left & 31) == 0; // means 32bytes aligned
```

8.4 Cache blocking

To be filled

9 MRZ formats

MRZ / MRTD (Machine Readable Zone / Travel Documents) format is standardized by the ICAO (International Organization for Standardization) in Document 9303. The document is divided in 7 parts:

1. Machine Readable Travel Documents Part 1: Introduction
2. Machine Readable Travel Documents Part 2: Specifications for the Security of the Design, Manufacture, and Issuance of MRTDs
3. Machine Readable Travel Documents Part 3: Specifications Common to all MRTDs
4. Machine Readable Travel Documents Part 4: Specifications for Machine Readable Passports (MRPs) and other TD3 Size MRTDs
5. Machine Readable Travel Documents Part 5: Specifications for TD1 Size Machine Readable Official Travel Documents (MROTDs)
6. Machine Readable Travel Documents Part 6: Specifications for TD2 Size Machine Readable Official Travel Documents (MROTDs)
7. Machine Readable Travel Documents Part 7: Machine Readable Visas

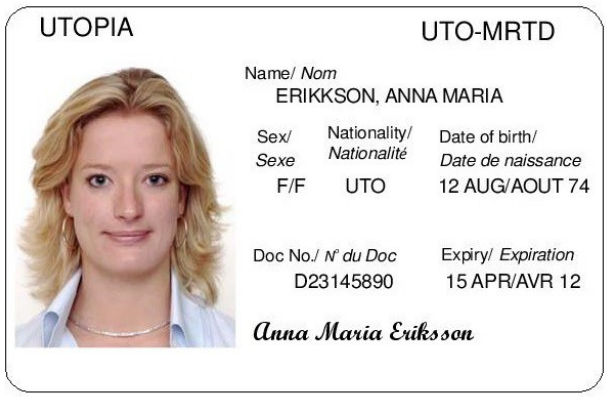
Fortunately you don't need to read all these documents to understand the MRZ / MRTD format our use our SDK. We have done the work for you.

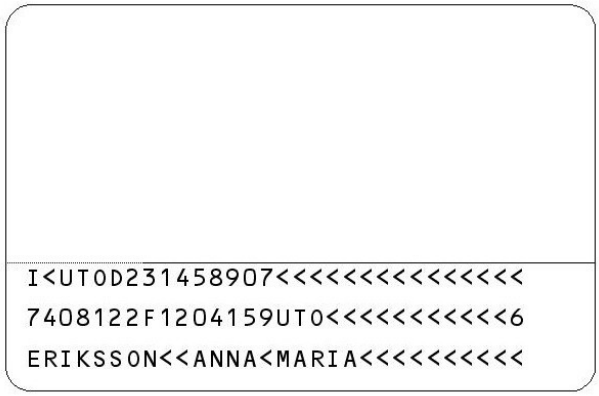
There are 3 main formats: TD1, TD2, TD3 and Visas (MRVA and MRVB).

9.1 TD1

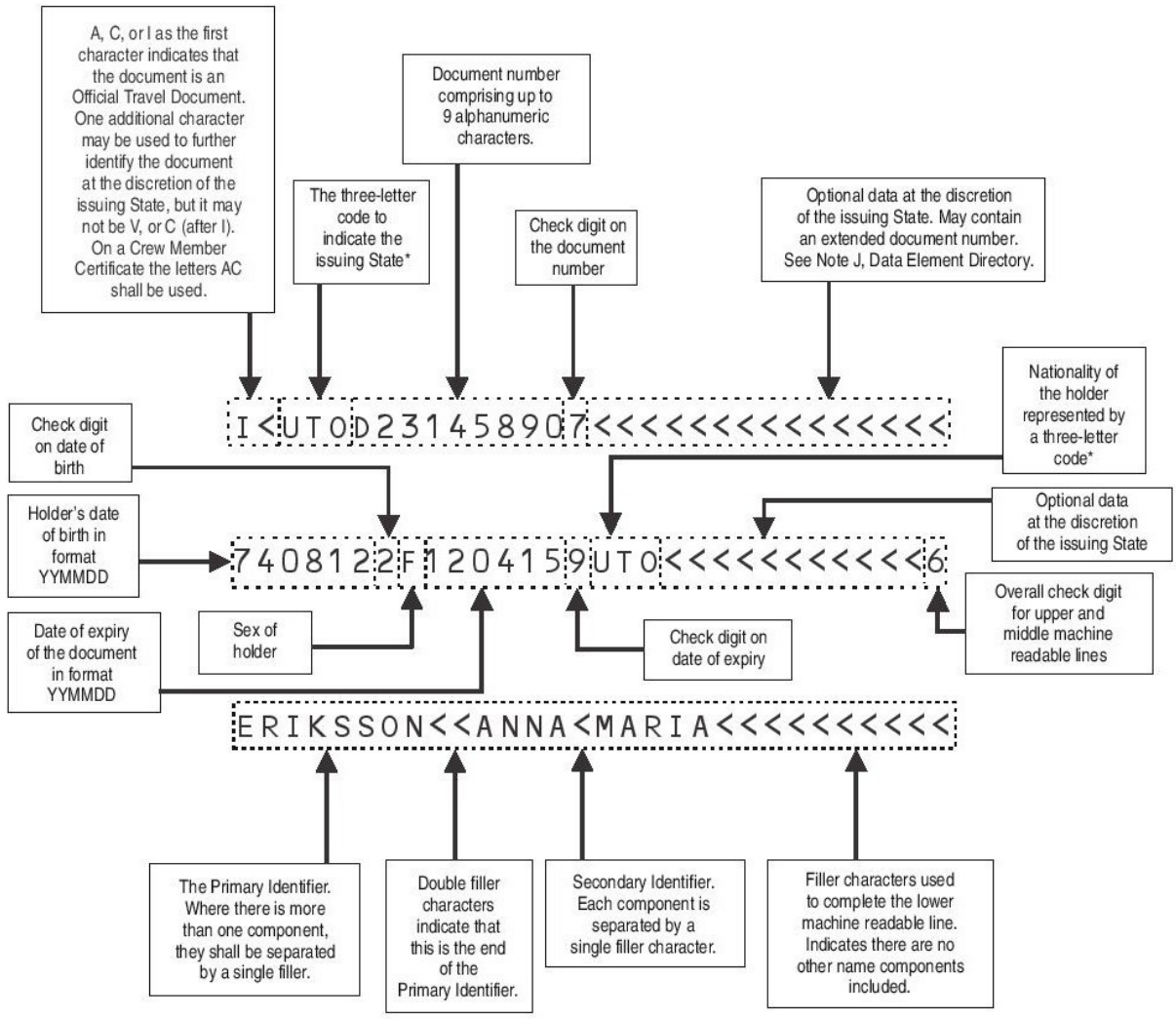
This format is defined in Machine Readable Travel Documents Part 5: Specifications for TD1 Size Machine Readable Official Travel Documents (MROTDs) document.

Here is a sample image (recto/verso):





Here is the format:



Construction of the 3-line MRZ data on a TD1 Size MROTD

Note 1.— (*) Three-letter codes are given in Doc 9303-3.
Note 2.— Dotted lines indicate data fields; these, together with arrows and comments boxes, are shown for the reader's understanding only and are not printed on the document.
Note 3.— Data is inserted into a field beginning at the first character position starting from the left. Any unused character positions shall be occupied by filler characters (<).

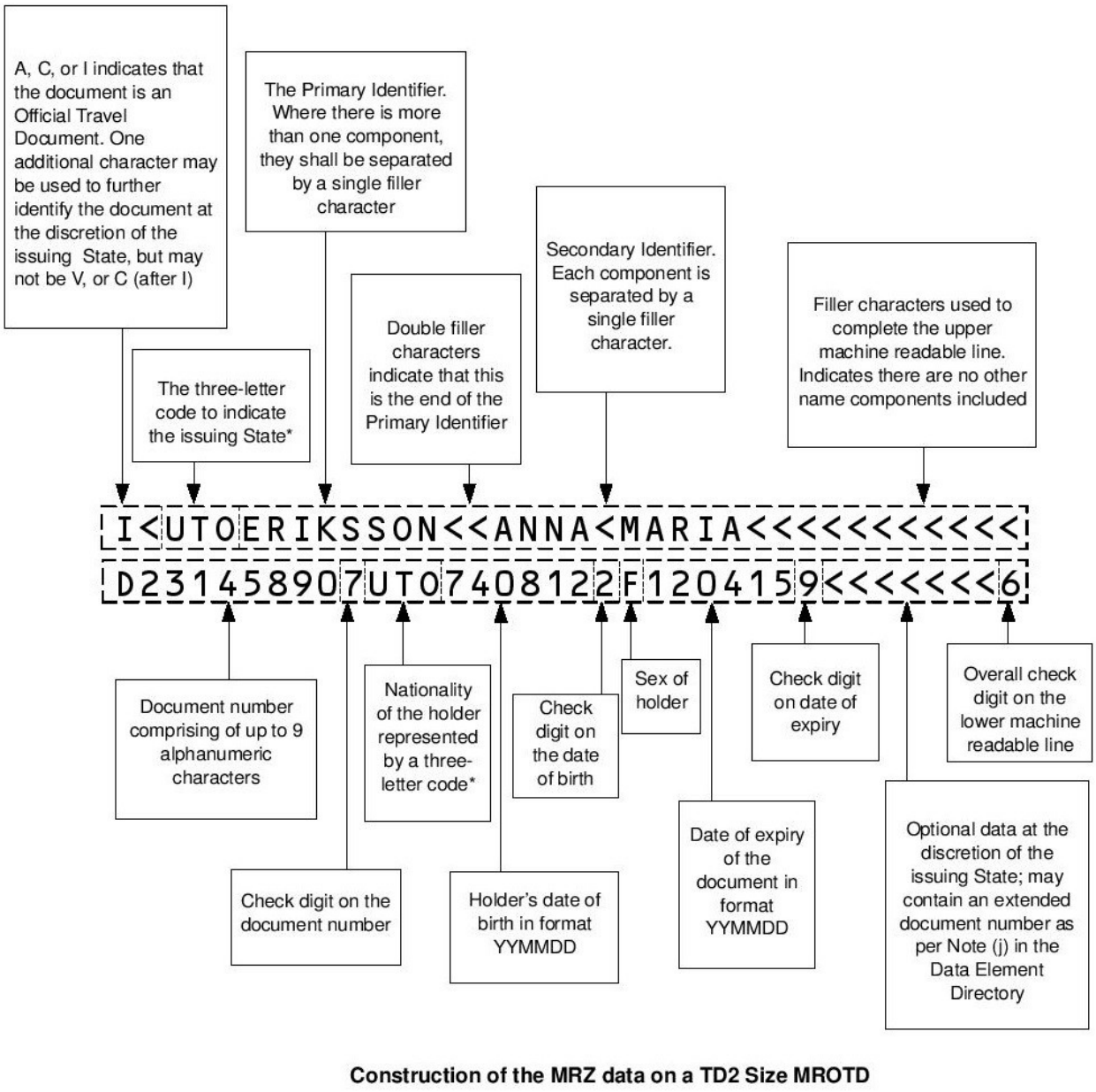
— — — — — — — —

9.2 TD2

This format is defined in Machine Readable Travel Documents Part 6: Specifications for TD2 Size Machine Readable Official Travel Documents (MROTDs) document.

Here is a sample image:

Here is the format:



Construction of the MRZ data on a TD2 Size MROTD

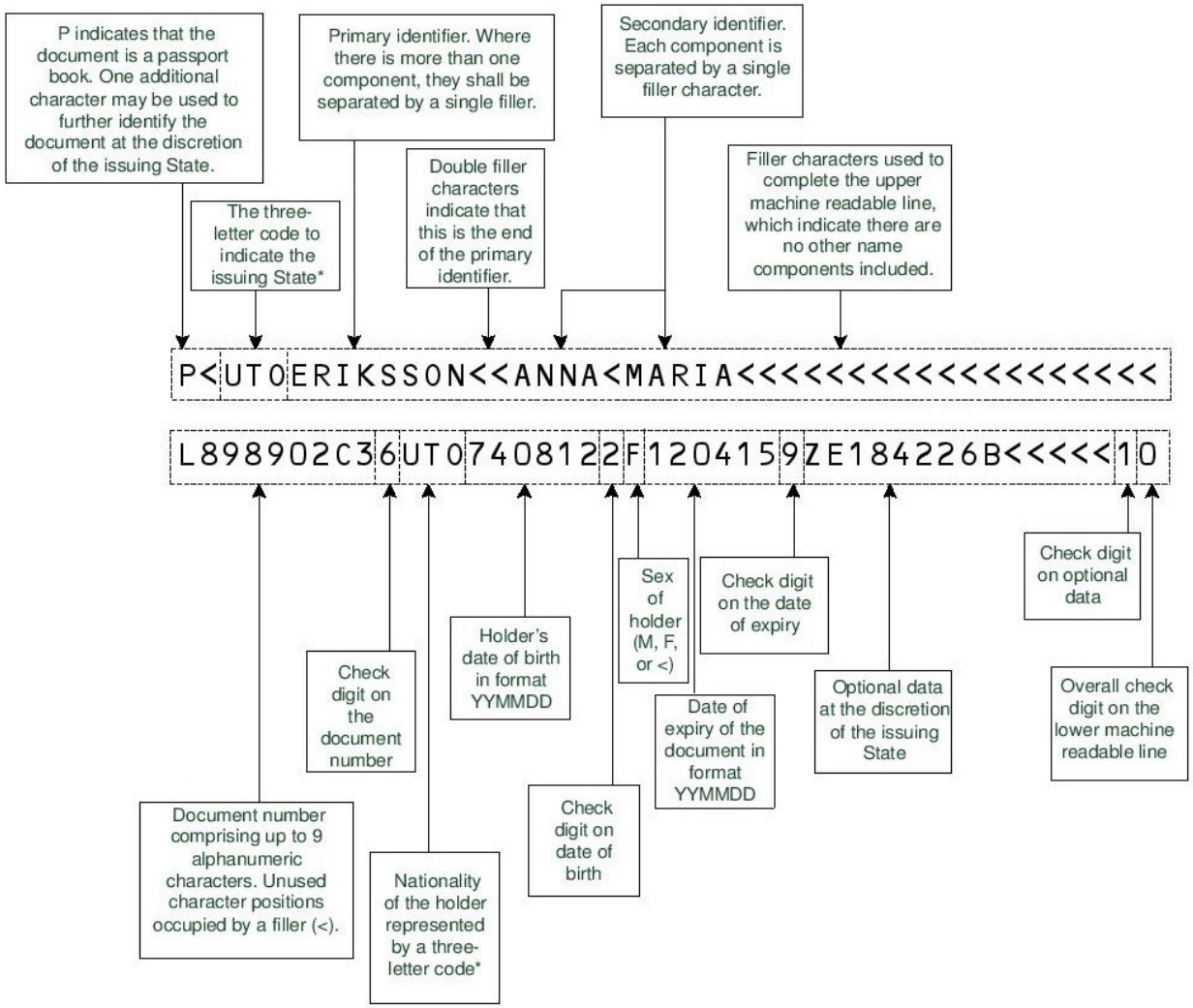
* Three-letter codes are given in Doc 9303-3.

9.3 TD3

This format is defined in Machine Readable Travel Documents Part 4: Specifications for Machine Readable Passports (MRPs) and other TD3 Size MRTDs document.

Here is a sample:

Here is the format:



Example showing the sequence and content of data elements in the MRZ

*Note 1.— * Three letter codes are given in Doc 9303-3.*

Note 2.— Dotted lines indicate data fields; these, together with arrows and comment boxes, are shown for the reader's understanding only and are not printed on the document.

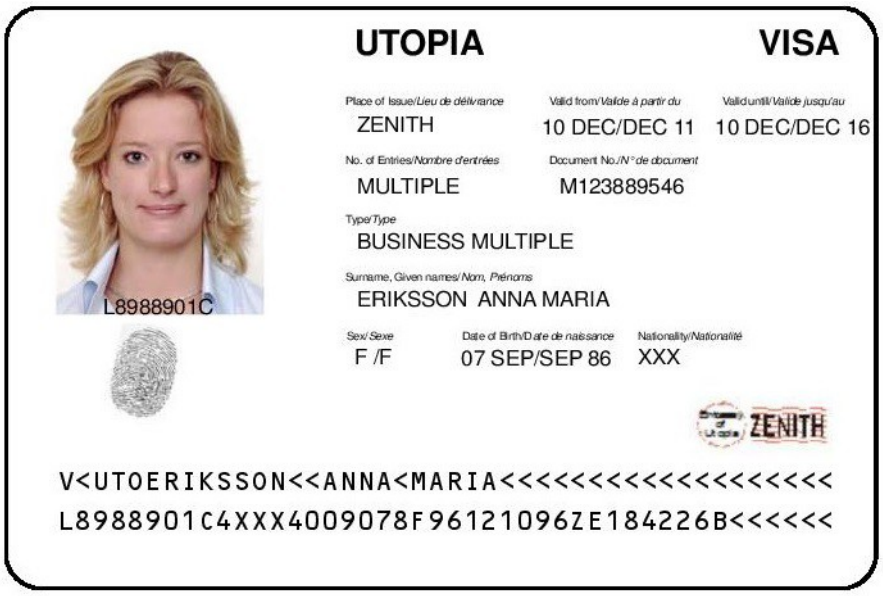
Note 3.— Data is inserted into a field beginning at the first character position starting from the left. Any unused character positions shall be occupied by filler characters (<).

9.4 Visas

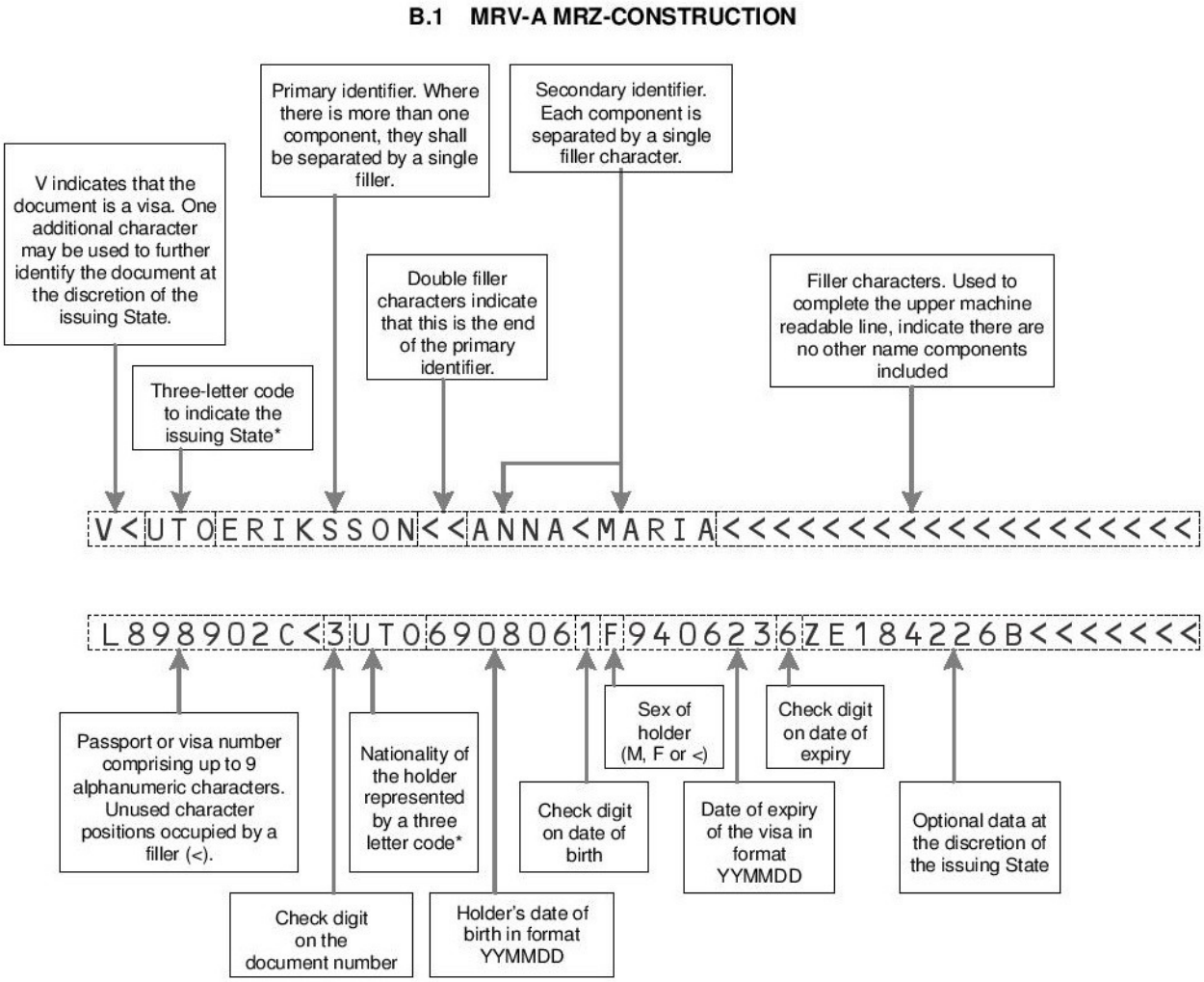
This format is defined in Machine Readable Travel Documents Part 7: Machine Readable Visas document and have two subtypes: MRVA and MRVB.

9.4.1 MRVA

Here is a sample:



Here is the format:



MRV-A MRZ construction:

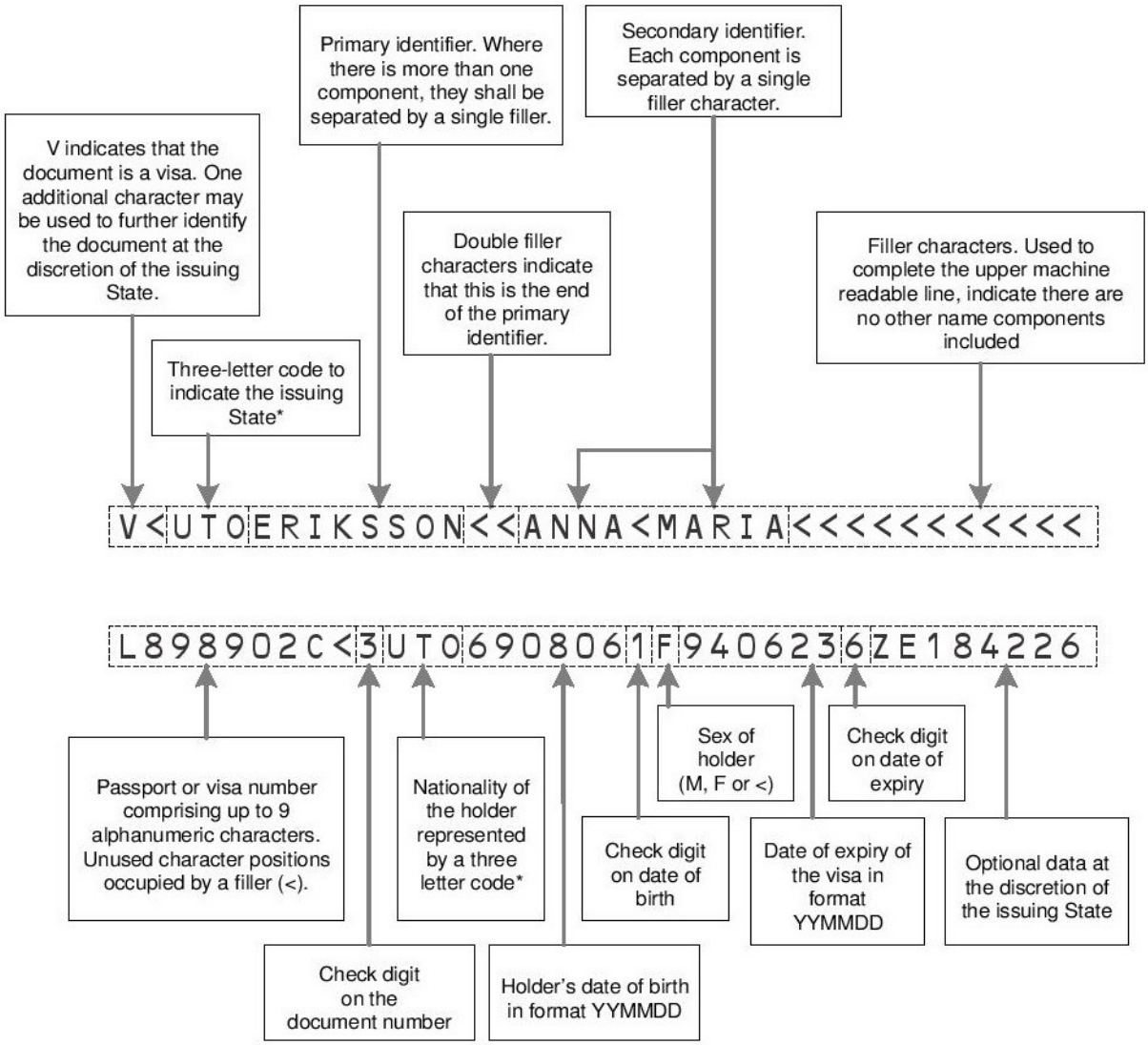
- ***Three letter codes are given in Doc 9303-3.**
- **Dotted lines indicate data fields; these, together with arrows and comment boxes, are shown for the reader's understanding only and are not printed on the document.**
- **Data are inserted into a field beginning at the first character position starting from the left. Any unused character positions shall be occupied by filler characters (<).**

9.4.2 MRVB

Here is a sample:

Here is the format:

B.2 MRV-B MRZ-Construction



MRV-B MRZ construction:

- ***Three letter codes are given in Doc 9303-3.**
- **Dotted lines indicate data fields; these, together with arrows and comment boxes, are shown for the reader's understanding only and are not printed on the document.**
- **Data are inserted into a field beginning at the first character position starting from the left. Any unused character positions shall be occupied by filler characters (<).**

10 MRZ parser

The SDK contains C++ code to parse the MRZ lines returned using the API. The sample application is at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/tree/master/samples/c++/parser. This sample application uses regular expressions which means you can easily migrate the code to C#, Python or Java (no change to the regex).

If you're dealing with non-standard formats and struggling to write the right regular expressions then, don't hesitate to contact us via our dev-group and we'll help you.

Here are some samples we'll use in the next sections:

TD1	I<UTOD231458907<<<<<<<<<<<<<<< 7408122F1204159UTO<<<<<<<<<<<6 ERIKSSON<<ANNA<MARIA<<<<<<<<<<
TD2	I<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<< D231458907UTO7408122F1204159<<<<<<<6
TD3	P<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<<<<<<<<<< L898902C36UTO7408122F1204159ZE184226B<<<<<10
MRVA	V<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<<<<<<<<<< L8988901C4XXX4009078F96121096ZE184226B<<<<<<
MRVB	V<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<< L8988901C4XXX4009078F9612109<<<<<<<<

10.1 Determine the document type

The MRZ lines (array of strings) returned by the SDK are sorted from top to bottom.

The first operation is to deteermine the document type. You can determine the type using a single line of code:

```
const MRZ_DOCUMENT_TYPE type = (lines.size() == 3 && lines.front().size() == 30) ?
      MRZ_DOCUMENT_TYPE_TD1 : ((lines.front().size() == 44 && lines.size() == 2) ?
(lines.front()[0] == 'P' ?       MRZ_DOCUMENT_TYPE_TD3 : MRZ_DOCUMENT_TYPE_MRVA) :
((lines.front().size() == 36 && lines.size() == 2) ?  (lines.front()[0] == 'V' ?
MRZ_DOCUMENT_TYPE_MRVB : MRZ_DOCUMENT_TYPE_TD2) :     MRZ_DOCUMENT_TYPE_UNKNOWN));
```

10.2 Parsing TD1 format

TD1 format has 3 lines, each has 30 characters.

10.2.1 Line 1

Regular expression:

Regular expression	
([A|C|I][A-Z0-9<]{1})([A-Z]{3})([A-Z0-9<]{9})([0-9]{1})([A-Z0-9<]{15})	
Group #1	Document type. A, C or I as the first character.
Group #2	3 letters country code.
Group #3	Document number, up to 9 alphanumeric characters.

Group #4	Check digit on the document number.
Group #5	Optional data at the discretion of the issuing state.

Sample result:

Data	
I<UTOD231458907<<<<<<<<<<<<<<<	
Group #1	I<
Group #2	UTO
Group #3	D23145890
Group #4	7
Group #5	<<<<<<<<<<<<<<<

10.2.2 Line 2

Regular expression:

Regular expression	
([0-9]{6})([0-9]{1})([M|F|X|<]{1})([0-9]{6})([0-9]{1})([A-Z]{3})([A-Z0-9<]{11})([0-9]{1})	
Group #1	Holder's date of birth in format YYMMDD.
Group #2	Check digit on the date of birth.
Group #3	Sex of holder
Group #4	Date of expiry of the document in format YYMMDD.
Group #5	Check digit on date of expiry.
Group #6	Nationality of the holder represented by a three-letter code.
Group #7	Optional data at the discretion of the issuing state.
Group #8	Overall check digit for upper and middle MRZ lines.

Sample result:

data	
7408122F1204159UTO<<<<<<<<<<6	
Group #1	740812
Group #2	2
Group #3	F

Group #4	120415
Group #5	9
Group #6	UTO
Group #7	<<<<<<<<<<<
Group #8	6

10.2.3 Line 3

Regular expression	
([A-Z0-9<]{30})	
Group #1	Names

Sample result:

data	
ERIKSSON<<ANNA<MARIA<<<<<<<<<<	
Group #1	ERIKSSON, ANNA MARIA

10.3 Parsing TD2 format

TD2 format has 2 lines, each has 36 characters.

10.3.1 Line 1

Regular expression:

Regular expression	
([A|C|I][A-Z0-9<]{1})([A-Z]{3})([A-Z0-9<]{31})	
Group #1	Document type. A, C or I as the first character.
Group #2	3 letters country code.
Group #3	Primary Identifier.

Sample result:

Data	
I<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<<	
Group #1	I<

Group #2	UTO
Group #3	ERIKSSON, ANNA MARIA

10.3.2 Line 2

Regular expression:

Regular expression	
([A-Z0-9<]{9})([0-9]{1})([A-Z]{3})([0-9]{6})([0-9]{1})([M|F|X|<]{1})([0-9]{6})([0-9]{1})([A-Z0-9<]{7})([0-9]{1})	
Group #1	Document number, up to 9 alphanumeric characters.
Group #2	Check digit on document number.
Group #3	Nationality. 3 letters country code.
Group #4	Holder's date of birth.
Group #5	Check digit on the date of birth.
Group #6	Sex of holder
Group #7	Date of expiry of the document.
Group #8	Check digit on the date of expiry.
Group #9	Optional data at the discretion of the issuing state.
Group #10	Overall check digit

Sample result:

Data	
D231458907UT07408122F1204159<<<<<<<6	
Group #1	D23145890
Group #2	7
Group #3	UTO
Group #4	740812
Group #5	2
Group #6	F
Group #7	120415
Group #8	9
Group #9	<<<<<<<
Group #10	6

10.4 Parsing TD3 format

TD3 format has 2 lines, each has 44 characters.

10.4.1 Line 1

Regular expression:

Regular expression	
(P[A-Z0-9<]{1})([A-Z]{3})([A-Z0-9<]{39})	
Group #1	Document type. P as the first character.
Group #2	3 letters country code.
Group #3	Primary Identifier.

Sample result:

Data	
P<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<<<<<<<<<<<<	
Group #1	P<
Group #2	UTO
Group #3	ERIKSSON, ANNA MARIA

10.4.2 Line 2

Regular expression:

Regular expression	
([A-Z0-9<]{9})([0-9]{1})([A-Z]{3})([0-9]{6})([0-9]{1})([M|F|X|<]{1})([0-9]{6})([0-9]{1})([A-Z0-9<]{14})([0-9]{1})([0-9]{1})	
Group #1	Document number, up to 9 alphanumeric characters.
Group #2	Check digit on document number.
Group #3	Nationality. 3 letters country code.
Group #4	Holder's date of birth.
Group #5	Check digit on the date of birth.
Group #6	Sex of holder
Group #7	Date of expiry of the document.
Group #8	Check digit on the date of expiry.
Group #9	Optional data at the discretion of the issuing state.
Group #10	Check digit on the optional data.

Group #11	Overall check digit.

Sample result:

Data	
L898902C36UT07408122F1204159ZE184226B<<<<<10	
Group #1	L898902C3
Group #2	6
Group #3	UTO
Group #4	740812
Group #5	2
Group #6	F
Group #7	120415
Group #8	9
Group #9	ZE184226B<<<<<
Group #10	1
Group #11	0

10.5 Parsing MRVA format

MRVA format has 2 lines, each has 44 characters.

10.5.1 Line 1

Regular expression:

Regular expression	
(V[A-Z0-9<]{1})([A-Z]{3})([A-Z0-9<]{39})	
Group #1	Document type. V as the first character.
Group #2	Issuing state. 3 letters country code.
Group #3	Primary Identifier.

Sample result:

Data
V<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<<<<<<<<<<<<

Group #1	V<
Group #2	UTO
Group #3	ERIKSSON, ANNA MARIA

10.5.2 Line 2

Regular expression:

Regular expression	
([A-Z0-9<]{9})([0-9]{1})([A-Z]{3})([0-9]{6})([0-9]{1})([M|F|X|<]{1})([0-9]{6})([0-9]{1})([A-Z0-9<]{16})	
Group #1	Document number, up to 9 alphanumeric characters.
Group #2	Check digit on document number.
Group #3	Nationality. 3 letters country code.
Group #4	Holder's date of birth.
Group #5	Check digit on the date of birth.
Group #6	Sex of holder
Group #7	Date of expiry of the document.
Group #8	Check digit on the date of expiry.
Group #9	Optional data at the discretion of the issuing state.

Sample result:

Data	
L8988901C4XXX4009078F96121096ZE184226B<<<<<<	
Group #1	L8988901C
Group #2	4
Group #3	XXX
Group #4	400907
Group #5	8
Group #6	F
Group #7	961210
Group #8	9
Group #9	6ZE184226B<<<<<<

10.6 Parsing MRVB format

MRVB format has 2 lines, each has 36 characters.

10.6.1 Line 1

Regular expression:

Regular expression	
(V[A-Z0-9<]{1})([A-Z]{3})([A-Z0-9<]{31})	
Group #1	Document type. V as the first character.
Group #2	Issuing state. 3 letters country code.
Group #3	Primary Identifier.

Sample result:

Data	
V<UTOERIKSSON<<ANNA<MARIA<<<<<<<<<<<	
Group #1	V<
Group #2	UTO
Group #3	ERIKSSON, ANNA MARIA

10.6.2 Line 2

Regular expression:

Regular expression	
([A-Z0-9<]{9})([0-9]{1})([A-Z]{3})([0-9]{6})([0-9]{1})([M|F|X|<]{1})([0-9]{6})([0-9]{1})([A-Z0-9<]{8})	
Group #1	Document number, up to 9 alphanumeric characters.
Group #2	Check digit on document number.
Group #3	Nationality. 3 letters country code.
Group #4	Holder's date of birth.
Group #5	Check digit on the date of birth.
Group #6	Sex of holder
Group #7	Date of expiry of the document.
Group #8	Check digit on the date of expiry.
Group #9	Optional data at the discretion of the issuing state.

Sample result:

Data
L8988901C4XXX4009078F9612109<<<<<<<<

Group #1	L8988901C
Group #2	4
Group #3	XXX
Group #4	400907
Group #5	8
Group #6	F
Group #7	961210
Group #8	9
Group #9	<<<<<<<<

11 Data validation

What is nice with the MRZ / MRTD specifications is that they contain different **check digits** to make sure that the most important fields (document number, expiry date, date of birth...) are valid. We'll explain how the validation is done in the next sections.

The SDK comes with C++ sample code to validate MRZ / MRTD data. The code is at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/tree/master/samples/c++/validation.

A **check digit** consists of a single digit computed from the other digits in a series. Check digits in the MRZ are calculated on specified numerical data elements in the MRZ. The check digits permit readers to verify that data in the MRZ is correctly interpreted.

A **special check** digit calculation has been adopted for use in MRTDs. The check digits shall be calculated on **modulus 10** with a continuously repetitive weighting of **731 731** ..., as follows:

- **Step 1.** Going from left to right, multiply each digit of the pertinent numerical data element by the weighting figure appearing in the corresponding sequential position.
- **Step 2.** Add the products of each multiplication.
- **Step 3.** Divide the sum by 10 (the modulus).
- **Step 4.** The remainder shall be the check digit.

For data elements in which the number does not occupy all available character positions, the symbol **<** shall be used to complete vacant positions and shall be given the value of zero for the purpose of calculating the check digit.

When the check digit calculation is applied to data elements containing alphabetic characters, the characters **A** to **Z** shall have the values **10** to **35** consecutively, as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Next sections explain how to implement the above algorithm using C++. The entire code could be found at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/blob/master/samples/c++/validation/main.cxx and it's used in the validation sample application.

The SDK itself doesn't contain the validation code for the simple reason that we want it to be generic to work with any MRZ format even if the data is malformed or non-standard.

11.1 Local variables and macros

Weighting:

```
static const int __Weights[] = { 7, 3, 1 };
```

Mapped values:

```
static std::map<char, int> __MappedValues;
const std::string charset = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
for (int i = 0; i < static_cast<int>(charset.size()); ++i) {
	__MappedValues[charset[i]] = i;
}
__MappedValues['<'] = 0;
```

Weighted sum:

```
#define MRZ_COMPUTE_WEIGHTED_SUM(_line_, _start_, _end_, w) { \
		for (size_t i = _start_, j = w; i <= _end_; ++i, ++j) { \
			sum += __MappedValues[(_line_)[i]] * __Weights[j % 3]; \
		} \
	}
```

Validity check:

```
#define MRZ_CHECK_VALIDITY(_line_, _start_, _end_, _check_, _ret_) { \
		const std::string __line__ = (_line_); \
		int sum = 0; \
		MRZ_COMPUTE_WEIGHTED_SUM(__line__, _start_, _end_, 0); \
		_ret_ = ((sum % 10) == __MappedValues[__line__[_check_]]); \
	}
```

11.2 Validating TD1 format

```
bool documentNumber, dateOfBirth, dateOfExpiry, upperAndMiddleLines;

MRZ_CHECK_VALIDITY(lines[0], 5, 13, 14, documentNumber);
MRZ_CHECK_VALIDITY(lines[1], 0, 5, 6, dateOfBirth);
MRZ_CHECK_VALIDITY(lines[1], 8, 13, 14, dateOfExpiry);

int sum = 0;
MRZ_COMPUTE_WEIGHTED_SUM(lines[0], 5, 29, 0);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 0, 6, 25);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 8, 14, 32);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 18, 28, 43);
upperAndMiddleLines = (sum % 10) == __MappedValues[lines[1][29]];
```

11.3 Validating TD2 format

```
bool documentNumber, dateOfBirth, dateOfExpiry, composite;

MRZ_CHECK_VALIDITY(lines[1], 0, 8, 9, documentNumber);
MRZ_CHECK_VALIDITY(lines[1], 13, 18, 19, dateOfBirth);
MRZ_CHECK_VALIDITY(lines[1], 21, 26, 27, dateOfExpiry);

int sum = 0;
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 0, 9, 0);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 13, 19, 10);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 21, 34, 17);
composite = (sum % 10) == __MappedValues[lines[1][35]];
```

11.4 Validating TD3 format

```
bool passportNumber, dateOfBirth, dateOfExpiry, personalNumber, composite;

MRZ_CHECK_VALIDITY(lines[1], 0, 8, 9, passportNumber);
MRZ_CHECK_VALIDITY(lines[1], 13, 18, 19, dateOfBirth);
MRZ_CHECK_VALIDITY(lines[1], 21, 26, 27, dateOfExpiry);
MRZ_CHECK_VALIDITY(lines[1], 28, 41, 42, personalNumber);
```

```
int sum = 0;
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 0, 9, 0);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 13, 19, 10);
MRZ_COMPUTE_WEIGHTED_SUM(lines[1], 21, 42, 17);
composite = (sum % 10) == __MappedValues[lines[1][43]];
```

11.5 Validating MRVA and MRVB formats

```
bool documentNumber, dateOfBirth, dateOfExpiry;

MRZ_CHECK_VALIDITY(lines[1], 0, 8, 9, documentNumber);
MRZ_CHECK_VALIDITY(lines[1], 13, 18, 19, dateOfBirth);
MRZ_CHECK_VALIDITY(lines[1], 21, 26, 27, dateOfExpiry);
```

12 Improving the accuracy

The code provided on Github (https://github.com/DoubangoTelecom/ultimateMRZ-SDK) comes with default configuration to make everyone almost happy. You may want to increase the speed or accuracy to match your use case.

12.1 Detector

This section explains how to increase the accuracy for the detection layer.

12.1.1 Segmenter accuracy

As explained in the configuration section the segmenter accuracy accepts 5 values (JSON strings): **`veryhigh`**, **`high`**, **`medim`**, **`low`**, and **`verylow`**. The default value is **`high`**.

If the SDK fails to detect some MRZ lines then, consider using **`veryhigh`** accuracy. With **`veryhigh`** value the detection accuracy will increase but this comes with a cost: *high CPU usage and slow detection*. We recommend increasing the image resolution and making sure that the MRZ lines are as straight as possible instead of changing the accuracy level.

12.1.2 Minimum number of MRZ lines

If you're trying to detect MRZ data with at least 2 lines which is the case for all standard formats then, say it loudly by setting the corresponding JSON configuration entry. In addition to removing some orphans (single MRZ lines) it will improve the speed. Removing the orphans increase the precision score without decreasing the recall value.

12.1.3 Region of interest

Unlike other applications you can find on the market we don't define a region of interest (ROI), the entire frame is processed to look for MRZ lines. Setting a ROI could decrease the false-positives and improve the precision score without decreasing the recall value. The deep learning model used for the detection is very accurate (high precision and high recall) and should not output false-positives if you're using the default recommended values.

12.2 Recognizer

This section explains how to increase the accuracy for the recognizer layer.

12.2.1 Data validation

In the previous sections we explained how to use the check digits to validate the most important fields (e.g. document number, expiry date, date of birth...) from the MRZ data.

When you're processing images from a video stream you can use the data validation process to make sure the result from the SDK is correct. If the validation fails for one frame then, just drop it and process the next one until you get something valid. You can also use the recognition scores/confidences values to ensure that other fields (e.g. holder's name, optional data...) without check digits are also valid.

12.2.2 Interpolation

As explained in the configuration section, the interpolation operation accepts 3 values (JSON strings): `bicubic`, `bilinear`, and `nearest`. The default value is `bilinear`.

The interpolation operations are used when pixels are scaled, deskewed or deslanted. `bicubic` offers the best quality but is slow as there is no SIMD or GPU acceleration yet. `bilinear` and `nearest` interpolations are multithreaded and SIMD accelerated. For most scenarios `bilinear` interpolation is good enough to provide high accuracy/precision results while the code still runs very fast.

Change the interpolation value to **bicubic** if you're having low recognition score.

12.2.3 Score threshold

The SDK returns a score/confidence value for each MRZ line. These scores are in percentage and within [0.0, 100.0]. When the score is `>=90%` then, you're sure that every character in the MRZ line is correct. A score `>=70%` means almost every character in the MRZ line is correct.

We recommend using a threshold value equal to `70%` and using the validation process to make sure everything is correct.

13 Improving the speed

Our implementation is massively multithreaded, SIMD and GPGPU accelerated but on some low-end devices it could be slow.

This section explains how to improve the speed (frame rate).

13.1 GPU / CPU workload balacing

A device contains a CPU and a GPU. Both can be used for math operations. You can use `gpgpu_workload_balancing_enabled` configuration entry to allows using both units. On some devices the CPU is faster and on other it's slower. When the application starts, the work (math operations to perform) is equally divided: 50% for the CPU and 50% for the GPU. Our code contains a profiler to determine which unit is faster and how fast (percentage) it is. The profiler will change how the work is divided based on the time each unit takes to complete. This is why this configuration entry is named "workload balancing".

On x86 there is a known issue and we only recommend enabling this option on ARM devices. **On ARM device this could speedup the detection by up to 100%.**

13.2 Segmenter

As explained in the configuration section, the segmenter accurracy accepts 5 values (JSON strings): `veryhigh`, `high`, `medim`, `low`, and `verylow`. The default value is `high`.

If you're using a low-end mobile device then, consider using `medium` value.

13.3 Interpolation

As explained in the configuration section the interpolation operation accepts 3 values (JSON strings): `bicubic`, `bilinear`, and `nearest`. The default value is `bilinear`.

The interpolation operations are used when pixels are scaled, deskewed or deslanted. `bicubic` offers the best quality but is slow as there is no SIMD or GPU acceleration yet. `bilinear` and `nearest` interpolations are multithreaded and SIMD accelerated. For most scenarios `bilinear` interpolation is good enough to provide high accuracy/precision results while the code still runs very fast.

Make sure the interpolation value is `bilinear` (recommended) or `nearest` (not recommended).

13.4 Region of interest

Unlike the other applications you can find on the market we don't define a region of interest (ROI), the entire frame is processed to look for MRZ lines. The default resolution used in our sample application is HD (720p). If you're using a low end mobile device then, consider setting a region of interest instead of downscaling the resolution.

13.5 Device orientation

When the device is on portrait mode then, the image is rotated 90 or 270 degree (or any modulo 90

degree). On landscape mode it's rotated 0 or 180 degree (or any modulo 180 degree). On some devices the image could also be horizontally/vertically mirrored in addition to being rotated.

Our deep leaning model can natively handle rotations up to 45 degree but not 90, 180 or 270. There is a pre-processing operation to rotate the image back to 0 degree and remove the mirroring effect but such operation could be time consuming on some mobile devices. We recommend using the device on landscape mode to avoid the pre-processing operation.

13.6 Memory alignment

Make sure to provide memory aligned data to the SDK. On ARM the preferred alignment is 16-byte (NEON) while on x86 it's 32-byte (AVX). If the input data is an image and the width isn't aligned to the preferred alignment size, then it should be strided. Please check the memory management section for more information.

13.7 Planar formats

Both the detector and recognizer expect a grayscale image as input but most likely your camera doesn't support such format. Your camera will probably output YUV frames. Converting YUV frames to grayscale is a nop (very fast), we just need to map the Y plane.

13.8 Reducing camera frame rate

The CPU is a shared resource and all background tasks are fighting each other for their share of the resources. Requesting the camera to provide high resolution images at high frame rate means it'll take a big share. It's useless to have any frame rate above 25fps. What is very important is the frame resolution. Higher the resolution is better the detection and recognition qualities will be. Try to use very high (2K if possible) resolution but low frame rate.

14 Benchmark

It's easy to assert that our implementation is fast without backing our claim with numbers and source code freely available to everyone to check.

Rules:

- We're running the processing function within a loop for #100 times.
- The **positive rate** defines the percentage of images with at least #2 MRZ lines. For example, 20% positives means we will have #80 **negative** images (no MRZ lines) and #20 positives (with MRZ lines) out of the #100 total images. This percentage is important as it allows timing both the detector and recognizer.
- We're using `high` accuracy for the segmenter and `bilinear` interpolation.
- All positive images contain a least #2 MRZ lines.
- The initialization is done outside the loop.

The concept of **negative** and **positive** images is very important because in most use cases you'll:

1. Start the application
2. Move the application to the MRZ zone to recognize the data

You only need a single "good frame" to recognize the MRZ lines. But, between step #1 and step #2 the application has probably processed more than #200 frames (40fps * 5sec). So, in such scenario the application have to process #201 frames before reaching the "good frame": #200 negatives and #1 positive. If processing negative frame is very slow then, the application won't be able to catch this "good frame" at the right time. A slow application will do one of these strategies:

1. **Drop frames to keep the impression that the application is running at realtime:** In such scenario the positive frames will most likely be dropped (probability = 1/201 = **0.49%**) which means reporting the time when this single "good frame" is caught.
2. **Enqueue the frames and process them at the application speed:** This is the worse solution because you could run out of memory and when the application is running slowly then, you can spend several minutes before reaching this single "good frame".

A fast application will run at 40fps and catch this "good frame" as soon as it's presented for processing. This offers a nice user experience.

	0% positives	20% positives	50% positives	70% positives	100% positives
Core i7-4790K (Windows 8)	877 millis **114 fps**	1975 millis 50.61 fps	3736 millis 26.76 fps	4901 millis 20.40 fps	6526 millis 15.32 fps
iPhone7 (iOS 13)	1990 millis **50.23 fps**	4325 millis 23.11 fps	7982 millis 12.52 fps	10595 millis 9.43 fps	14201 millis 7.04 fps
Galaxy S10+ (Android 10)	2825 millis **35.39 fps**	7575 millis 13.20 fps	12960 millis 7.71 fps	17636 millis 5.67 fps	21069 millis 4.74 fps
Raspberry Pi 4 (Raspbian OS)	4335 millis **23.06 fps**	13555 millis 7.37 fps	27878 millis 3.58 fps	37399 millis 2.67 fps	47797 millis 2.09 fps

More information on how to build and use the application could be found at https://github.com/DoubangoTelecom/ultimateMRZ-SDK/blob/master/samples/c++/benchmark/README.md. For Android and iOS the Benchmark application is in samples/android/benchmark and samples/ios/benchmark folders.

Please note that even if Raspberry Pi 4 have a 64-bit CPU Raspbian OS uses a 32-bit kernel which means we're loosing many SIMD optimizations.

15 Best JSON config

Here is the best config we recommend:

- **YUV420P image format as input**: This format is easy to convert to grayscale (format expected by the recognizer and detector). You should prefer YUV420P instead of YUV420SP (NV12 or NV21) as the later is semi-planar which means the UV plane is interleaved. De-interleaving the UV plane take some extra time.
- **720p image size**: Higher the image size is better the quality will be for the detection and recognition parts. 720p is a good trade-off between quality and resource consumption. Higher image sizes will give your camera a hard time which means more CPU and memory usage.
- **2 for minimum number of lines**: All standard MRZ formats requires at least 2 lines. Requiring a minimum of 2 lines improves the precision score (more orphans will be removed) and speedup the processing (no recognition on orphans). JSON config: `"min_num_lines": 2`
- **0% for minimum score**: We recommend using a 0% threshold to make sure no recognition result will be rejected. Use data validation to decide whether to keep or reject a result. Another sanity check in addition to data validation is to make sure all MRZ lines have the same number of characters. JSON config: `"min_score": 0.0`

The configuration should look like this:

```
{
    "debug_level": "warn",
    "min_score": 0.0,
    "min_num_lines": 2
}
```

16 Debugging the SDK

The SDK looks like a black box and it may look that it's hard to understand what may be the issue if it fails to recognize an image.

Here are some good practices to help you:

1. Set the debug level to "verbose" and filter the logs with the keyword "doubango". JSON config: `"debug_level": "verbose"`
2. Maybe the input image has the wrong size or format or we're messing with it. To check how the input image looks like just before being forward to the neural networks enable dumping and set a path to the dump folder. JSON config: `"debug_write_input_image_enabled": true, "debug_internal_data_path": "<path to dump folder>"`. Check the sample applications to see how to generate a valid dump folder. The image will be saved on the device as "`ultimateMRZ-input.png`" and to pull it from the device to your desktop use adb tool like this: `adb pull <path to dump folder>/ultimateMRZ-input.png`
3. The computer vision part is open source and you can match the lines on the logs to https://github.com/DoubangoTelecom/compv

17 Frequently Asked Questions (FAQ)

17.1 Why the benchmark application is faster than VideoRecognizer?

The VideoRecognizer application have many background threads to: read from the camera, draw the preview, draw the recognitions, render the UI elements... The CPU is a shared resource and all these background threads are fighting against each other for their share of the resources .

18 Known issues

There is no known issue.

Please use the issue tracker to open new issue: https://github.com/DoubangoTelecom/ultimateMRZ-SDK/issues