Microfan Document Generator Manual

# Purpose

The purpose of this project is to automate the device manuals creation for Microfan. As all the info needed for the various device manuals is located in databases, another application is generating a JSON file based on those info. The manual will be created according to this file.

# Structure of the JSON file

A structure of the JSON file can be found in [Appendix A](#_Appendix_A). The values inside brackets [] indicate a mock value of the parameter. For the full file go to: (solution folder) -> Sources -> properJsonFile.json.

# Structure of the project

The solution consists of 4 projects: Document.Generator.App, Microfan.AutoDocumentation.dll, CotentBlock.Models and TocAdder.

* **Document.Generator.App**: is responsible for calling the Microfan.AutoDocumentation.dll and printing the return of the dll.
* **TocAdder**: Ready project from Eric White (the developer of Open XML) to update a table of contents automatically when the user opens a MS Word file.
* **Document.Generator.Dll**: Printing the C# objects to a Word file can be found here. This projects also makes use of the TocAdder project.
  + Where it consist of MarkDown(PatternMatcher)for catches all the text and convert to Bold, Italics and Underline.
  + And Functionality for parsing the JSON file to C# objects.
* **ContentBlock Models :** Its a Shared Library where is the core of the project. Where it consist of all the **Content Block** (image, Function, text, Path, remark)
  + It Consist Of ContentBlock\_Pars convert the .net and Json type.
  + It Deserialized the Json to a Specified .Net Code

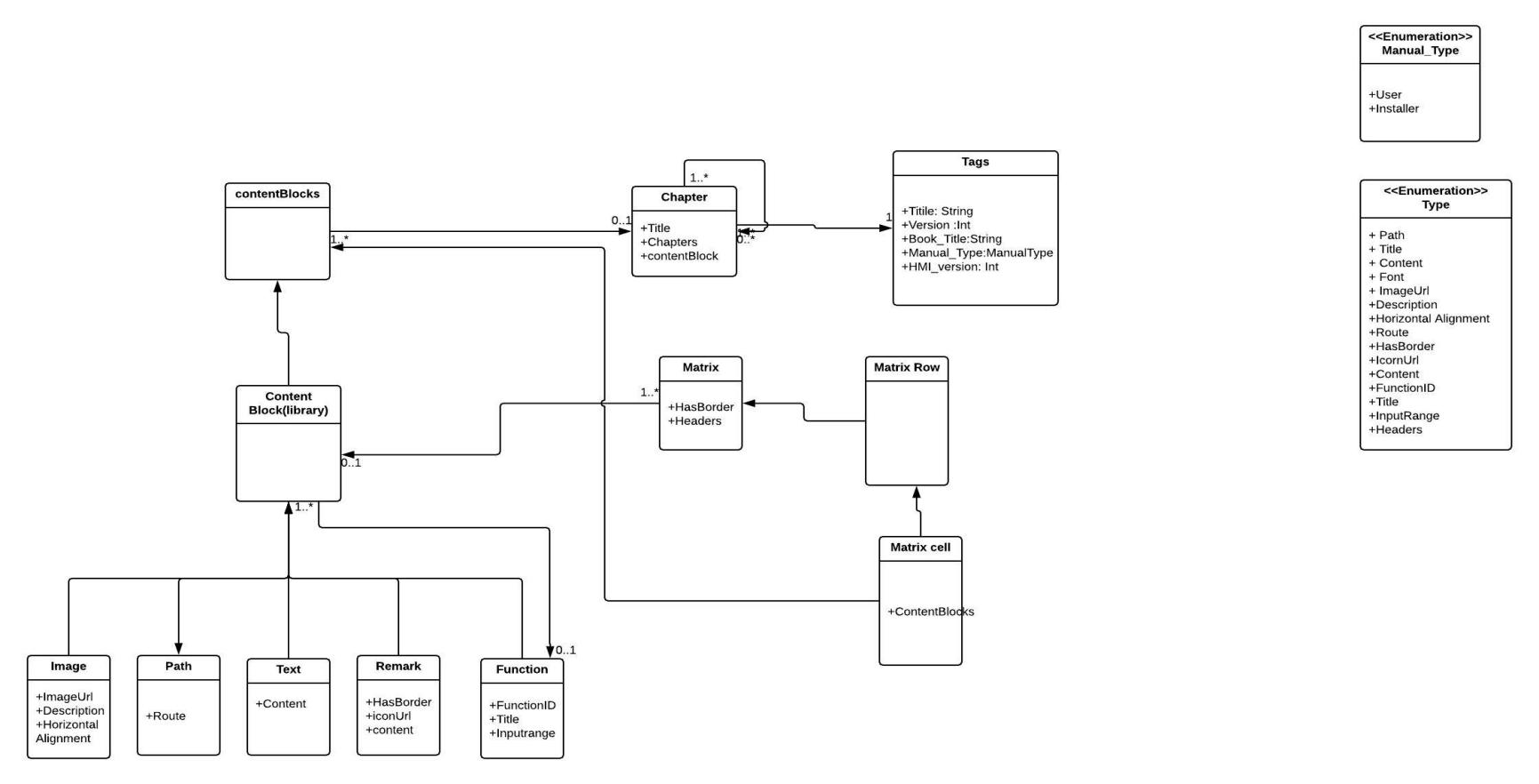
## Folder structure

Inside Document.Generator solution folder the following important folders are located:

* **ConsoleApp1**: the project folder of the Document.Generator.App.
* **MarkDownToOpenXMl:** Its a open source library where it convert the normal text into Markdown Objects.(it completely convert the text.txt file to word doc )
  + Only PatternMatcher, Range class are used in Document\_generator for creating the MarkDown Components.
* **TocAdder**: the project folder of the TocAdder.
* **MicrofanAutoDocumentation**: the project folder of Microfan.AutoDocumentation.Dll
  + **Final** **Documents**: this folder contains the output of the application, namely the device’s manual as a MS Word file.
  + **Images**: all the images needed for the document are placed there. The image elements in the json file contain only the name of the image in the ‘Source’. The rest of the path is supplied but the application in the form of a const string. This folder includes images and also static pages as images.
  + **Sources**: the json file to be parsed is located here.
  + **Templates**: the MS Word templates that will be used for building the word document are located here.
* **BulletList\_reference\_Links:** links and references for creating bullet list(Readme File)

## Class diagram

The class diagram of **Document.Generator.Dll**.



* **Tags**: It holds the basic info of the document such as title, version, language etc.
* **Chapter**: it consists of a chapter title and a list of content blocks.
* **ContentBlock**: holds a list of ContentBlock that together create a block of content. The list may contain 1 or more ContentBlock.

## ContentBlock types

The ContentBlock holds which holds the sub-type of the element. The sub-types are: , Path, Matrix, Image, Text, Function and Remark.

### Path

This is an element indicating a path that a user must follow to reach specific functionality. Ex: system->settings->date->change date. It holds only a string array “Route” which contains the various intermediate steps. The steps inside the array are placed in the **proper order**.

### Matrix(To Do !!)

.

### Image

This is an element denoting an image in the word file. It holds a string “Source” to indicate the source path of the image. [**Important**: in the mock Json file, the “source” contains only the name of the file. The rest of the path is combined in WordElementWriter.AppendImage(), for the full path of the file, using a cont string for the images folder location.] It also holds a string “Description” to save the description of the image. Moreover, it holds an enum indicating the “horizontal alignment” of the image, as well as 2 integer values to save the width and the length of the image.

### Text

This is the simplest type of Element. It contains a string variable named “Content” which holds the actual contents of this text instance. Moreover, it captures the text and looks for pattern match (\*\*Bold\*\*, `italics`\_underline\_) inside the text and it will print according to it.

### Function

A function in the scope of this project is a piece of information that describes a specific function in the device. It holds the function’s ID (integer), its title (string), the input range of that function (also a string) and a contentBlock (list of elements, mostly text and some images). An example from the original manual follows (font is changed to clearly indicate what belongs to the function and what doesn’t):

**Slave ID for testing**

*Input range: 0-499*

*Function ID : 4541144*

Here, enter the slave’s ID. IDs 1 – 249 is for slaves via RS485 communication, IDs 250 – 499 is for slaves via I2C (in this case, the slave PCB is connected to the ARGOS P1 main PCB via a flat cable).

### Remark

Remarks are pieces of text that contain important info and stand out of the normal text flow. The Remark Element holds a Boolean value to indicate if it has borders, a string value named “Content” to indicate the text inside the remark and a string “IconUrl” to indicate the path to the image to be placed next to the content. The following is an example of remark from the installer manual:



**Important!** Be careful with manually setting/resetting relay positions of, for example, feed systems. After ending the test mode, they can be in the wrong position.

# JSON Parser

## ParserJson.cs

Inside JsonParser folder in the project, the files for parsing the JSON to C# objects are located. The ParserJson.cs is responsible for getting the attributes of the document (such as title, version etc), creating a Document.cs instance, hold it to memory and also iterating through the various elements inside the chapters.

## ContentBlock\_Pazer.cs

ContentBlock\_pazer is in the ContentBlock.Model (Shared Library)

This class is responsible for the actual parsing of each element. Contains a public dynamic Object JsonConvert.DeserializeObject which accepts a JObject, defines its Type and sends it to the corresponding method to be parsed properly (CreateRemark(), CreateImage() etc). Each of these methods returns the corresponding C# Element.

# Word template

It is important to use the template provided in the project, to print the elements to a word file. It contains custom headings, styles and macros that make it as close as possible to the original Microfan’s manuals. The headings are numbered and preserve the numbering in the indent. The macro prevents a pop-up message for updating the table of contents when the file is opened.

## Word writer

Inside MicrofanAutoDocumentation project there is a WordWriter folder. The classes responsible for writing to the word file can be found there.

### WriteToFile.cs

This class is the entry point for writing to the word file. It contains only the ToFile(Document) method that uses dependency injection to get the C# Document object. The constant strings of the template path and the output file path are placed here. It is also responsible for iterating through each chapter’s elements and send them to WordElementWriter.cs for writing them to the file.

### WordElementWriter.cs

This class is responsible for appending every element to the document's body.This class follows the implementation of the ContentBlock\_Pazer.cs in the sense that it contains a public method that accepts an element, and then passes it (according to its ContentBlockType) to the corresponding private method for appending it to the file.Where each ContentBlock is created.

### PattenMatcher

This class which is used form an open source library where it convert the normal text into Markdown Objects. The main Objective of the Pattern class is to create a pattern for the markdown Items and checks if there is any match found inside the “Content” in the TextCotentBlock.

# NuGet packages

The NuGet packages used in this solution are the following:

* **Newtonsoft.Json v11.0.2:** this package is used for parsing the JSON file.
* **DocumentFormat.OpenXml v2.8.1:** this package is used for printing the C# objects to the word file.
* **SixLabors.ImageSharp v1.0.0-beta0005:** this package was used for getting the metadata of image files to determine width, height and possible resize if needed.

## Open XML SDK Productivity tool

This tool can be downloaded [here](https://www.microsoft.com/en-us/download/details.aspx?id=30425). It is very useful tool as it opens the xml files of the docx file.

## Bugs

1. Bullet List is implemented but it couldn't catch pattern properly.

# To Do !

Implementing the matrix

# Appendix A

Mock JSON file of the project.

{

"Tags": {

"title": "ARGOS M1/M2/M3",

"version": "V20180718",

"BOOK\_TITLE": "ARGOS P1",

"type": "User manual",

"language": "EN",

"HMI\_Version": "3.09"

},

"documentParts": [

{

"chapters": [

{

"title": " First Chapter 1",

"contentBlocks": [

{

"type": "Text",

"content": "\_Regularly clean the screen with a soft clean and damp cloth\_ "

},

{

"type": "remark",

"hasBorder": "true",

"iconUrl": "remark.jpg",

"content": "Regularly clean the screen with a soft clean and damp cloth"

},

{

"type": "image",

"ImageUrl": "default.jpg",

"content": "ARGOS M1 screen",

"horizontalAlignment": "Center"

},

{

"type": "function",

"functionid": 10020,

"title": "Minimum/Maximum measurements reset time ",

"inputRange": "00:00-23.59 hh:mm",

"contentBlocks": [

{

"type": "path",

"route": "system.settings.functions.climate.moisture"

},

{

"type": "Text",

"content": "\*\*First Chapter Content\*\*"

}

]

}

]

}

]

}

]

}