Development done on FreeType and key points

* **Overview of font rendering**

1. Whatever that is showed on screen either in websites or any UI of our operating system is converted from a design into pixels. OS is responsible for this translation of design to pixels
2. Font rendering is different for different systems and font formats.
3. Hinting is one importing terminology which font files provide to rasterizer for applying specific things for better display.
4. When a text is displayed on screen, this very precise, ideal shape needs to be expressed with a more or less coarse grid of pixels (rasterization).
5. In grayscale rendering, a pixel that is on the border of the original shape becomes gray, its brightness depending on how much it is covered by the ideal black shape. As a result, the contour appears much smoother, and design details are represented. It is called antialiasing.

* **Windows Font Driver Module Focus**

Every Font Driver in FreeType is providing some format specific services. **Windows Font** is one of the Driver module which is built simple. It supports windows fonts of formats .fnt and .fon. We targeted this font driver because its comparatively less complex than the other Driver modules. After completing the process which I will explain later we found that this is a very old font driver and it is not used now days i.e. the file format it supports which are mentioned above are not used today. So, when we reached the testing phase where we were supposed to test it by providing a file to our client application to check if the respective font is printed on screen or not we failed. The reason of failure was after a long search we found that these files are no more available and are not supported on Linux especially. So, we had to move our target driver to proceed in our FreeType testing and project work. This time we have targeted the **Type1 Driver module**.

* **Type1 Font Driver Module Focus**

As mentioned in previous paragraph windows font is not widely used today so all the changes we did were of no use because we couldn’t test them. So, to test the configuration of new font driver module in FreeType and checking the flow and other things we have shifted to another default driver module called Type1. This font Driver supports Postscript font files of extensions .pfa and .pfb. We studied the original Type1 (not core functionality) but the flow and tried to figure out what we can extract. I will mention some key points learned below that how a new driver can be configured. We face challenges configuring it because this driver is different from simple windows font driver, because it is providing multiple services to FreeType other modules and it is also dependent on other modules.

* **Key elements in Type1 source**

Every Font Driver consists of some same files which can be found in all driver modules with their own functionality like

1. **Jamfile:** which is platform independent build system. It can be used as an alternate of make file.
2. **Module.mk:** This is a module definition make file. In this file, we mention the name of our module its class name and print the text and format this Module will represent. What we learnt is that no matter your module is empty or not this make file is run and it auto generates the Ftmodule.h in the objs directory when FreeType is compiled.
3. **Rules.mk:** This is the make file in which we define the rules for this module. Like source directory of this module, the main C file of this module and core header files like errors etc.
4. **t1afm:** Afm stands for Adobe Font Metrics (contains Kerning etc.). This file extension is supported by Type1 like it supports the ASCII and Binary files .PFA and .PFB.
5. **t1driver:** This is the high-level driver interface. In this file the modules driver class is declared and defined. The definition of driver class is written in t1driver.c.
6. **t1errors:** Every font Driver has its own error files where the prefix and Base errors codes are defined.
7. **t1gload:** This file is responsible for loading glyph. Every font driver has its own Glyph loaded.
8. **t1load:** This file is the Type 1 font data loader. Not sure but may be these files is responsible for reading all the data from file very fast.
9. **t1objs:** This file is responsible for all the object declaration and definition of modules. The main thing about it is that it defines some key objects used in driver module specific class definition.
10. **t1parse:** This file is the Type1 parser. No details known.
11. **t1tokens:** The matching rules are defined in the file `t1tokens.h' using several macros calls PARSE\_XXX. This file helps in parsing and is used in load data file. (not sure).
12. **type1:** This file just loads the Type 1 driver components.

* **Steps of configuring new empty driver module in FreeType**

To register an empty driver module without any errors and warnings following steps are done as mentioned below.

1. Create a new module.mk file and define the driver name, and class to be opened for this driver and some text containing extension to be shown on compile time.
2. Write the rules.mk file for this driver and define the new driver module path the key C file and header files along with all important component files related to this module.
3. Declare the driver module class in driver.h and just include some files like driver’s error.h and driver.h in driver.c file.
4. Include all components in main driver c file in our case it is type1\_test.c file and also add t1tokens.h as is in directory.
5. Add the new driver in modules.cfg and compile FreeType with make command you will see no errors.
6. Check the Objs directory and you will see ftmodule.h file autogenerated with our new driver module class and its .o and .lo files.

* **Steps of configuring replica driver module in FreeType**

For the replica module, all the .h and .c files of afm, load, gload, parse, obj, driver etc. are loaded and driver module class definition is defined in driver.c file. Everything is compiled without any errors and warnings. We may change all the module specific variables in our new driver module.

Till now we have not added any file or modified any code or any source files of include directory or any other directory except the src directory for our new directory module. The main part of our new module registration and configuration is the new driver module class declaration and definition in the drivers .c and .h files respectively. Because when client applications try to access the modules there is an api of FreeType called modapi which is responsible for module management in FreeType. To have our new module available at run time we have to properly define its name version etc.

* **Client application to access the modules and some key points of FreeType**

Client applications can add, remove, upgrade modules at run time. Module management API has its own functions for client applications like add module, get module etc. To make sure our new module is properly registered we made one client application which actually takes one name of a module of FreeType and tells whether it really exists in short it is registered or not. It worked fine in our case and our new module which is replica of type1 driver is configured in FreeType. Some key points found while all these things.

1. The library object is the *parent* of all other objects in FreeType 2. You need to create a new library instance before doing anything else with the library. Similarly, destroying it will automatically destroy all its children (i.e. faces and modules).
2. Another alternative is to create a fresh new library instance by calling the function FT\_New\_Library (), defined in the <FreeType/ftmodule.h> public header file. This function will however return an "empty" library instance with no module registered in it. You can "install" modules in the instance by calling FT\_Add\_Module () manually.
3. Calling FT\_Init\_FreeType () is a lot more convenient, because this function basically registers a set of default modules into each new library instance by calling FT\_Add\_Default\_Modules. This function actually goes to ftmodule.h in config file where all module names and classes are mentioned.

* **Steps of finding that how a font format is recognized by FT and passed to respective driver that deals with it.**

1. Every font driver has a parser which basically deals with implementation of tables needed by loader, opening format specific files like in Type1 case opening .pfa and .pfb files to extract their top level and private dictionaries.

* **Email and response for the above issue**

> My query is that i am unable to find the place in FreeType which  
> decides that which driver to call.  I mean when a .ttf file  
> is inputted than where FreeType decides that Truetype driver has to  
> deal with this for rasterizing.  
  
It's actually very simple.  The main loop for iterating over the font  
drivers is in file `src/base/ftobjs.c', function  
`ft\_open\_face\_internal' (lines 2250ff in the current version of the  
git repository).  It eventually calls  
  
  clazz->init\_face  (line 1210)  
  
which points to a module's font initialization routine.  If this  
routine succeeds – this is, the module was able to successfully parse  
the font header –, this font driver is taken, otherwise the next one  
is tried.  
  
The order of font modules is given in file `modules.cfg'; FreeType's  
`Makefile' constructs the header file `ftmodule.h' based on this data.  
  
> According to my research there is a tool named Ftrandom in tools  
> directory but i am not sure if it is used for this purpose or the  
> parse file in each driver main source directory.  
  
The `ftrandom' tool is used to create fuzzy (i.e., buggy or invalid)  
input to test whether the library can properly reject it.  It is not  
part of the library itself.

* **Font config issue for windows font and FTVIEW program**

>> Simply download some windows bitmap files!  This is the first hit  
>> on google for searching `.fon download':  
>>  
>>   <http://www.2200freefonts.com/b-fon-b-fonts.php>  
>  
> Actually i tried these fonts before but these are windows  
> executable's and i am unable to install them in Linux for my  
> testing.  
  
What are you talking about?  I tried the first font on this page,  
  
  <http://www.2200freefonts.com/new/911porscha_bitmap0.zip>  
  
and it is a plain zip file!  You certainly can extract ZIP files on  
your GNU/Linux box, right?  
  
Even if they were self-extracting files there \*are\* Unix tools that  
can handle them.  For example, there is the glorious `file' command  
that tells you the exact type of the file; it usually can recognize  
self-extracting files, telling you which program you have to use.  
Additionally, you can always use `wine' to execute the EXE file.  
  
BTW, FreeType displays this font just fine at 13px.

> Actually i extracted the same way and copied in my /usr/share/fonts  
> and even in ~/.fonts, but hey don’t appear in (fc-list) after i  
> manually rebuild the font cache (fc-cache -f -v), so i can  
> immediately use them everywhere.  
  
It is very likely that fontconfig doesn't support Windows bitmap  
fonts – I don't know.  However, `ftview' works just fine.

**Notes on reading type 1 font files**

* Type 1 font files are postscript files.
* They come with two extensions .pfa and .pfb
* The difference between these two is that .pfb extension has binary section stored as raw bytes whereas .pfa stores that in hex characters.
* A Type 1 font file has three segments. In order:
  + (a) an initial ascii part (printed ascii characters)
  + (b) the middle part (binary format differs in .pfa and .pfb)
  + (c) a final ascii part (printed ascii characters)

**Dynamic library linking problem and solution**

> I am working in linux platform. I am changing some code in Freetype and  
> looks like they are not applied or linked. Even i tried to comment all the  
> code of function FT\_Init\_FreeType and called it with my application and  
> still it worked. I compiled Freetype without errors and then installed it  
> via sudo make install that means it should always invoke these freetype  
> files, whenever i call it with my application.  
  
I think this problem is not related to FreeType at all; it is rather a  
generic problem with dynamic libraries, and you should make yourself  
better acquainted with this topic.  
  
`make install' by default installs into `/usr/local'.  I guess your  
system is set up so that the dynamic linker looks into `/usr' first.  
Since virtually all GNU/Linux platforms come with `libfreetype' in  
`/usr', this version will be linked to.  You can use the `ldd' program  
to check which dynamic libraries your application is using.  
  
Possible solutions.  
  
  . Ensure that `/usr/local' gets found first by the dynamic linker,  
    `ld.so'.  Note that this can still be problematic since other  
    system DLLs might still depend on libfreetype from `/usr'.  
  
  . Use the LD\_PRELOAD environment variable.  This works 100%.  
    However, it is suited for testing only.  
  
  . Use static linking.  This works 100%, too.

**Implementation for Test Driver**

1. All Type 1 fonts must begin with the comment: %! This enables a file containing a PostScript program to be easily identified as such. It is important that every Type 1 font program—indeed, every PostScript language program—start with a “%!” comment; otherwise, it may not be given the appropriate handling in some operating system environments. Reference

http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/font/pdfs/T1\_SPEC.pdf

1. Changing a type1 font file header to a custom header file. As type1 font file always start with a text “%!PS-AdobeFont” according to FreeType author so changing this text can do the trick for testing out custom font.
2. Modifying the test Type1 driver for this custom font file in a way that it only works for this font file. I.E. whenever this font file is passed as input to FreeType, it automatically calls the new custom driver based on the custom header text.
3. Changing the parsing file of this test driver setting the header format according to font file header.

A test1 driver is configured in FreeType that only runs whenever a file with header text mtsummyfont123 is passed. For this a type1 font file was modified its header was changed to custom header with custom text as original type1 font driver recognize the header by reading header comments so these comments were changed in a way that type1 test driver recognizes this file and runs the parser and sub routines.

Type1 driver test module is also modified in parsing file where it checks headers. For testing 2 tests are made.

1. Providing a font file that is not a post script default type 1 font and not a font file that our test driver accepts I.e. a file that has custom headers. For this case, our test worked as message was alerted that this font is either not a type1 driver module file not a test driver font file.
2. Second test was when a custom header font file was inputted via client application the parsing area where it checks that is it the right file module to parse was passed and successfully font file data was showed.

**FOA module Development**

For FOA module development which will be the module of FreeType that will provide bridge functionality between the inner element of FreeType and outer elements. The test driver module created will be modified in a way that it will implement all the core modules of FOA like communication module, configuration module, management module etc. To go further in this step Type1 font driver needs to be understand more deeply so that we can understand how other modules are linked with this driver like from base layer to this module and how this module calls other helping modules for further help.

**How FreeType calls a driver module**

Whenever a font file of any format is inputted via any client application to FreeType, the file in base layer ftobjs.c calls the module listed in ftmodule.h in an order. Each module is called and checked whether that module could parse the font file if yes, this module is taken for further routines otherwise the next module mentioned in ftmodule.h is taken and so on this process is proceeded.

**Understanding flow of Type1 driver module source code**

**Ftobjs.c and Ftobjs.h**

This file contains the definition and implementation of all internal FreeType classes. The internal classes mean they cannot be accessed from outside FreeType like high level client functions. All the internal drivers and other modules can access these classes and functions etc. This file is in base layer, and this file picks a module from the list of all registered modules in ftmodule.h file. Every driver module picks the header of the font file passed and checks if it can parse this header if its parsed it is selected for further routines like picking all face related information from the font file. This face information is returned from driver module which parsed this file and then base layer returns the information to the application.

**Inner connection interface**

For inner connection interface, we need to interact the base layer with our FOA module internally. Or calling our FOA module interfaces from some other driver module. This will show that how internally we can interact with FOA module. After reading the article documentation and email <https://www.freetype.org/freetype2/docs/design/design-6.html> we realized that we simply cannot call each driver interfaces directly because a module can have its static attributes and even if a module is a driver module it can have standard functions like face\_init etc that can be accessed so for adding a new custom function we cannot simple use these two structiers i.e. FT\_Module and Ft\_Driver as quoted in documentation “There are two main kinds of interfaces: *module* interfaces, and *services*.

Module interfaces are defined for each module. For example, every font driver provides its own set of procedures for use in the base layer, which are registered in an FT\_Driver. This way, very different font drivers can be used in the same way in the base layer.

Services are cross-module interfaces. These provide functionality needed in several font drivers.

Services are created when code from one module needs to be used in another. Rather than include files from another module, a service is created instead. Now, the other module just needs to include the header defining the interface.”

EMAIL: > Is there a way we can call a custom function in a driver module from

> another driver module?

Only with hacks that lead to code like

#include "../foo/bar"

(note the ugly `..'). I don't consider this a clean solution.

> For example i have a custom function in Type1 driver module like

> T1\_Face\_Info\_Custom\_Func( ). Can i call this function directly from

> base layer or any other driver module like TrueType driver module.

See above.

> I tried to use FT\_Get\_Module\_interface() but it looks like it can

> only work if a service is made.

Yes. Recently, GSoC student Ewald Hew has added some documentation to

describe the creation of services, see

So, for our FOA module we will try both ways i.e.

1. Ugly way by adding header files of custom module
2. Making our custom function as service

* **Ugly way by adding header files of custom module**

A function is declared and defined in Type1\_test driver module. This function takes two arguments a face object and a module class pointer. In base layer file FT\_objs.c a custom header of Type1\_test driver is included so that custom interface can be called and accessed directly this is the ugly way of calling functions from one module to the other. In the success block which is only executed whenever a font driver module successfully parses and gets face related data out of data, this custom interface of Type1\_test driver module is called and face pointer along with module pointer which parsed this font file is passed to print the respective data on screen. The main reason of calling this method from base layer was to confirm the test that inner connection module can connect FreeType core elements like base layer with FOA module which in this case type1\_test driver module. As this approach of heading custom headers is not appropriate we will do the same thing by making a custom service which this type1\_test module provides and can be used easily by calling this service function. For this test 3 files are modified base layer ftobjs.c, Type1\_test driver modules t1objs.h and t1objsc.c.

* **Making our custom service layer as service**

Services are cross-module interfaces. These provide functionality needed in several font drivers. Services are created when code from one module needs to be used in another. Rather than include files from another module, a service is created instead. Now, the other module just needs to include the header defining the interface.

So, for implementing a service for our replica Type 1 test driver I created one new service that can be easily called from any module of FreeType I tested the service by calling it through base layer. This layer is basically the same function which was used in above method to print face object but instead of including the header of file where this service interface is implemented a function pointer of service is called from anywhere which easily calls this custom interface of FOA/replica type1\_test driver.

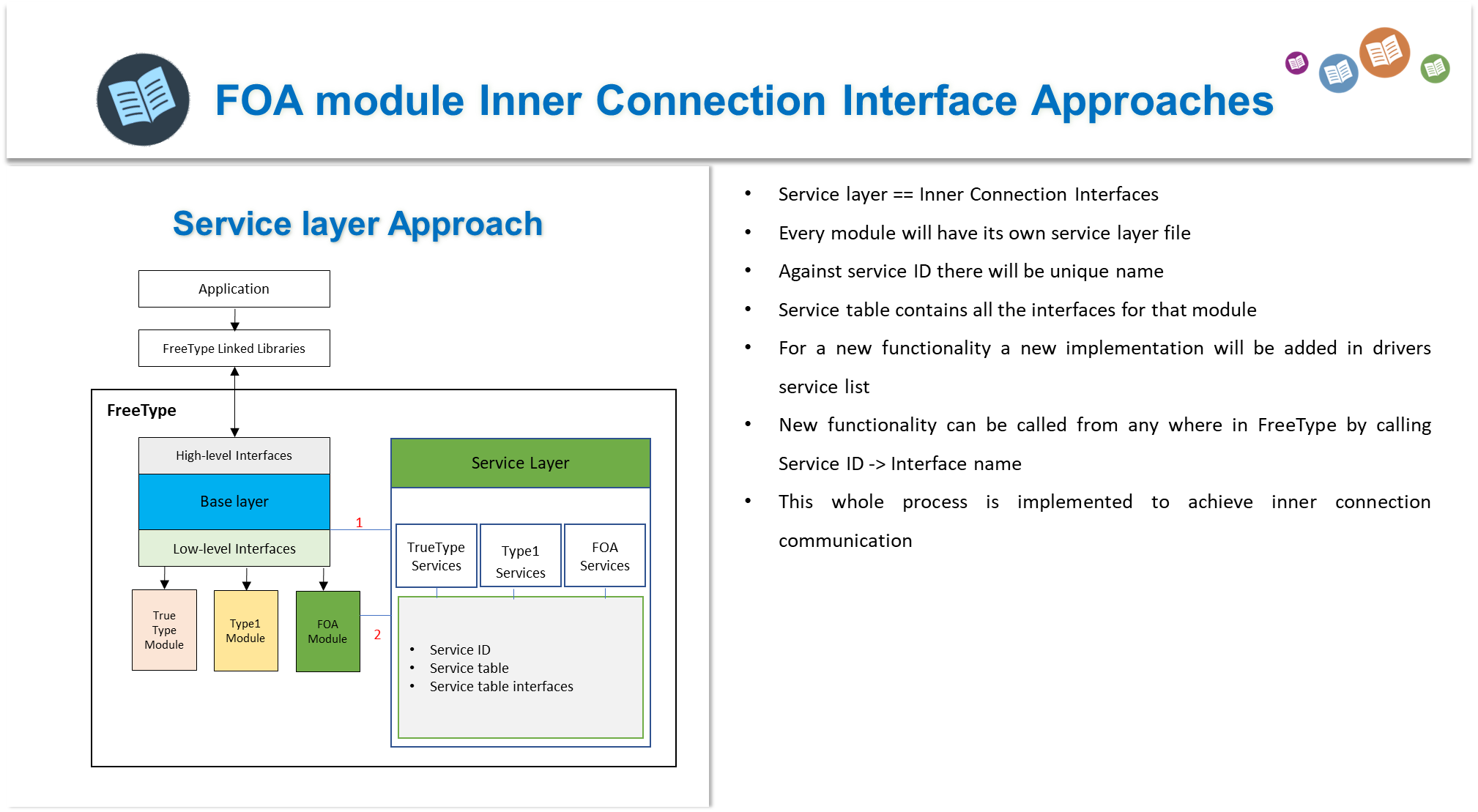
I followed the tutorial in the following link and studied code of window font service <https://www.freetype.org/freetype2/docs/design/design-6.html>

For the inner connection interface for FOA module we will be creating custom service list having all the interfaces that we need on base layer or in any driver module. These interfaces can start with a prefix inner\_connection so that we should differentiate that these interfaces are for inner connection interface.

**Main Idea of inner connection interface:**

The main idea we got after research is that modules can interact with each other share each other’s functions. For sharing each other functions for example if we need to a function func () of module A in module B, we need to make a service in module A that service will only declare the signature of func () than in module A we will define that service and its service table that will contain that func () and its implementation. In module B where we want to use that interface we will simple add the service header of that module A service .h file where that common func () is defined. And call that service by passing the service Id and service list name i.e. service table to FT\_FACE\_FIND\_GLOBAL\_SERVICE() function of FreeType that will return us all the interfaces available in that service list.

Inner connection interface will be a service layer inside FreeType which will allow core FreeType modules to call FOA module and its common functionality.

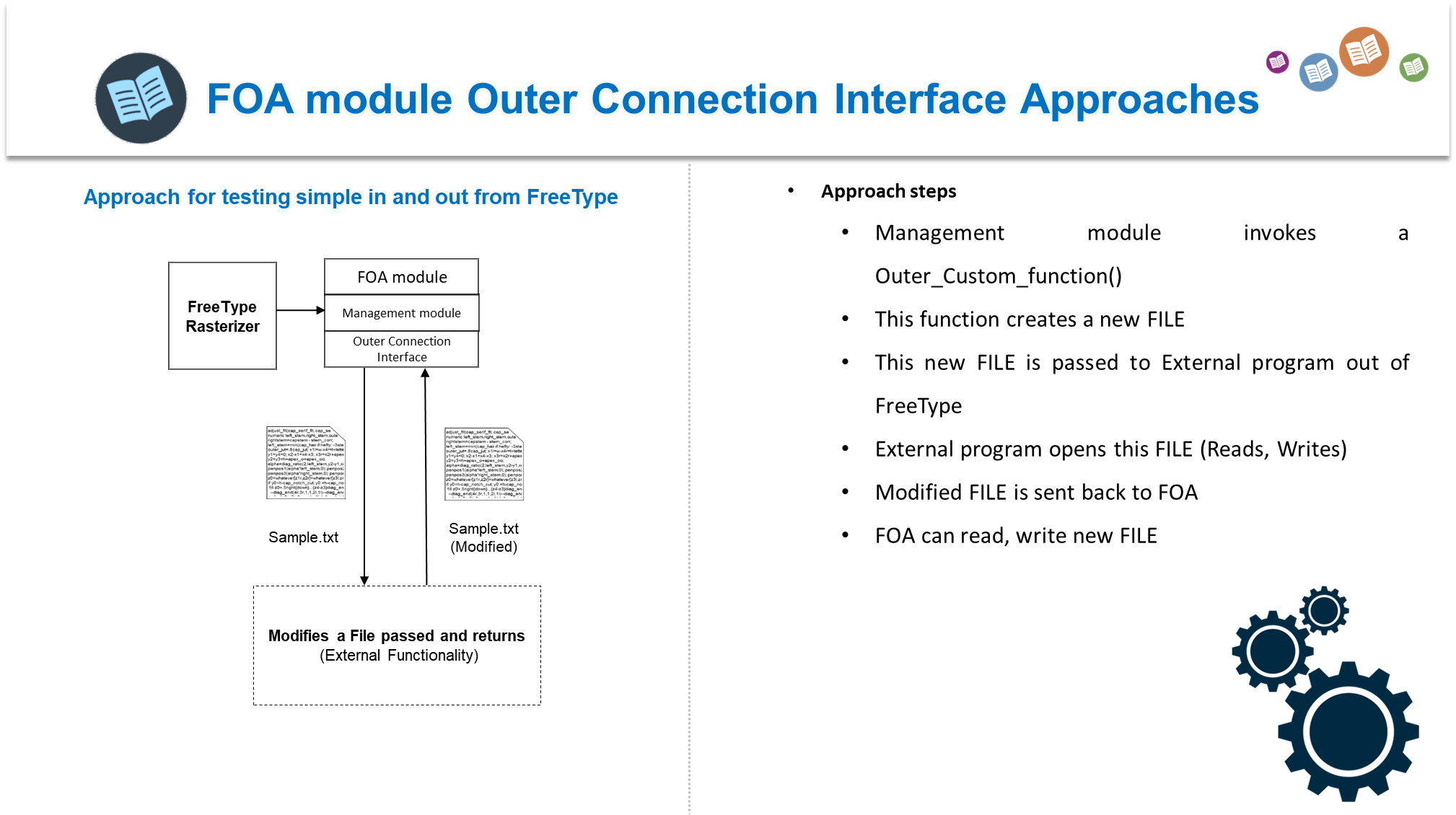


**Outer connection interface**

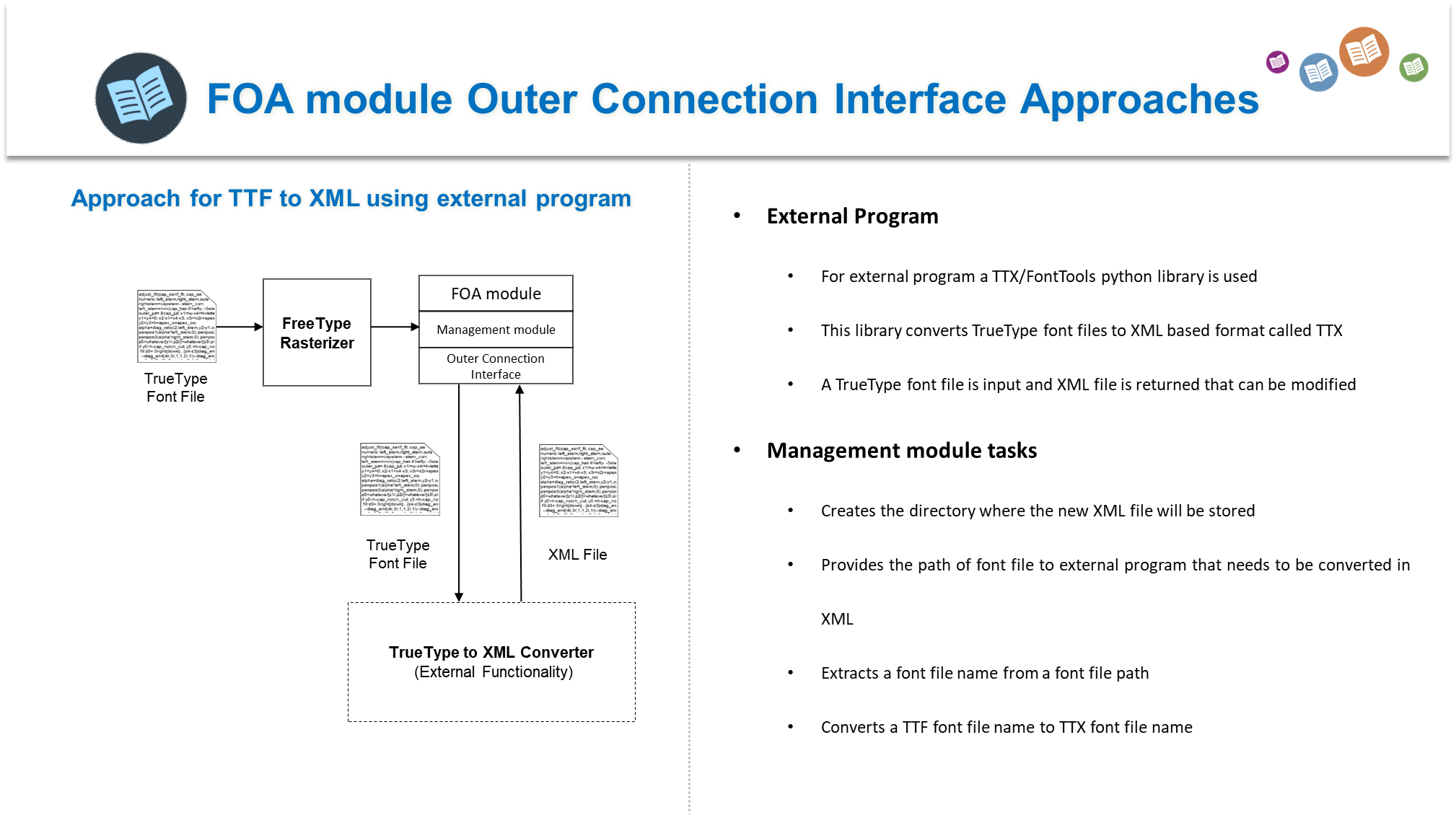
Outer connection interface is all about how FreeType can send a request to an external program. Going out from FreeType during its execution is called Outer connection interface. As we cannot directly include the header files of external program considering our project we will take the help of system calls for going to external C program.

The main idea for using Outer connection interface is that when an application will request a METAFONT to be displayed on screen, this request will come to FOA module. FOA module will need to check that if the font file input is METAFONT by checking its extension, then send it to external program called Mftrace. To go to external program from FreeType to Mftrace and getting the new Type1 font file is the responsibility of Outer connection interface.

For this whole scenario, we are implanting different approaches. First approach is about Creating a simple text file inside FOA module and sending this file to external program that modifies this text file. When it returns to FOA module, then FOA module can access the modified file and perform read, write operations.



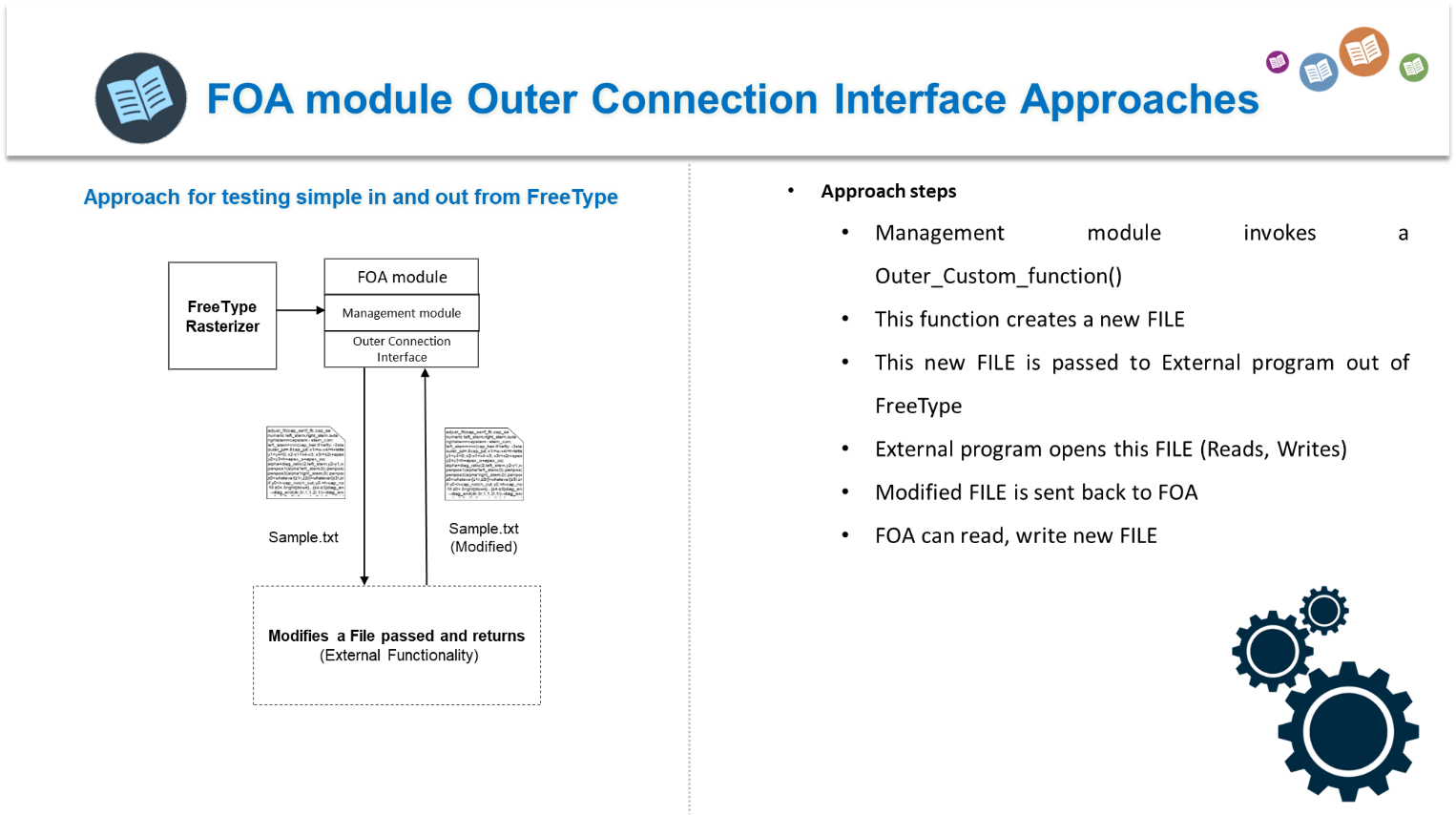
Another approach used for testing this Outer connection involved a much better external program that is called TTX/FontTools. This python program is responsible for creating a XML file from a True type font file. We sent a request from FOA module to this external program and received xml file in our own requested directory path with name as shown in figure below.



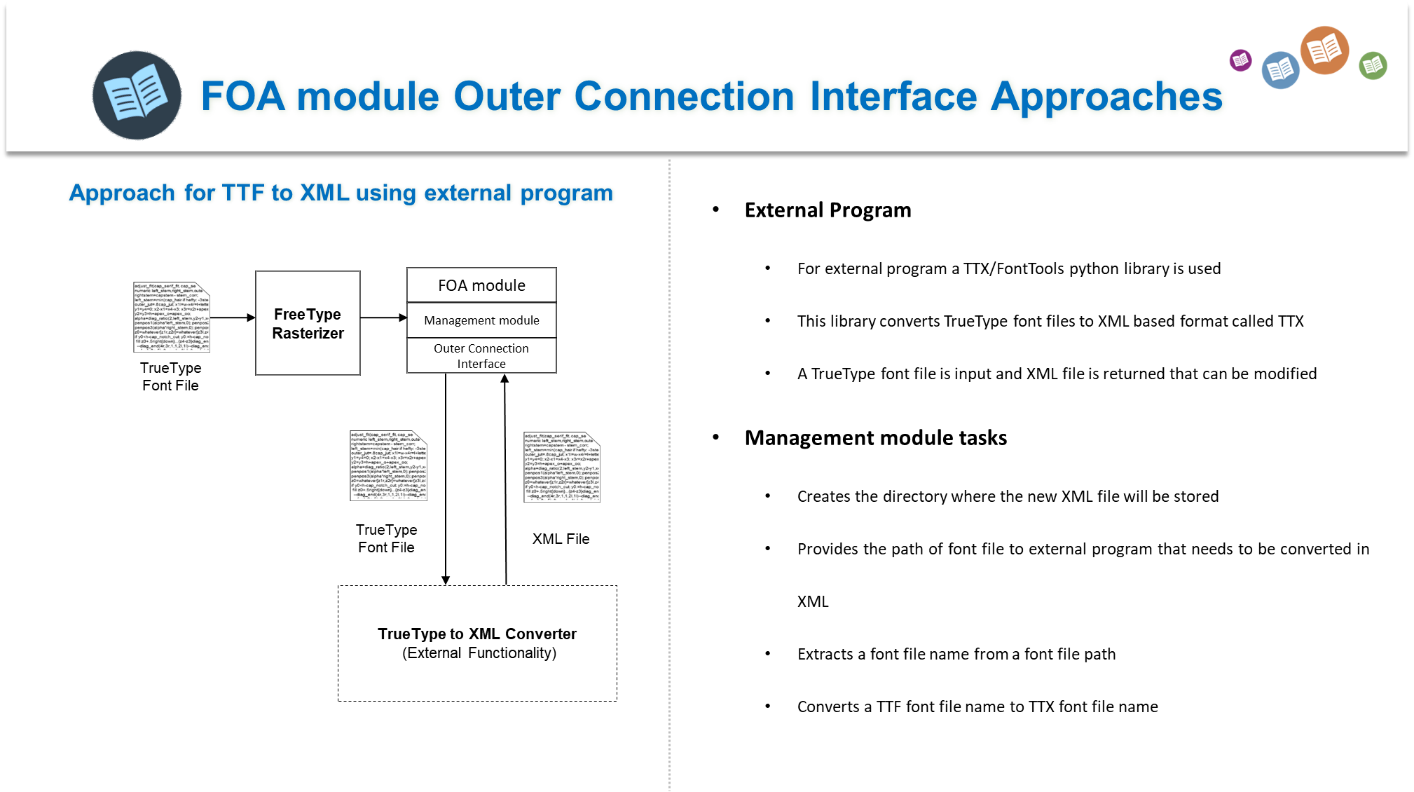
**Now our main approach is to use Mftrace as external program and try to convert a METAFONT into .PFB format and get font information.**

In my last presentation, I talked about some approaches I used for testing my outer connection interface and simple management module functionality.

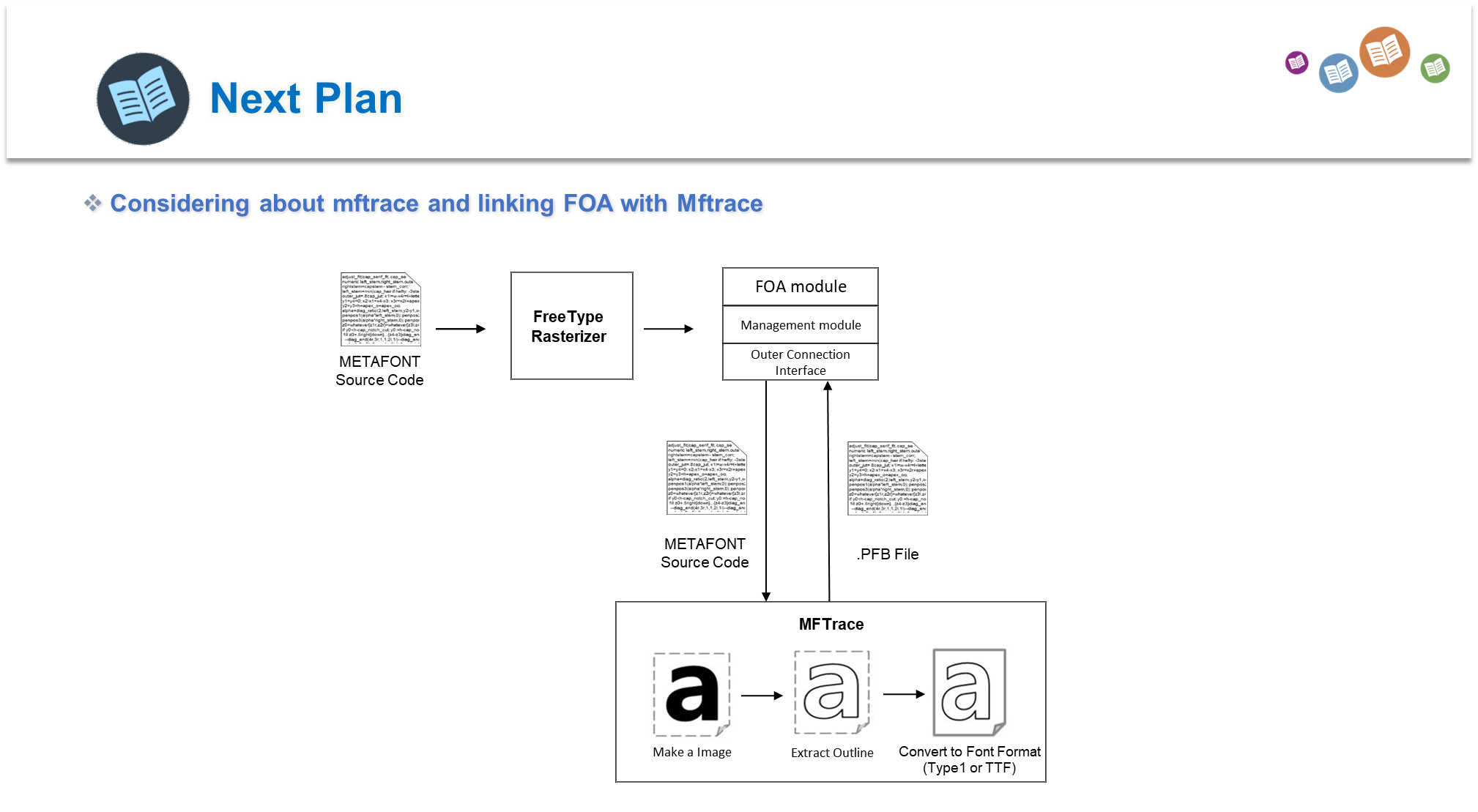
**First approach:**



**Second approach:**



**MAIN approach considering my Project and FOA module purpose:**

**This approach is responsible for following tasks**

1. An application program that is responsible for displaying font on screen sends a METAFONT file to base layer

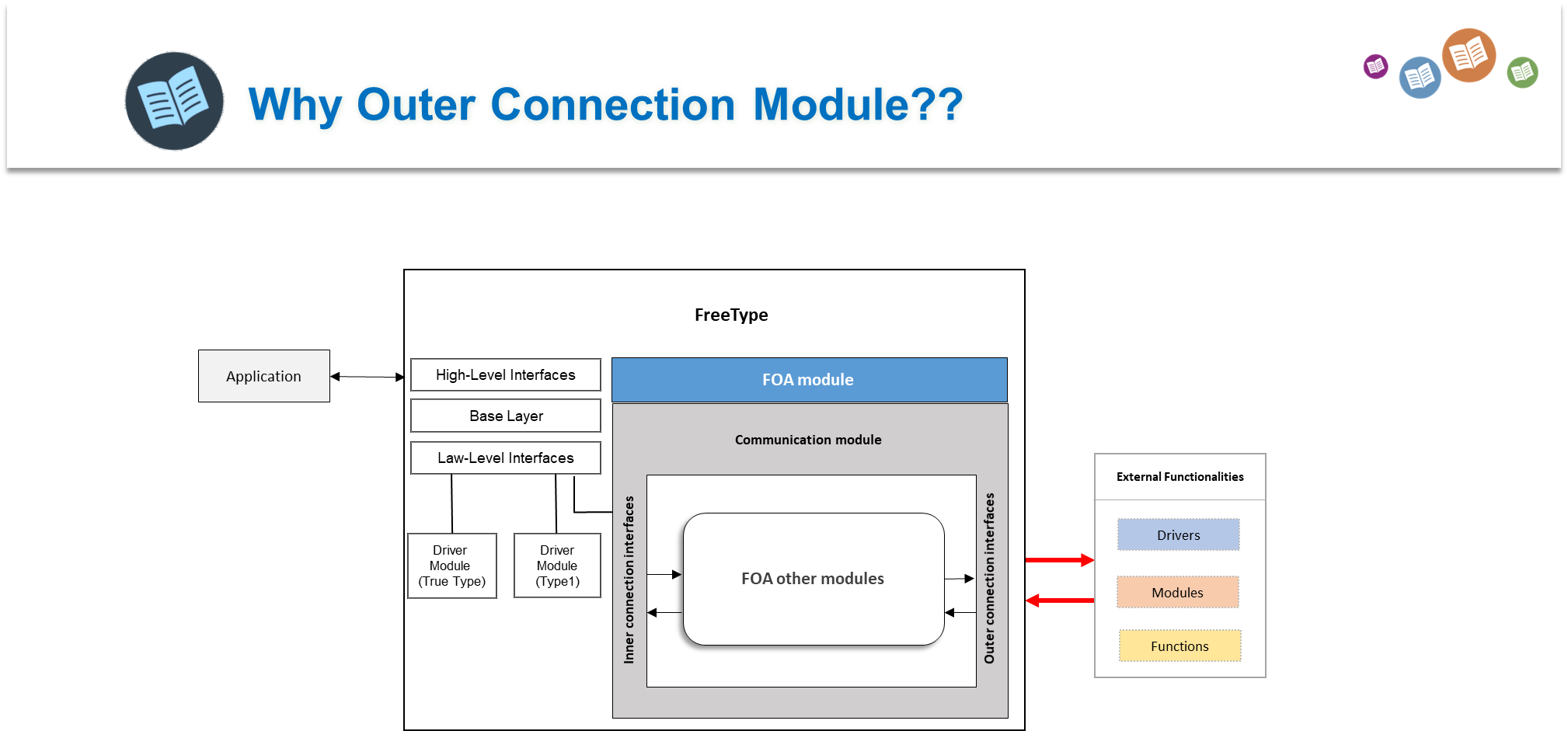
* **(The Xft application which I made uses font pattern metadata by using fc-list command (and copying that in my program) but this way I cannot input a METAFONT FILE to FreeType? So, do we need to consider about Ftview demo program of FreeType which displays a font on screen??)**

1. Base layer calls FOA module that sends this file to Mftrace which is external program in this scenario

* (**As, driver modules are responsible for getting all the font related information like face, character maps, glyphs etc. Our FOA module cannot be considered a driver module because it is not a font driver which processes a font file so, can I convert this into a helping module and on base layer calling it whenever there is a METAFONT file? Will it be a good approach?**)

1. Mftrace sends back a type1 font file created in some specific directory to FOA module.
2. **Now how will FOA module pass this font file to Type 1 driver module?**

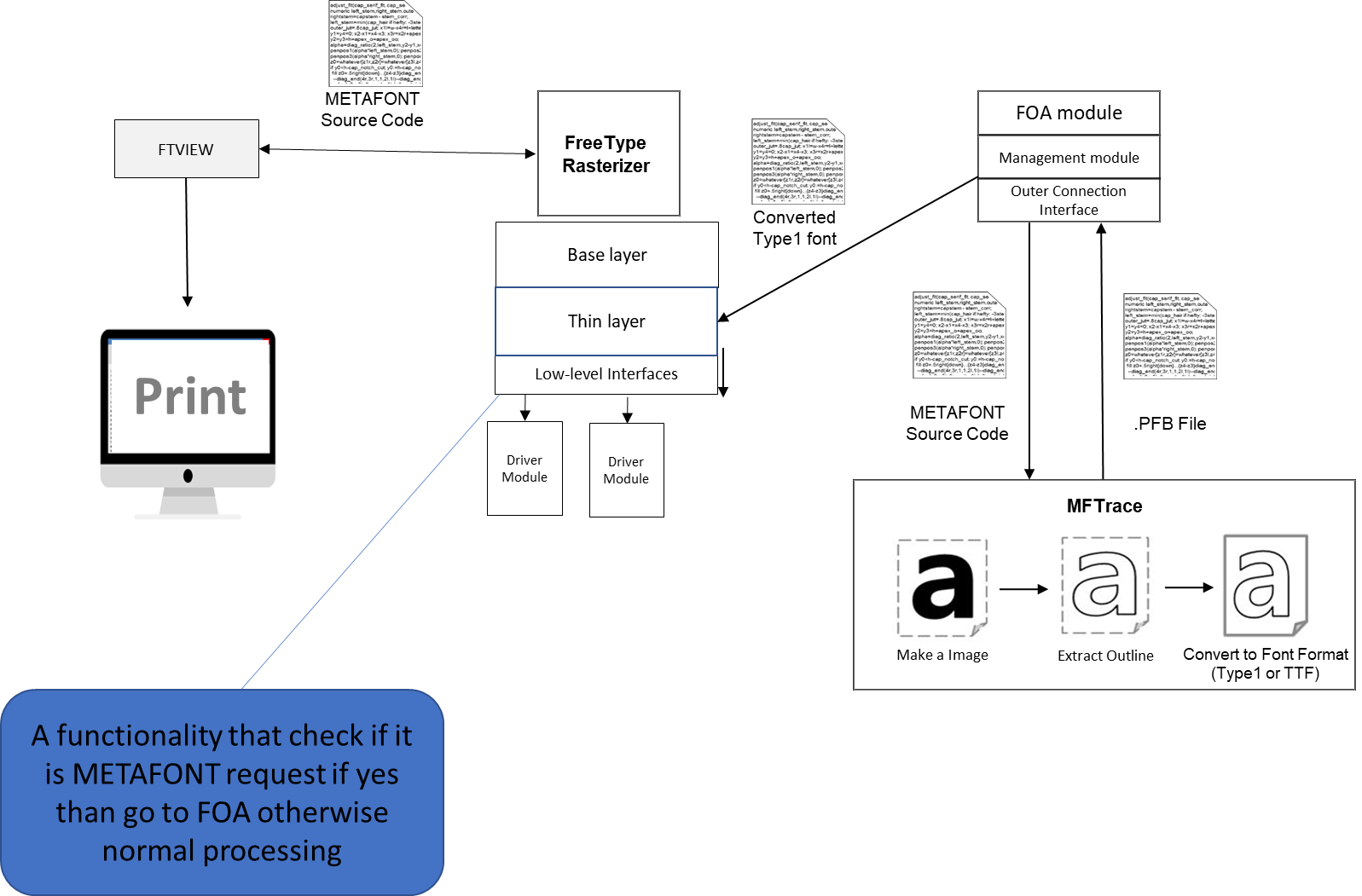
* **(Because base layer is responsible for iterating a loop over font drivers we cannot directly call Type1 driver module. If we want to call directly a driver module we must make a service layer approach.)**



To overcome these Questions, my current approach will be like as follows:

1. For client application that will input a METAFONT and display on screen will be FtView which is demo program of FreeType and is a font viewer.
2. Request will come to base layer as arguments using FT\_New\_Face() to iterate on drivers etc. before sending the request to driver I will check manually using custom functionality that if the requested font is METAFONT (By checking the arguments of file path that it contains .mf file)
3. If it’s a METAFONT file FOA will do the processing of sending file to external program Mftrace for converting METAFONT file into Type1 file
4. When new Type1 file is successfully created and returned to base layer (when function call for FOA is finished) normal routine is resume and FreeType just works normally and passed the file (new type1 font file from mftrace ) to Type 1 driver.
5. And Rendering can be done using Ftview font viewer.

**NOTE: FOA will be made helper module in this whole process because font drivers are responsible for font file processing**

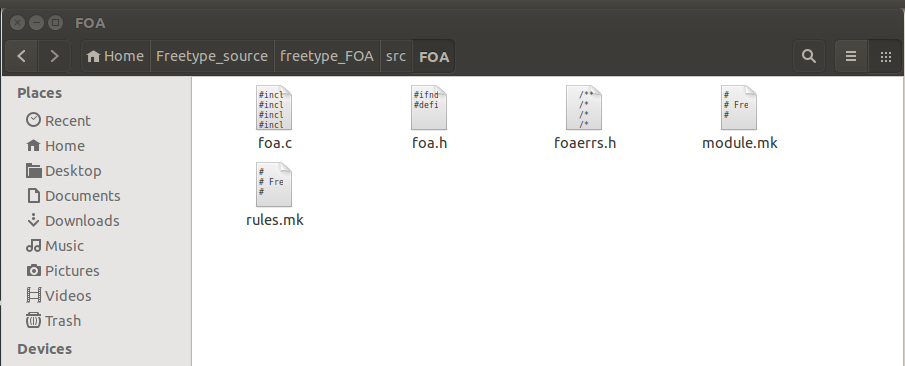
****

**Creating FOA Helping module for FreeType**

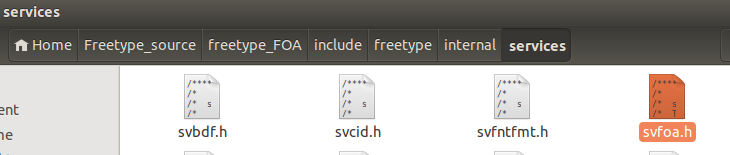
As FOA module will not be a driver module because it is not responsible for processing a font file in detail. We have decided to convert FOA driver module into FOA helping module. Helping module in FreeType are those modules which are responsible for sharing functionalities between other modules. FOA helping module will be responsible for dealing with Metafont file. It will be called on Base layer to check that if the given input file for processing is Metafont. If yes than it will interact with Mftrace external module for converting Metafont file into Type1 file and pass it to its driver module etc.

**Steps for creating a new Helping module in FreeType**

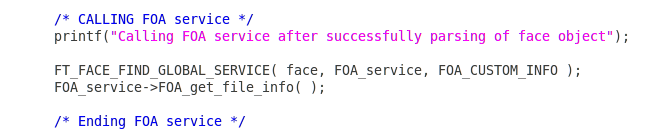
* Create a new folder in FreeType/src directory as FOA
* Create a file FOA.c and implement FOA helping module class (FT\_Module\_Class\_) and service functions for get\_interface
* Other files include foa.h, foaerrs.h, module.mk, rules.mk use reference of other modules to write these as shown in the figure beloww.



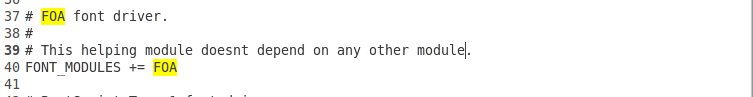
* Once the simple FOA module is setup in src directory it’s time for creating a service layer for this module. Service layer will allow us to use its module get interface functions all over FreeType.
* In the directory path **FreeType/include/freetype/internal/services** we create a new file called svfoa.h that contains the name of the service and the function and their prototypes. For more detail about how services are created refer to this link <https://www.freetype.org/freetype2/docs/design/design-6.html>



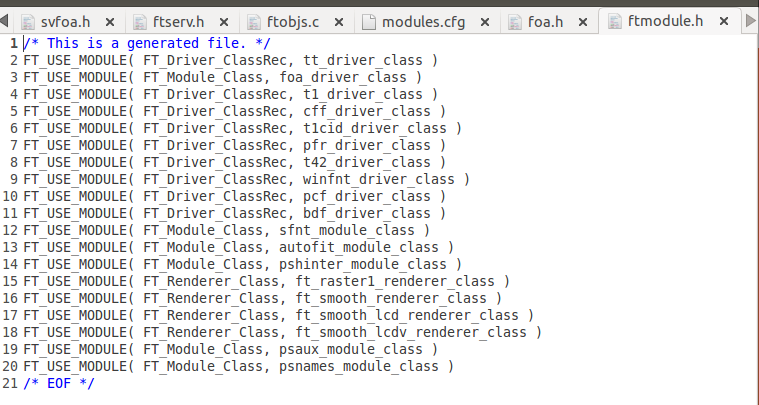
* After creating the service, we must create a marcos in include/freetype/internal for our new FOA service header file.
* We must call in base layer somewhere to our new FOA module by invoking one of its function under service we created.
* For this modify the file src/base/ftobjs.c which contains all the functions related to new faces i.e. whenever FreeType is invoked
  + Include the service header file marcos u created in ftobjs.h file
  + Declare object of FT\_Service\_FOAInfo so that u can call functions contained in this Service. This is our FOA service
  + Now select any position u want to call the FOA service function and using this code u can access FOA service and its functions



* Once base layer is updated with new service call now is the time to add the module in FreeType. For this we must update module.cfg on root of FreeType folder. Add the snippet and module is registered inside FreeType.



* After configuring FOA u will notice that a new ftmodule.h is created automatically by FreeType in objs directory



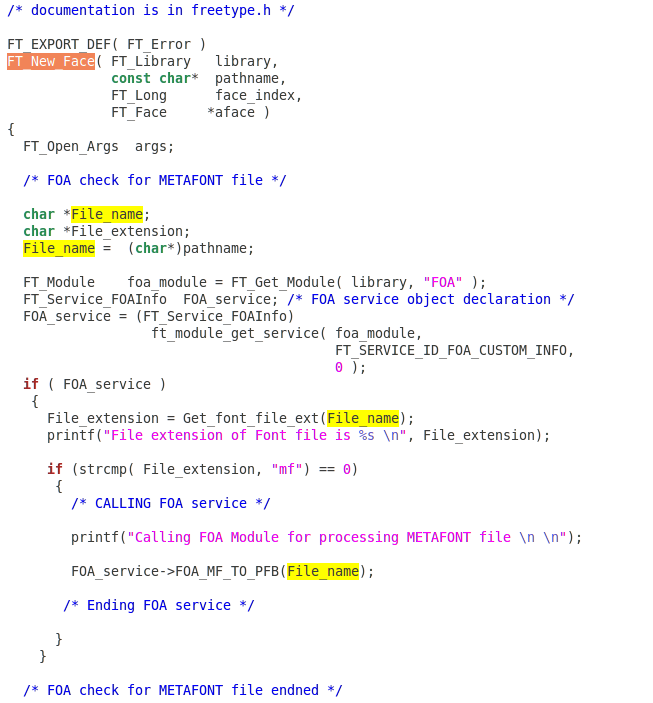
**How to call FOA module in base layer of FreeType**

Once the module is setup and ready to use there should be some way of calling this module function where ever we want on FreeType. As our FOA module is a helping module that will allow users to directly render METAFONT using FreeType. We need to check in base layer that if a font file that is inputted to FreeType is METAFONT or not. If yes than we must call FOA module for further processing.

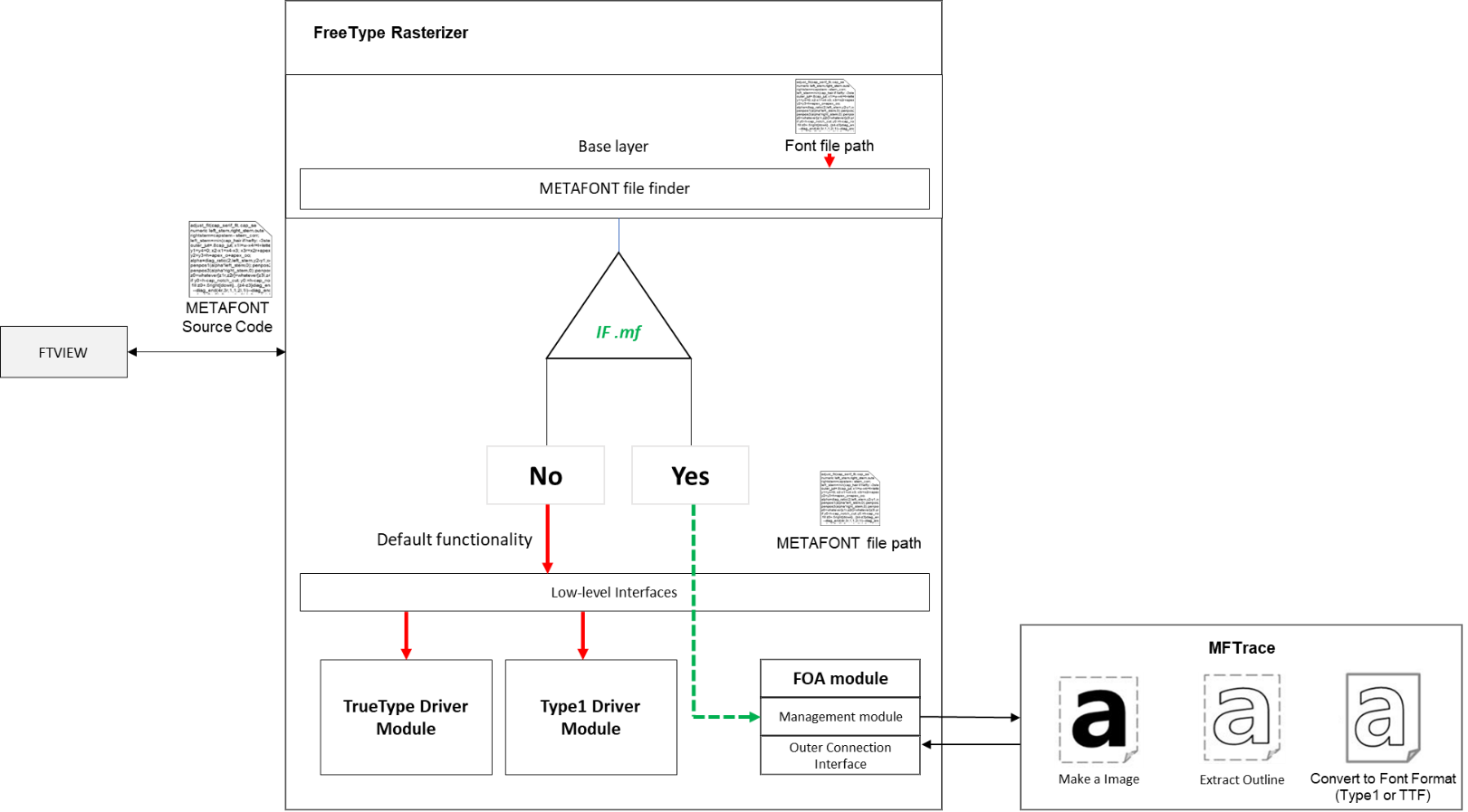
Every client application which renders the font on screen needs to pass the font file path to FreeType’s FT\_New\_Face () this function calls other functions of base layer for dealing with font file and once it finds the driver that parsed that font file header it performs other operations.

So, we have decided to modify this function and we add our helping layer inside this function. Our layer will do the following tasks

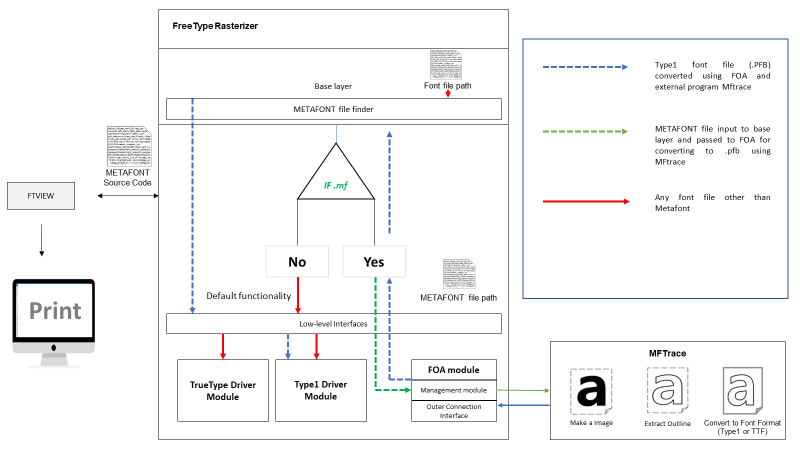
* Call the FT\_Get\_Module () to load FOA module
* Load FOA service of this module object
* From font file path extract its extension (writing custom function for this task)
* Call the FOA module function if extension is “mf” i.e. some\_metafont\_file.mf
* Function will be responsible for further processing



**Figure below the code in form of architecture**



**Detailed Architecture for displaying METAFONT using FOA and Mftrace is**

****

**Output after implementing above architecture 😊**

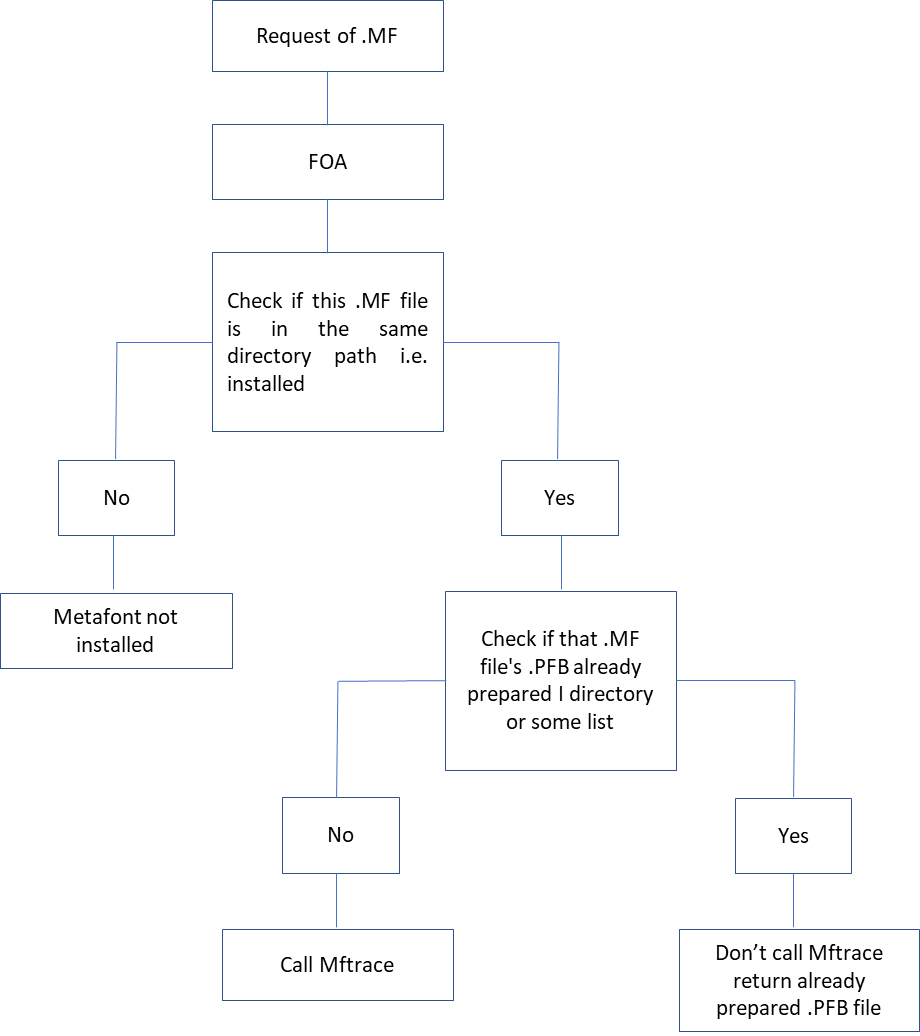
A screenshot of a cell phone

Description generated with very high confidence

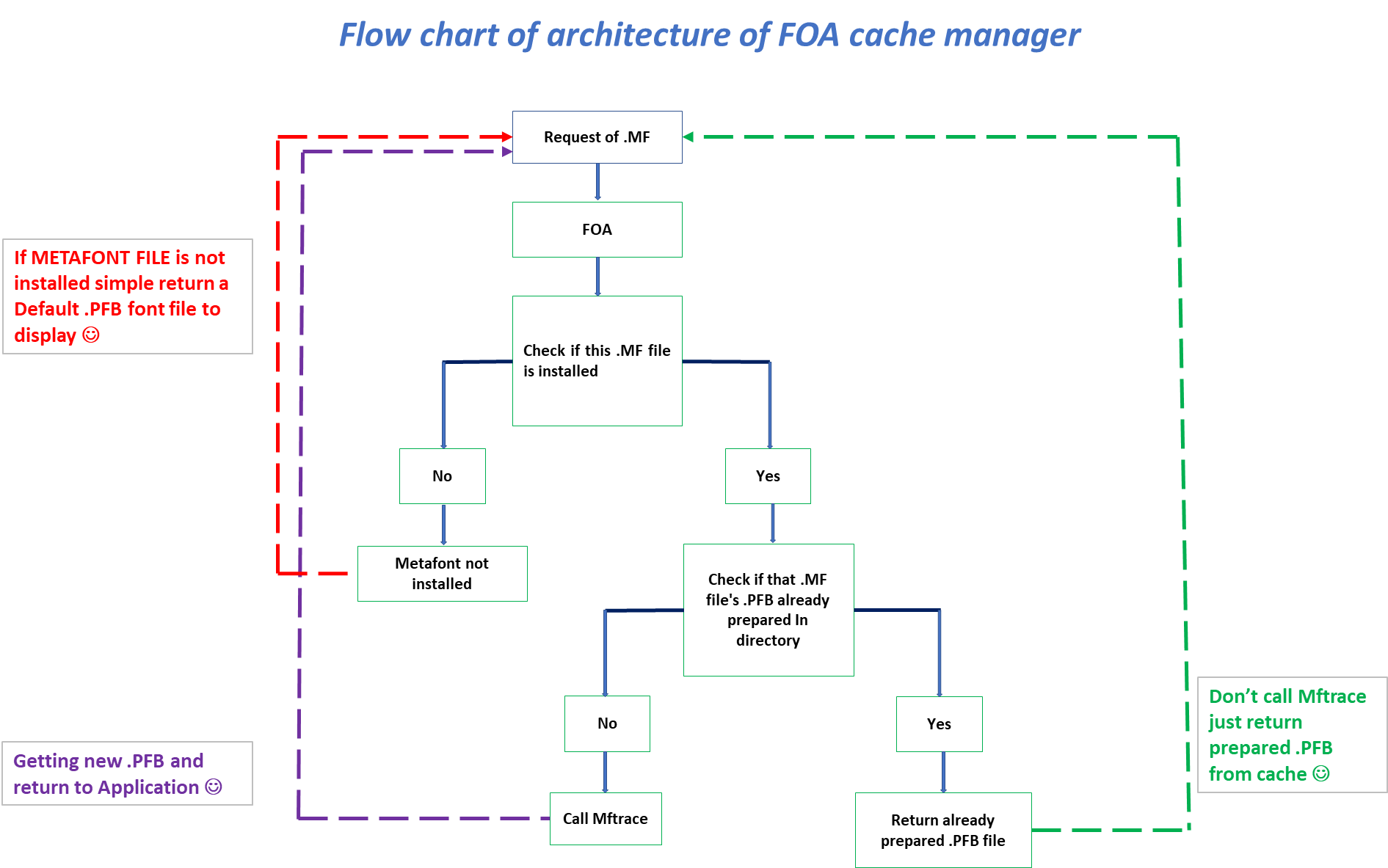
**Creating a kind of cache management system for FOA**

So, after setting up basic FOA module, getting results, etc. it’s time for making it more in detail. For that one sub module in management layer is very important. That module can be named as **FOA cache manager.** This module will be responsible dealing with METAFONT storage management. Like when a request will be done to FreeType to display Metafont on screen, this module will be responsible for following tasks.

* Check if that METAFONT file is present in cache (some table list or directory)
* If no, send not found flag which means that METAFONT is not installed or display some default Metafont converted .PFB on screen
* If yes, then again check, if that requested METAFONT file’s .PFB is already prepared in cache or not.
  + If yes, don’t call Mftrace, just return the converted .PFB file directly to sender of request
  + If no, Call Mftrace and prepare .PFB file and store that new PFB in cache

**Flow chart of architecture FOA cache manager **

**Detailed View**

****

**Testing performance results**

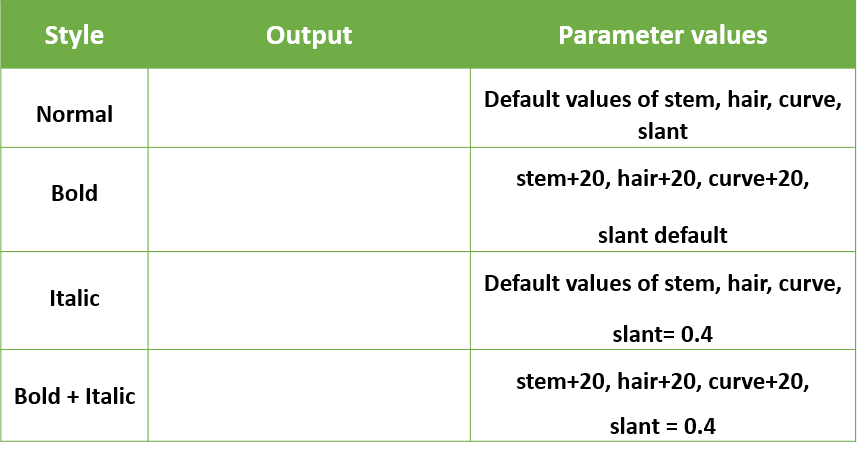
|  |  |  |  |
| --- | --- | --- | --- |
| **Style** | **Output** | **Font files** | **Time (m.s.)** |
| Normal |  | FreeSerif.ttf | 3 ms (3-5) |
| Bold |  | FreeSerifBold.ttf | 4 ms (4-6) |
| Italic |  | FreeSerifItalic.ttf | 4 ms (4-6) |
| Bold + Italic |  | FreeSerifBoldItalic.ttf | 5 ms (5-7) |

Free Serif font family and their performance with FOA module

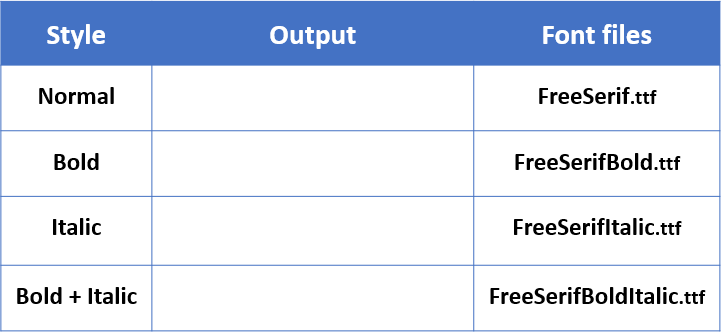
|  |  |  |
| --- | --- | --- |
| **Module type** | **Font name** | **Time (m.s.)** |
| Truetype default driver module | Freeserif.ttf | 6.5 ms (4-9) |
| Type1 default driver module | Century Schoolbook L Roman | 5.5 ms (3-8) |
| FTL module | Computer Modern | 8 ms (6-10) |

|  |  |  |
| --- | --- | --- |
| **Style** | **Output** | **Font files** |
| **Normal** |  | **FreeSerif.ttf** |
| **Bold** |  | **FreeSerifBold.ttf** |
| **Italic** |  | **FreeSerifItalic.ttf** |
| **Bold + Italic** |  | **FreeSerifBoldItalic.ttf** |

|  |  |  |
| --- | --- | --- |
| **Style** | **Output** | **Parameter values** |
| **Normal** |  | **Default values of stem, hair, curve, slant** |
| **Bold** |  | **stem+20, hair+20, curve+20,**  **slant default** |
| **Italic** |  | **Default values of stem, hair, curve,**  **slant= 0.4** |
| **Bold + Italic** |  | **stem+20, hair+20, curve+20,**  **slant = 0.4** |







|  |  |  |  |
| --- | --- | --- | --- |
| **Style** | **TrueType driver**  **Avg. Time** | **Meta Integrator driver**  **Avg. Time** | **MFCONFIG**  **Avg. Time** |
| **Normal** | 4.5 ms (3-6) | 6 ms (5-8) | 70 ms (50-80) |
| **Bold** | 4 ms (4-6) | 7 ms (6-9) | 85 ms (70-100) |
| **Italic** | 4 ms (4-6) | 6 ms (4-7) | 105 ms (70-110) |
| **Bold + Italic** | 5 ms (5-7) | 8 ms (6-10) | 100 ms (90-120) |