FAMP

Function And Macro based Protocol

Please Read:

As of current, **FAMP** has to be installed *locally*. It is currently a work-in-progress to make it to where **FAMP** *does not* have to be installed locally.
It is also a work-in-progress for **FAMP**-related tools. This includes tools that initialize a **FAMP** project, reset projects, configure projects, run projects etc.
This documentation was created, and written, while beta version of **FAMP** were being written. Do not take this documentation with more than a grain of salt.

Alongside this, the protocol will not be deemed "done" until the protocol is capable of offering developers an API of which "libraries" can be written for the protocol. Read more on **FAMP**s api for writing libraries [link_to_page_here].

I assume you have gotten this PDF from my Github. If you find any possible errors(spelling mistakes, general mistakes, anything) then don't hesitate to tell me **@mocacdeveloper@gmail.com**

Introduction

FAMP stands for **F**unctional **A**nd **M**acro-based **P**rotocol. **FAMP** was inspired by the new uprising modern boot protocol *Limine*. *Limine* is based on a user-request/protocol-response type of system. **FAMP**, on the other hand, is based on a system where things happen *upon macro definition/function calling*. I decided I would take this approach because I liked the overall idea of "defining the protocol as the OS is written". The protocol depends on macro definitions so it knows what functionality it needs to have to suit your needs. It is rare to find yourself filling out a struct to accomplish tasks, however, I cannot promise you *won't ever* find yourself doing this.

Although the whole user-request/protocol-response system is quite clever, my decision to stick with a design of which the protocol depends on macro definitions and function calls has made the journey decently smooth. As for the pros of this particular design? That is a question that will most likely vary depending on the person.

FAMP aims on being able to be used with any sort of skill level. The aim is to provide every last possible feature/function that can be used to piece together the OS. With this, the developer will have access to all functions that are used to get the OS to work. The developer can, then, decide to use the hundreds of functions, or, decide to use the "top-layer" functions that do all the "dirty work" for you.
For those who want to take control of the wheel a bit more will have the opportunity to become flexible as to how your bootloader code functions all and all. With this, developers who want to take control of the wheel will be capable of implementing *there own* functionality for the way things are done. The protocol further enhances the flexibility by supporting "handles". In **FAMP**, a "handle" is a protocol-related function that can be replaced by a user-defined function. Most, *if not all*, **FAMP** functions are "handles". Certain functions *might not be* a "handle" function due to the fact the operation

of the function is critical.
SO, not only are you given *all the functionality* you need, not only are you given functionality that does the "dirty work" for you, you are also capable of overriding the internal functionality of the protocol as you would like. Obviously you have to make sure you're accomplishing the intended purpose of the given function. But, nevertheless, the protocol *allows you* to redefine it. Odds are, however, you won't be seeing yourself writing your own functionality for the protocol. Instead, what you will be doing is writing an amazing OS using **FAMP**