Introduction

For our fifth assignment, we will continue working on the Nucleo-64. This assignment will focus on optimizing for better performance and perhaps (at your choice) writing some ARM assembly. In addition, we will start to develop an underappreciated skill for professional engineers: the ability to come up to speed on someone else's code.

As before, you will develop a proper GitHub project for this assignment, complete with code, documentation, and README. Details are given below.

Assignment Background

In cryptography, a *key derivation function* is used to stretch a secret—usually a passphrase—into a longer binary key suitable for use in a cryptosystem. By their nature, key derivation functions must be expensive to compute, in order to protect the cryptosystem against brute force dictionary attacks.

One popular(though outdated) key derivation function is known as **PBKDF1**, which is defined in RFC 8018. It derives a key from a password using a salt and iteration count.

Although it is not strictly required for this assignment, you may wish to learn a little about cryptographic hashing functions by reading the introduction to this Wikipedia page: https://en.wikipedia.org/wiki/Cryptographic_hash_function.

To simplify things (somewhat), for this assignment you are being given a completely insecure hashing algorithm, which is called ISHA: The Insecure Hashing Algorithm. You are also provided reference C source code which computes PBKDF1, based on this insecure hashing algorithm ISHA.

Your assignment has following deliverables:
- Understand and document the current code structure including the call tree.
- Count and report the number of times each function in isha.c is being called before any optimizations.
- Write a simple profiler to count the number of calls for all functions in isha.c based on the systick timer by checking the program counter. Document and compare this call count with numbers found in previous step before performing any optimizations.
- Optimize the code for performance improvements.

Details of each of these are discussed below.

Overview of the Source

The reference PBKDF1-ISHA project may be found in your GitHub repository. Clone this repository on your local machine, start the STM32CubeIDE and **select File -> Open Projects from File System**... Click the Directory button and browse to the location of your repository.

What follows is a brief overview of this code:

- The files `isha.h` and `isha.c` implement the underlying ISHA algorithm. There are three externally visible functions in the ISHA API:
    - **`ISHAReset`** is used to restore an ISHA context to its default state. You can think of this as an initialization function.
    - **`ISHAInput`** is used to push bytes into the ISHA hashing algorithm. After a call to ISHAReset, ISHAInput may be called as many times as needed. Like SHA-1, the ISHA algorithm can hash up to 2^{61} bytes.
    - **`ISHAResult`** is called once all bytes have been input into the algorithm. This function performs some padding and final computations, and then outputs the ISHA hash—a 160-bit (20 byte) value.

  As a note, the ISHA code and API is exactly the same as the code and API used for SHA-1 hashing, with the exception that the internal function **`ISHAProcessMessageBlock`** has been tremendously simplified from the SHA-1 version.

- The files `pbkdf1.h` and `pbkdf1.c` rely on the ISHA API. There is one main function here:
    - **`pbkdf1`** can be used to derive keys of arbitrary length, based on a password and salt specified by the caller. (You can think of the password and salt as arbitrary character strings; for instance, in Wi-Fi authentication, the password is what the user considers to be the password for a given Wi-Fi network, while the salt is the SSID's name.) The output of PBKDF1 is a derived key, DK, of the length in bytes specified by the dkLen parameter.

  This function is fully standards-compliant: If you were to substitute a proper SHA-1 implementation into the pbkdf1 function, you would have regular PBKDF1 implementations, and you could compute the key necessary.

- The files `pbkdf1_test.h` and `pbkdf1_test.c` provide a small test suite to ensure correctness. Two functions are provided (**`test_isha`**, and **`test_pbkdf1`**); these functions execute test cases against the corresponding function being tested.

- The file `main.c` is the application entry point. Importantly, within this file you will find the function for time reporting.

Our goal is to profile the functions `isha.c` and improve the speed of pbkdf1 as reported by this function.

Implementation Details

To set you on the right track, you should first write a summary of how the code currently behaves, with the call tree from main to all isha functions. I recommend doing this prior to attempting any changes to the code, because this exercise will help you learn the current code. You should include in this report the full call stack and the number of times each key function is called. You should also include information about how much time each function consumes overall, to help focus your optimization efforts on the places with highest leverage.

As delivered to you, the code has the following characteristics in Debug mode with STM32CubeIDE(Build 1.15.1) with output set to UART console using Program Flash:

Compile Option	Speed (msec)	Size of .text (bytes)
-O0	694	12,584
-O3	244	11,496
-Os	310	9,712

You might wish to verify these results on your own hardware before beginning any optimization work. Your numbers may be slightly off due to hardware differences but they should be comparable to these.

Your code must be compiled with -O0 in DEBUG mode with UART output(not Console) to get a baseline of these numbers, and it must, of course, pass all of the built-in tests. We are not primarily focused on the size of your code, but your .text segment may not grow larger than 15,000 bytes including the profiler in Debug build with -O0 optimization as discussed below.

If you choose to write assembly code for some of the optimization, the assembly must be meaningfully commented, enough to prove that you understand what each line of assembly is doing and must have at least twenty lines of assembly. Lastly, you must not use more than 3 register variables in each file.

You should change the files `isha.h` / `isha.c` and if needed `pbkdf1.h` / `pbkdf1.c` for time optimization.
However, you can not change any function definitions except for the use of *register* modifier.

You will also need to add your code in ticktime.c for implementing a Program Counter(PC) based profiler. The other files in the tree are test code and necessary timing structure, and they should not be modified. If needed, you may add additional files but I would not expect it to be necessary other than the code required for PC base profiler. Document overview of all changes in your README file and put generous comments in your code to show the code fragments before and after your changes.

You may discover changes that would improve performance but would reduce the generality of the functions you are optimizing. This, in fact, is a common conundrum when optimizing code for embedded use. You may make such changes (provided your modified code still passes the built-in test suite), but you should document the resulting limitations you have added to the code.

I recommend *frequently* saving aside your work—ideally at every point where you are passing the tests. While optimizing, it is not unusual to inadvertently introduce a bug that causes you to start failing one or more test cases. When this happens, it is extremely convenient to have a recent, known-working checkpoint handy for comparison.

Following the guidelines above, I was able to achieve the following results in Console mode with Release configuration:

	-O0	-O3
C changes only with .text segment size	225 msec, 14592 bytes	225 msec, 12388 bytes
C changes, plus re-write one function in assembly with .text segment size	187 msec, 14706 bytes	187 msec, 12060 bytes

Since you are modifying existing code, your modifications should follow the coding style used in the original. Be sure not to leave “decoy code” (that is, code that is commented or #ifdef’d out, but not operational) in your checked-in files except to illustrate the changes you are making.

In previous assignments, you have created DEBUG and RUN build targets with slightly different behavior. Because the focus of this assignment is on optimization, you only need one build target which will be RELEASE(or RUN) and report the final optimization time and size from that build.

Profiler(Call Counting) Details

You will perform call counting before optimizations with two different methods and report the results in your README for all functions in `isha.c`. You only need to perform the call counting during **the time test**. See the `main` function for commented out statement blocks.

1. STATIC profiling: By using a static variable in each of the functions in `isha.c`. This will give the exact number of times functions in file `isha.c` are getting called.
2. PC profiling: By writing your profiler which will give approximate number of calls for each function in file `isha.c`. Here, you will need to modify the systick handler function and decipher the program counter(PC) when the systick interrupt occurred by examining the stack. The function `GetFunctionAddress` in `isha.c` should be called when the profiler is turned on to get the addresses of the functions in `isha.c`. With the PC in systick_handler, you can check if it belongs to one of the functions from isha.c and tally up the number of times the PC is found in one of those functions. You then need to print the total for each function in the *print_pc_profiler_summary* function. Ensure that your systick handler is not getting optimized by using the gcc pragmas. This way, your PC value will stay the same for -O0 vs -O3 optimizations.

   This method is mostly used when you don’t have access to or want to modify the source code that is being profiled. For example, if you did not have `isha.c` source code but it was only

available as an object file or in a library, you will need to use PC profiling and match the addresses of PC with the function address from the object file.

There are three sections in `main.c` where time tests is being performed:

1. Section 1: First is the general time test that you should be reporting for the execution time.
2. Section 2: This section is used for function call counting with static variables. You will need to implement the functions in the file `static_profiler.c` and also change functions in `isha.c` to update the counters for each function when the 'static_profile' has been turned on.

   void static_profile_on() ; // Turns on the call counting with static variable. Initializes the counters to 0.
   void static_profile_off(); //Turns off the call counting with static variable.
   void print_static_profiler_summary(); //Should print the static profiler results.

   The profiler summary should be printed as follows and will appear on Console or Terminal with a printf:

   Function: ISHAProcessMessageBlock Call count: <value>
   Function: ISHAPadMessage Call count: <value>
   Function: ISHAResult Call count:<value>
   Function: ISHAReset Call count:<value>
   Function: ISHAInput Call count: <value>

3. Section3: This section used for function call counting estimates with PC profiler(see below) and declared in `profiler.h`. Here, you will implement the PC profiler and the functions to turn the PC profiler on/off and print the summary in a file `profiler.c`.

   void pc_profile_on(); //Turns on the call counting with PC profiler
   void pc_profile_off();//Turns off the call counting with PC profiler
   void print_pc_profiler_summary(); // Print a summary of pc profiler results.

   The profiler summary should be printed as follows and will appear on Console or Terminal with a PRINTF:

   Function: ISHAProcessMessageBlock Call count: <value>
   Function: ISHAPadMessage Call count: <value>
   Function: ISHAResult Call count:<value>
   Function: ISHAReset Call count:<value>
   Function: ISHAInput Call count: <value>

Submission Details

This assignment is due on Wednesday, October 30 at 9:00 am. At that time, you should submit the following:

For this assignment, your repo should have an STM32CubeIDE project containing multiple .c and .h files. You can embed your optional assembly code in .c files or have separate .s files. **Please put all your sources including assembly code if any in sources directory.**

- In your README, please include your written report on how the code you were handed behaves.
- Your code must be tagged with "ready-for-grading" before the due date.

I expect that your STM32CubeIDE project will be based on the code I am delivering with the working-but-slow code.

You should report in your README what you changed in the code, and what speed you are now seeing, as reported by `time_pbkdf1 function`. Please also report the size of your .text segment.

Grading

Points will be awarded as follows:

Points	Item
15	README summarizing the code structure(3 pt), time your code is taking(1 pt), the size of .text segment(1 pt), static call count of isha functions(5 pt), PC profiler call count of the isha functions(5 pt), before any code optimizations.
10	Does your README.md report the time your code is taking post optimization(3 pt), does it include the size of your .text segment(2 pt), and is the .text segment the same or smaller than the limit 15000 bytes with Debug config and -O0 build(5 pt).
15	Code elegance: General code elegance(3 pt). Do your changes match the style of the surrounding code? (3 pt) Is your submission free of "decoy code"?(3 pt) Did you comment any tricky code you wrote in a clear way, so that it could be maintained in the future?(3 pt) Did you use less than 3 register variables per file?(3 pt)
15	Does resulting code post optimization pass all built-in tests? (Points are all or nothing: You must fully pass the test suite.)

10	Does your code have a static profiling and does it generate the call count correctly before any optimizations ?
10	Does your code have PC profiling and does the code runs and looks correct?
15	Does your code run in less than 230 msecs(Release config, -O3, Console output)?
10	Does your code run in less than 225 msecs(Release config, -O3, Console output)?
10	Extra credit: Did you write at least something(min 20 instructions) in assembly in this project and your code runs in less than 220 msecs? (All or nothing)

Good luck and happy coding!