Introduction

For our fifth assignment, we will continue working on the FRDM-KL25Z. 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 key derivation function is known as **PBKDF2**, which is defined in RFC 8018. PBKDF2 is used in a number of applications, including WPA2-PSK—perhaps the most widespread authentication system used today in deployed Wi-Fi networks.

As used in WPA2-PSK, the PBKDF2 function relies on calling HMAC-SHA1 8192 times; each call to HMAC-SHA1 in turn results in two calls to the SHA-1 secure hashing algorithm. 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.

For our purposes, PBKDF2 based on the full HMAC-SHA1 is quite complex. To simplify things (somewhat), for this assignment I have invented a completely insecure hashing algorithm, which I call ISHA: The Insecure Hashing Algorithm.

I have provided reference C source code which computes PBKDF2-HMAC-ISHA, using parameters similar to those used every time a device connects to Wi-Fi. At present, this reference source code is believed to be correct—that is, it computes accurate results. However, when running on our FRDM-KL25Z board, one call to this function consumes approximately 8.7 seconds when compiled with -O0.

Our marketing department is not happy with this performance, and you have been given the job to improve it.

Overview of the Source

The reference PBKDF2-ISHA project may be found on Canvas as a zip file.

What follows is a brief overview of this code:

- The files `isha.h` and `isha.c` implement the underlying ISHA algorithm. There are three 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 `pbkdf2.h` and `pbkdf2.c` rely on the ISHA API. There are two functions here:
    - **`hmac_isha`** computes the HMAC value for a given key and bytestream, using ISHA as the underlying hash. The output is also a 20-byte value.
    - **`pbkdf2_hmac_isha`** 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 PBKDF2 is a derived key, DK, of the length in bytes specified by the dk_len parameter.

  These two functions are fully standards-compliant: If you were to substitute a proper SHA-1 implementation into the HMAC function, you would have regular HMAC-SHA1 and PBKDF2-HMAC-SHA1 implementations, and you could compute the key necessary to sign on to a Wi-Fi network.
- The files `pbkdf2_test.h` and `pbkdf2_test.c` provide a small test suite to ensure correctness. Three functions are provided (**`test_isha`**, **`test_hmac_isha`**, and **`test_pbkdf2_hmac_isha`**); 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 following function:

```
/*
 * Times a single call to the pbkdf2_hmac_isha function, and prints
 * the resulting duration
 */
static void time_pbkdf2_hmac_isha()
{
  const char *pass = "Boulder";
  const char *salt = "Buffaloes";
```

```
  int iterations = 4096;
  size_t dk_len = 48;
  int passlen, saltlen;
  uint8_t act_result[512];
  uint8_t exp_result[512];
  const char *exp_result_hex = "7577B5FFB058195DE3978773B472E92D0216873EE1A2"\
    "170157C2054EDC41E58D7F949050253F8CE1D55E6B86E62AED3F";

  ticktime_t duration = 0;

  assert(dk_len <= sizeof(act_result));

  hexstr_to_bytes(exp_result, exp_result_hex, dk_len);
  passlen = strlen(pass);
  saltlen = strlen(salt);

  reset_timer();
  pbkdf2_hmac_isha((const uint8_t *)pass, passlen, (const uint8_t *)salt, saltlen,
      iterations, dk_len, act_result);
  duration = get_timer();

  if (cmp_bin(act_result, exp_result, dk_len)) {
    printf("%s: %u iterations took %u msec\r\n", __FUNCTION__,
        iterations, duration/10);
  } else {
    printf("FAILURE on timed test\n");
  }
}
```

Our goal is to improve the speed of pbkdf2_hmac_isha as reported by this function[1].

Implementation Details

To set you on the right track, you should first write a summary of how the code currently behaves, in the form of a technical memo to your development manager. 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:

Compile Option	Speed (msec)	Size of .text (bytes)
-O0	8744	21,020
-O3	2133	17,872
-Os	3147	15,068

[1] Because the parameters passed to pbkdf2_hmac_isha here are different than those used in Wi-Fi, the number of calls to each function lower down in the code is different from the values in the Assignment Background section.

You might wish to verify these results on your own hardware before beginning any optimization work.

Your deliverable must be compiled with -O0, 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 may not grow more than 5% larger than the version given to you (i.e., your .text must be smaller than 22,071 bytes).

If you choose to write assembly code, the assembly must be meaningfully commented, enough to prove that you understand what each line of assembly is doing.

You should change the files `isha.h` / `isha.c` and `pbkdf2.h` / `pbkdf2.c`. 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.

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.

You may use the `register` keyword exactly once, so choose carefully!

Following the guidelines above, I was able to achieve the following results:

	-O0	-O3
C changes only	1729 msec	849 msec
C changes, plus re-write one function in assembly	1187 msec	716 msec

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.

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 the DEBUG build target.

Submission Details

This assignment is due on Wednesday, March 16 at 9:30 am. At that time, you should submit the following:

- A private GitHub repo URL which will be accessed by SAs to review your code and documentation. For this assignment, this repo should have an MCUXpresso project containing multiple .c and .h files. **Please check the MCUXpresso tree directly into GitHub**—do not zip it or extract individual files from it.
- In your repo, please include your written report on how the code you were handed behaves. This document may be in either .docx or .pdf format.

I expect that your MCUXpresso project will be based on the code I am delivering with the working-but-slow code. For this reason, there is no need to submit the PDF for plagiarism checks for this assignment.

You should report in your README what you changed in the code, and what speed you are now seeing, as reported by `time_pbkdf2_hmac_isha`. Please also report the size of your .text segment.

Grading

Points will be awarded as follows:

Points	Item
20	Technical memo summarizing the structure and performance of the code as delivered to you
10	Code elegance: Do your changes match the style of the surrounding code? Is your submission free of "decoy code"? Did you comment any tricky code you wrote in a clear way, so that it could be maintained in the future?
20	Does resulting code pass all built-in tests? (Points are all or nothing: You must fully pass the test suite.)
10	Does your README.md report the time your code is currently taking, does it include the size of your .text segment, and is the .text segment below the target?
10	Does your code run in less than 5.5 seconds?
10	Does your code run in less than 4 seconds?
15	Does your code run in less than 2.8 seconds?
5	Does your code run in less than 2 seconds?
10	Extra credit: Does your code run in less than 1600 msec?

Good luck and happy coding!