PROJECT REPORT

-Bhaskar Ekansh P, 241EE213

Overview:

This project presents the design for a secure pin-locked password protection system on an ARM Cortex M3 based microcontroller, the LPC1768. It has been programmed entirely using the ARM assembly language, in the KEIL MicroVision IDE, without any complex software libraries. It uses low-level programming techniques, is lightweight and hardware friendly.

A 4-digit user PIN is checked against a list of valid pins. On successful authentication, a decrypted password and a success message are written to memory. A simulated LED blink pattern is also triggered using GPIO register-level programming. Incorrect PIN entries result in silent denial of access.

Key Features:

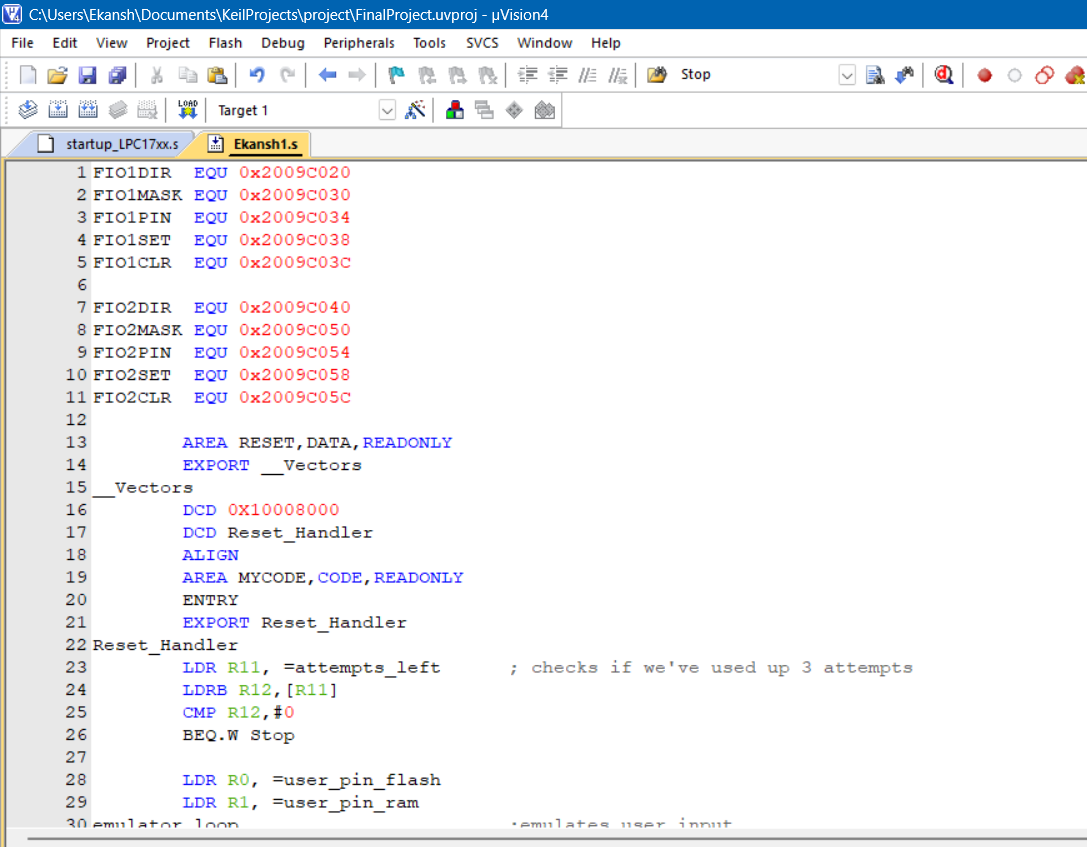
* Supports multiple valid user pins.
* Displays a success message upon correct pin entry.
* Visual confirmation in the form of simulated blinking of LED pins in a preset pattern.
* Maximum of 3 attempts at PIN entry are allowed, with a timeout between successive attempts.
* Simple XOR-based encryption protects the password.

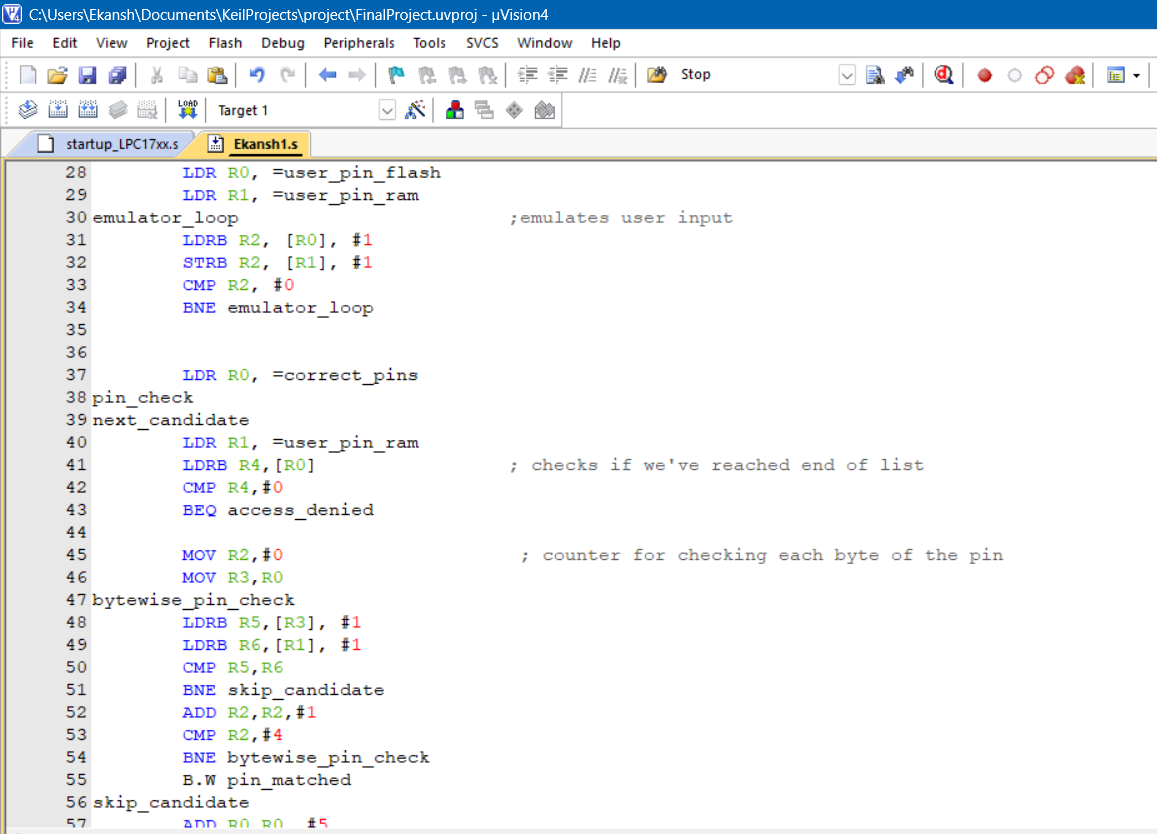
GitHub Repo:

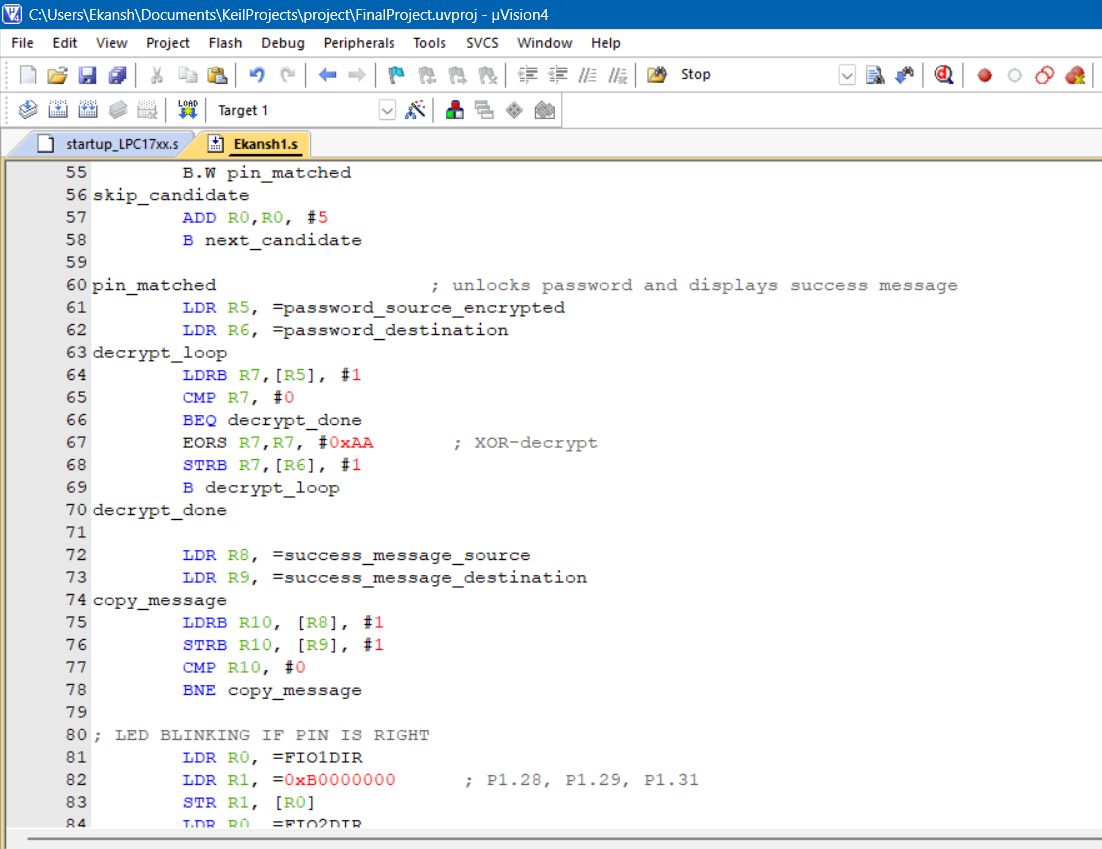
<https://github.com/Ekansh345/lpc176-pin_lock-project.git>

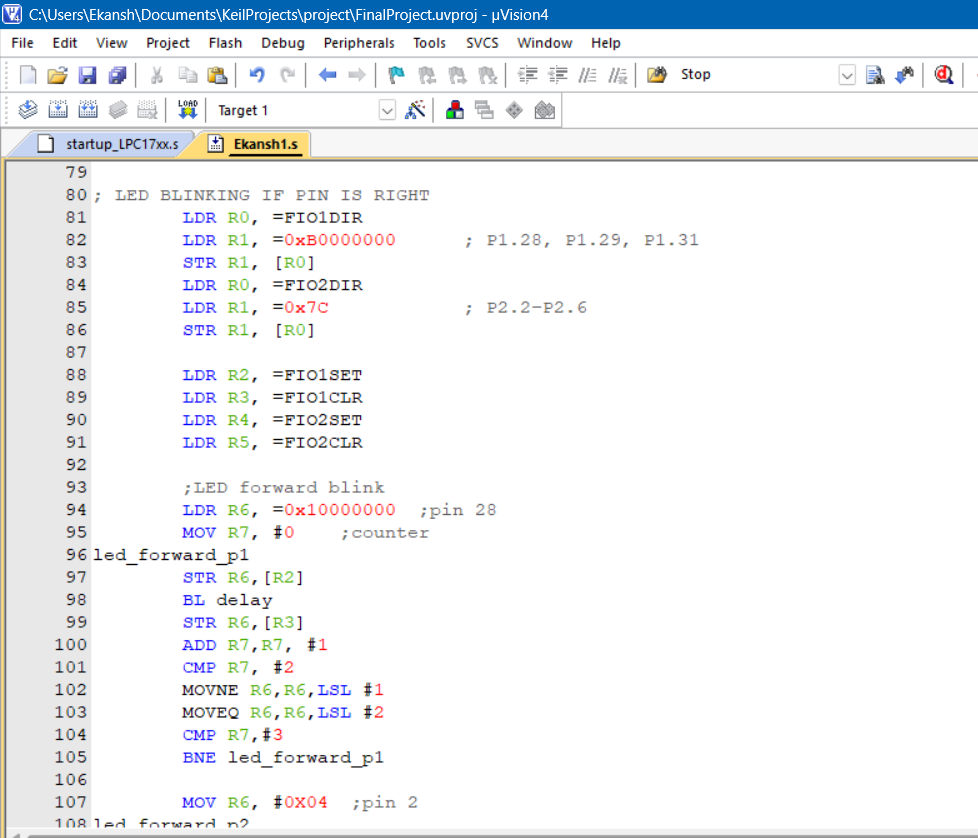
Code Screenshots:

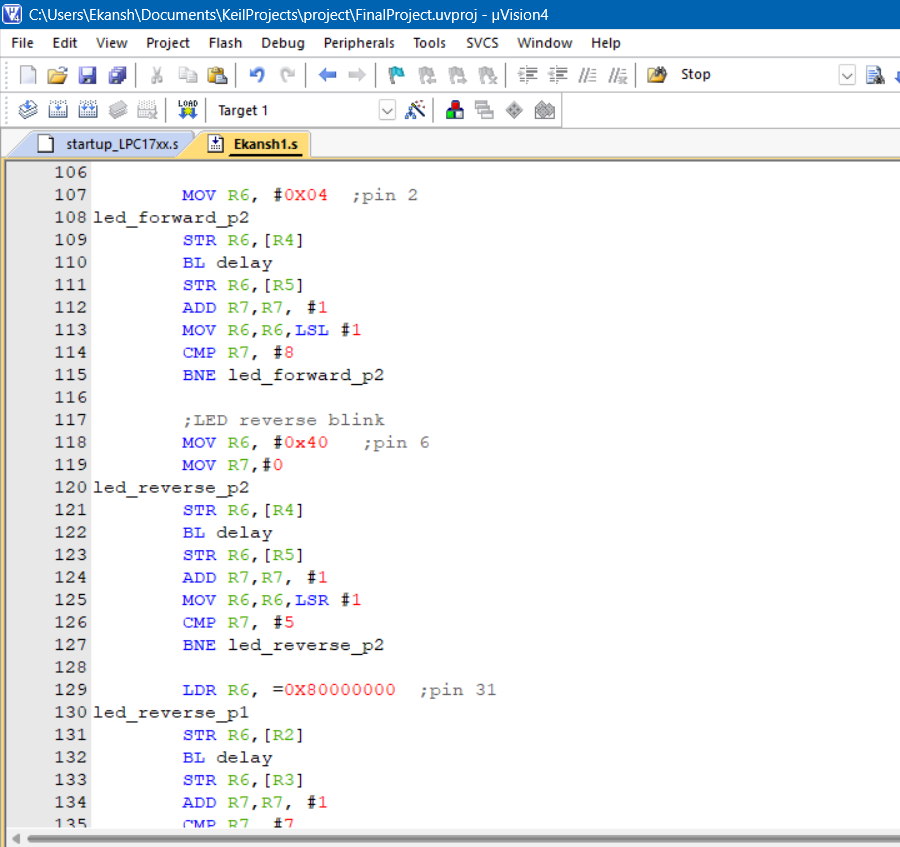
Link: <https://github.com/Ekansh345/lpc176-pin_lock-project/blob/main/Ekansh1.s>

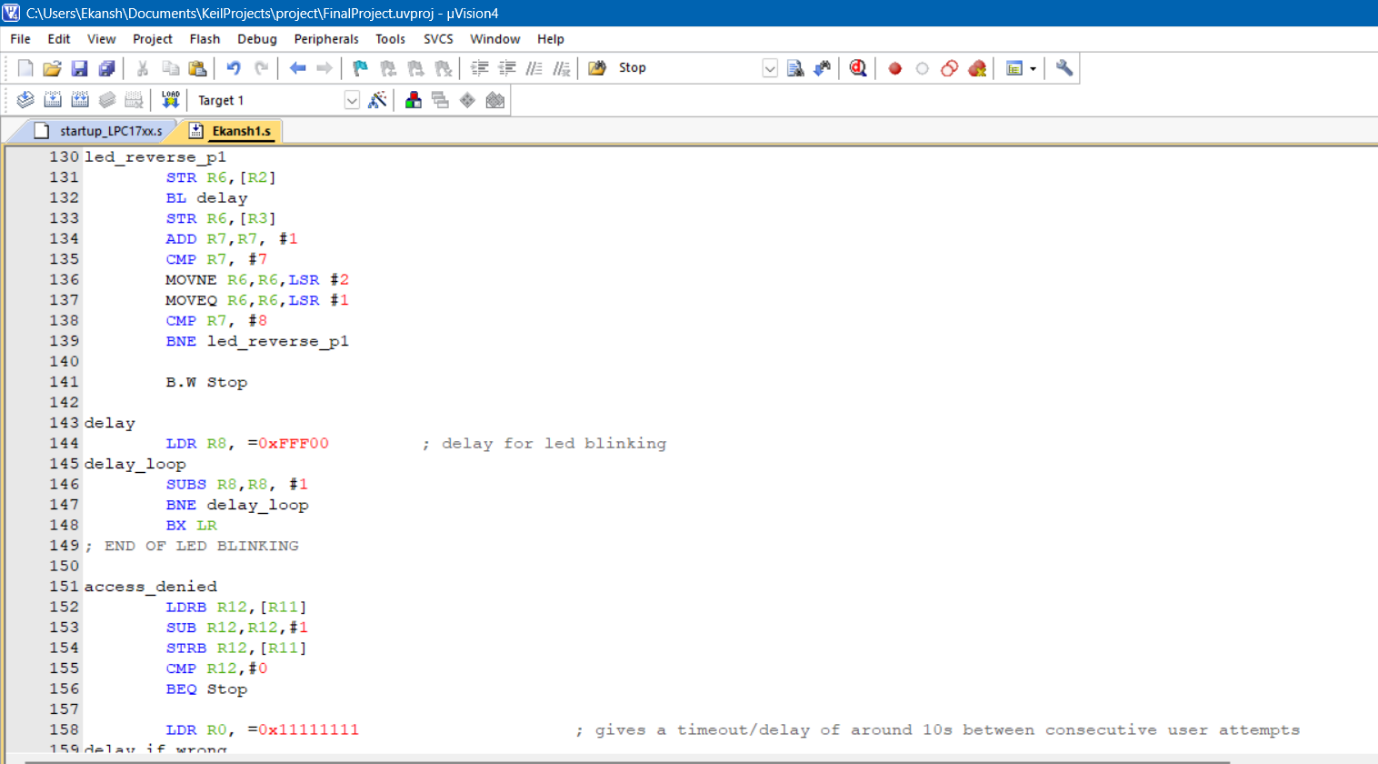


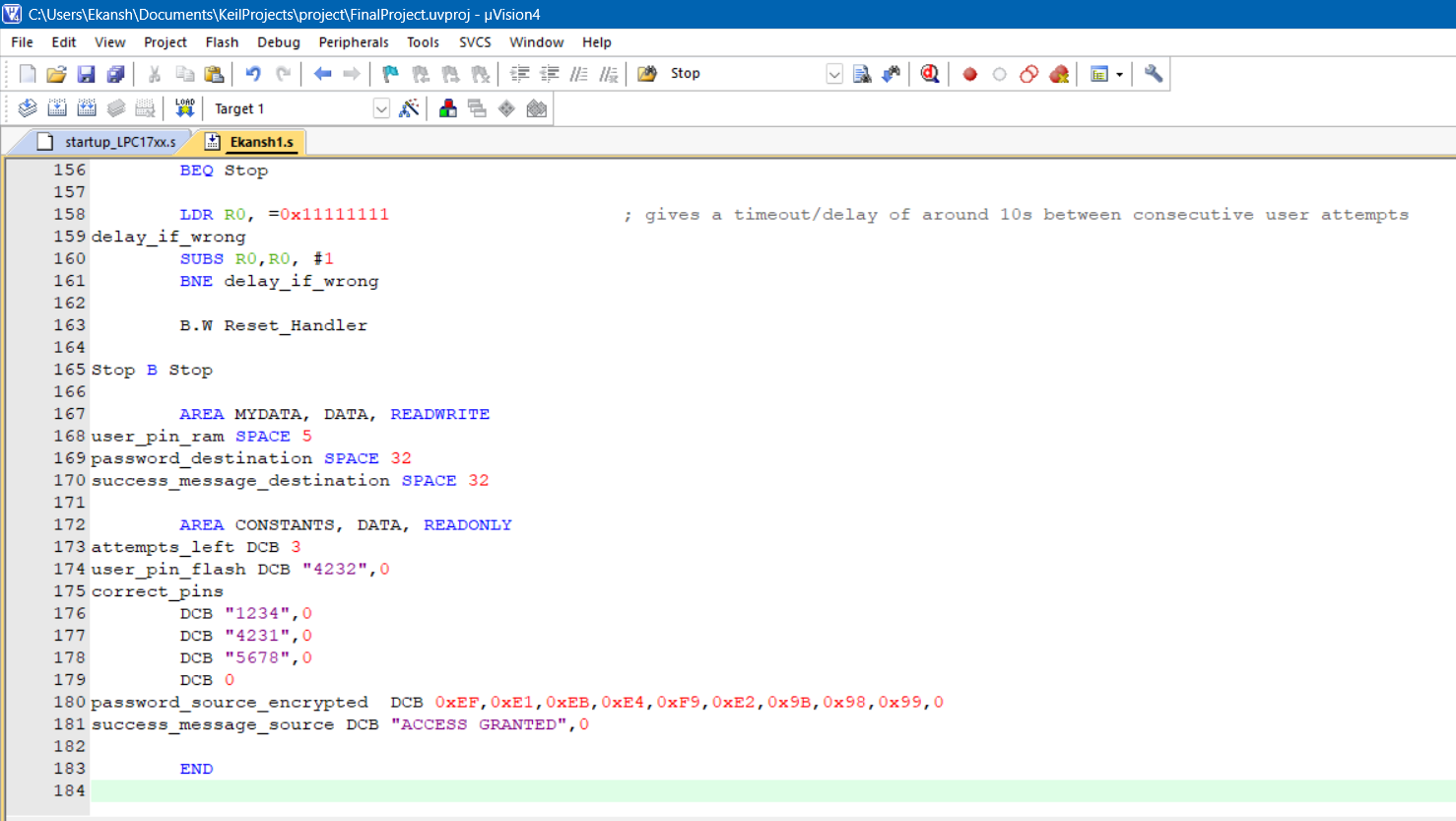












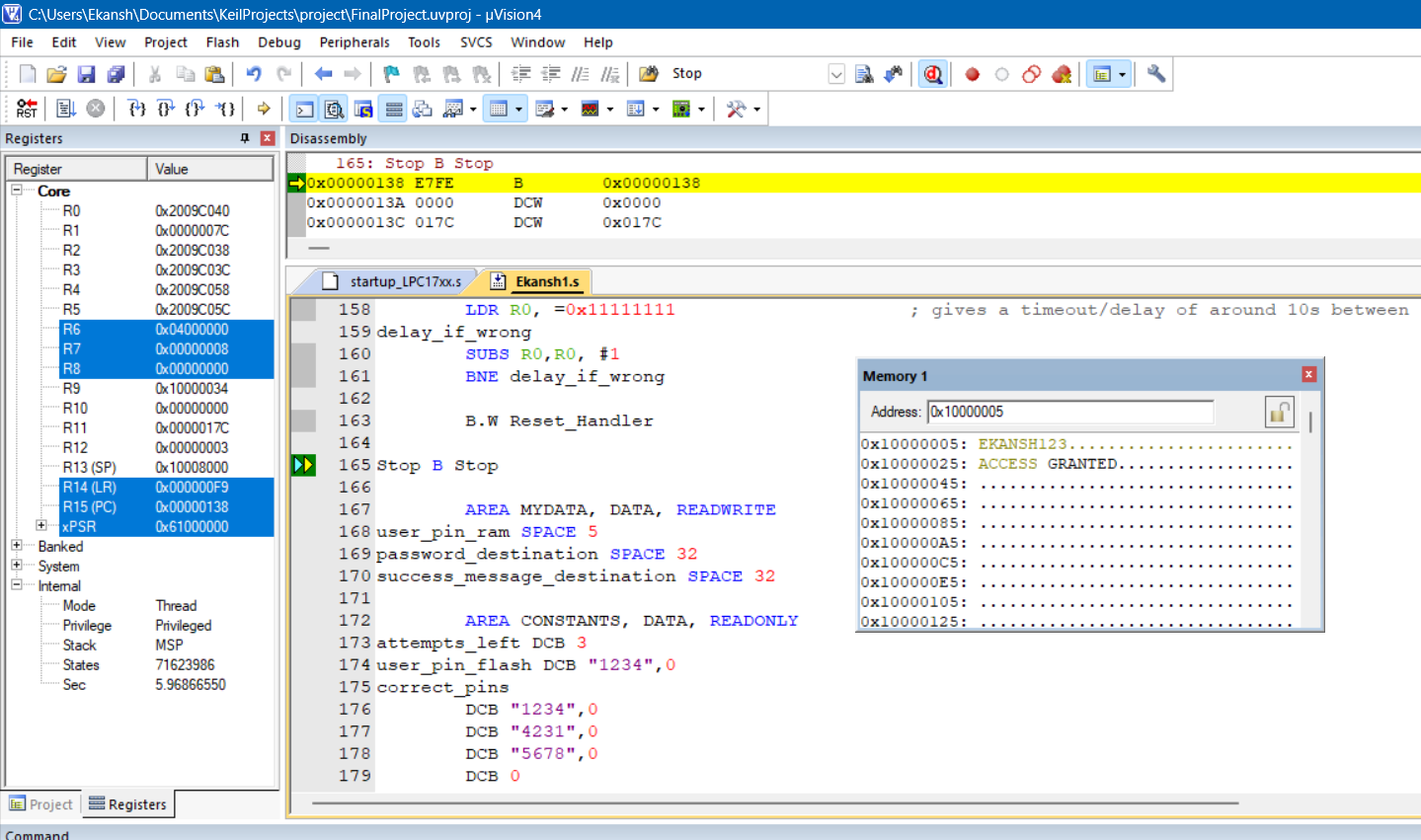
Line-by-line explanation of code:

* Lines 1 to 11- These lines initialise the GPIO registers. FIO stands for Fast Input Output. The number 1 or 2 signifies the port number. The memory addresses for these registers come from the user manual of the LPC1768.
  + The DIR register indicates whether a given pin in the port is set to input or output.
  + The MASK register configures whether the pin is to be used during read/write or it is to be masked.
  + The PIN register reads whether the pin value is currently set to a high or a low.
  + The SET register sets the pin value to a high (1).
  + The CLEAR register sets the pin value to a low (0).
* Lines 13 to 18- These lines define the Vector table. The vector table instructs the CPU on how to handle resets, interrupts, faults etc. Here, we have only defined the “Reset Handler”.
  + AREA is an assembly directive that defines a named section of memory. It gives a label for the section, defines whether that section contains code or data and whether it is read only or read/write.
  + The vector table is placed in a section called RESET, containing data that is read only.
  + \_\_Vectors is the label for the start of the vector table. EXPORT makes it visible to the linker.
  + 0x10008000 is the initial Main Stack Pointer. It points to the start of the “stack”.
  + Reset\_Handler is the label which contains the address of the Reset Handler function. It indicates where to start running code from on reset.
  + ALIGN ensures that the subsequent data is 4-byte (32 bit) aligned. It is an assembly directive.
* Line 18 onwards- From here, the code section begins. It is labelled in memory as MYCODE and placed in read only memory.
  + ENTRY is an assembly directive that tells the linker that this file contains the main entry point of the program. The label that comes after this is where the program starts running.
  + That label is Reset\_Handler. Here, this label is used during debugging/linking to tell the debugger/linker that this is where the user’s program starts from. In the vector table section, it is used by the CPU during actual runtime as the starting point after reset. In both places the label performs different functions but points to the same thing.
  + EXPORT makes the label visible to other objects like linker, debugger.
  + Line 22- This is where execution starts from.
* Line 23 onwards- The actual program of the project.
* Lines 23 to 26- Manage the attempt counter, which is initialised as 3. If it reaches zero, the program stops and no more user inputs are accepted.
  + B.W is used here for a long branch, because the branch is more than 2KB away from the instruction. So, the branch instruction uses 32 bits, to perform a longer jump/branch. (upto 16MB).
* Lines 28 to 34- Since we do not have the hardware to accept user input; for this project’s purposes, user input has been emulated by entering user input into ROM and then copying it to RAM for usage. In actual use, this would be skipped and user input would be accepted and stored in RAM directly.
  + This emulation is done by using a loop that copies each byte of the pin, one at a time.
  + user\_pin\_ram is a 5 byte space reserved in RAM
* Lines 37 to 58- These lines check the input pin against all the correct pins and take action based on the results of this check.
  + The correct pins are stored in an array whose memory address is initialised in R0. Every 5 bytes of memory correspond to a correct pin candidate. (4 bytes + null terminator)
  + R4 is used to check whether we’ve reached the end of the list of correct pins. The last entry in the correct\_pins array is a double word null terminator. Before beginning every iteration of the loop, we store the current 5 bytes of R0 that are being checked into R4. Then we check if these 5 bytes are the null terminator which was placed at the end of the correct\_pins array. If it is, we branch to the delay\_wrong loop because it means the user input pin is wrong.
  + R2 is the counter for the loop which checks the input pin against a candidate in the correct\_pins array. This checking is done byte by byte in the loop. We do this for each candidate by branching to skip\_candidate, if a byte doesn’t match, and then restarting the next\_candidate loop till all candidates are checked.
  + In skip\_candidate, we move the pointer in R0 forward by 5 bytes, to point to the start of the next candidate.
  + If the pin matches with a candidate, we branch to the pin\_matched section, for display of password, access message and blinking of LEDs.
* Lines 60 to 148- This section runs once the input pin matches a correct pin and is the program for display of the decrypted password in visible memory in RAM, display of a success message and LEDs blinking in a pattern to indicate correct pin entry.
* Lines 63 to 70- This loop decrypts the password. The password has been stored in memory with a simple encryption.
  + Encryption: Simple byte wise XOR with a “key” encrypts the password and byte wise XOR with the same key again decrypts it. A ^ B ^ B = A.
  + Here, the key 0xAA has been used.
* Lines 72 to 78- This is the loop to copy the success message “Access Granted” from ROM to a visible memory location in RAM, which has been initialised in R9.
* Lines 81 to 148- This is the code for the blinking of LEDs. The pattern for the LEDs is forward consecutive blinking by the LEDs followed by them blinking in a reverse order.
  + First, we set pins 28,29,31 of port1 and pins 2,3,4,5,6 of port2 to output by setting them to 1 in FIO1DIR and FIO2DIR.
  + Then we initialise FIO1SET, FIO1CLR, FIO2SET and FIO2CLR in R2, R3, R4, R5.
  + We start from pin no 28, initialising it in R6 as 0x10000000, setting it to high in FIO1SET, then we branch to a delay loop to make the on and off of consecutive LEDs visible. After the delay, we set it to low in FIO1CLR.
  + R7 is the counter to ensure we blink all the 8 LEDs.
  + We use R6 only for pin 29 and 31 by performing an LSL on the basis of counter value.
  + For port 2 forward blink, we initialise R6 as 0x04, then follow the same logic as for port1, using a delay loop, and performing LSL to blink ports 3,4,5,6.
  + In the reverse blink, we start from port 2, pin 6 as that was the last LED that blinked. We initialise it in R6 as 0x40, then run the blink loop with delay in between successive blinks. However, this time we perform an LSR based on counter value.
  + For port 1 reverse blink, we start from pin 31, initialised as 0x80000000 in R6, then use LSR and delay to make the LEDs blink.
  + After the LEDs have blinked, we branch to Stop to end execution.
* Lines 143 to 148- This is the delay loop for the LEDs blinking. We provide a delay between successive LED blinks to make them visible to us.
  + The logic of the delay loop is to busy the CPU to count for a certain amount of time, doing nothing else. We start from a large number and keep subtracting it till we reach zero, thus providing a delay between 2 instructions in our code.
* Lines 151 to 163- These lines manage the case if our input pin is wrong/doesn’t match with any of the correct pins.
  + There is no output or confirmation if input pin is wrong.
  + We limit the number of attempts at pin entry by counting them in R11. If less than 3, we branch to RESET\_HANDLER and allow input pin to pass through the program again.
  + We also force a timeout of about 10s between successive pin attempts by providing a delay with the same logic as the busy loop in the LED blinking.
  + If correct pin is not entered within 3 attempts, the program is forced to stop execution in a silent, infinite loop (Stop B Stop).
* Lines 167 to 181- These lines define the data and space we need for usage in our program
  + In the data section in RAM, with a MYDATA label, we reserve space for display of our success message and stored password. We also reserve space for the input pin.
  + In the data section in ROM, with a CONSTANTS label, we store:
    - Maximum number of attempts (here, 3)
    - user\_pin\_flash which is used for emulation of user input
    - Array of correct pins
    - The success message “Access Granted”
    - The encrypted password
* END is an assembly directive that marks the end of the file.

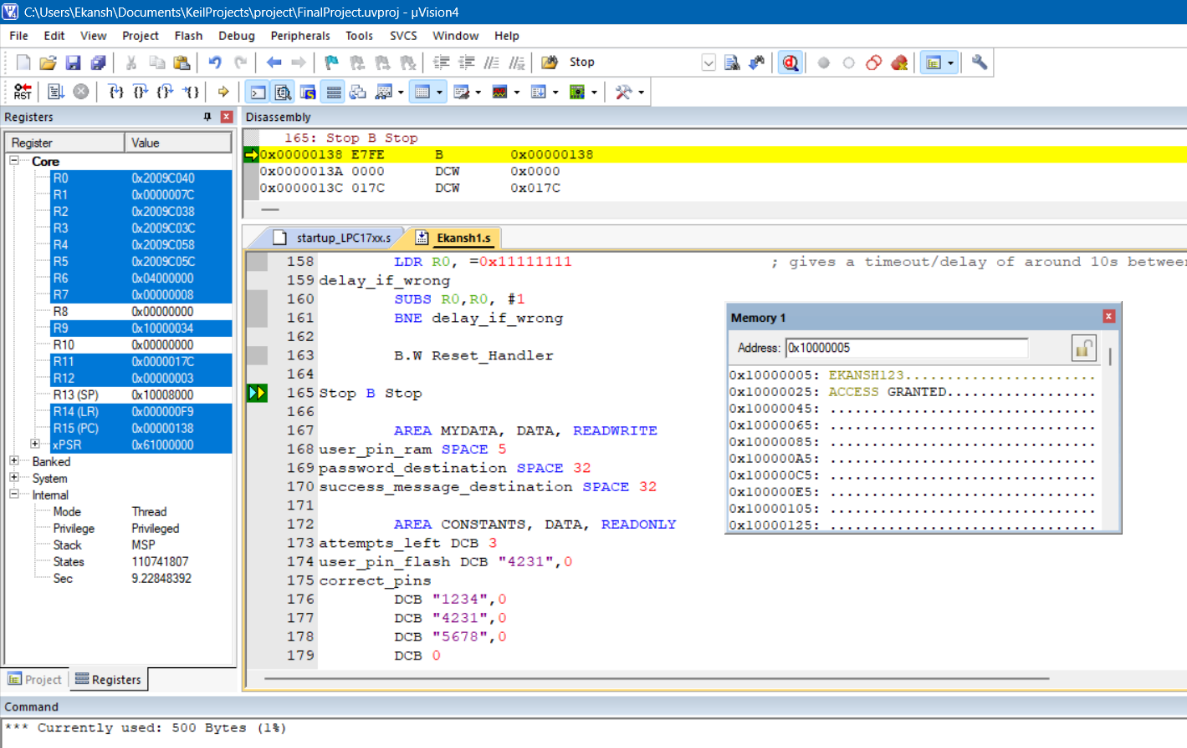
Output:

* When the input pin matches one of the correct pins, the stored password is decrypted and displayed at an exposed location in RAM. Along with the password, a success message is also displayed.

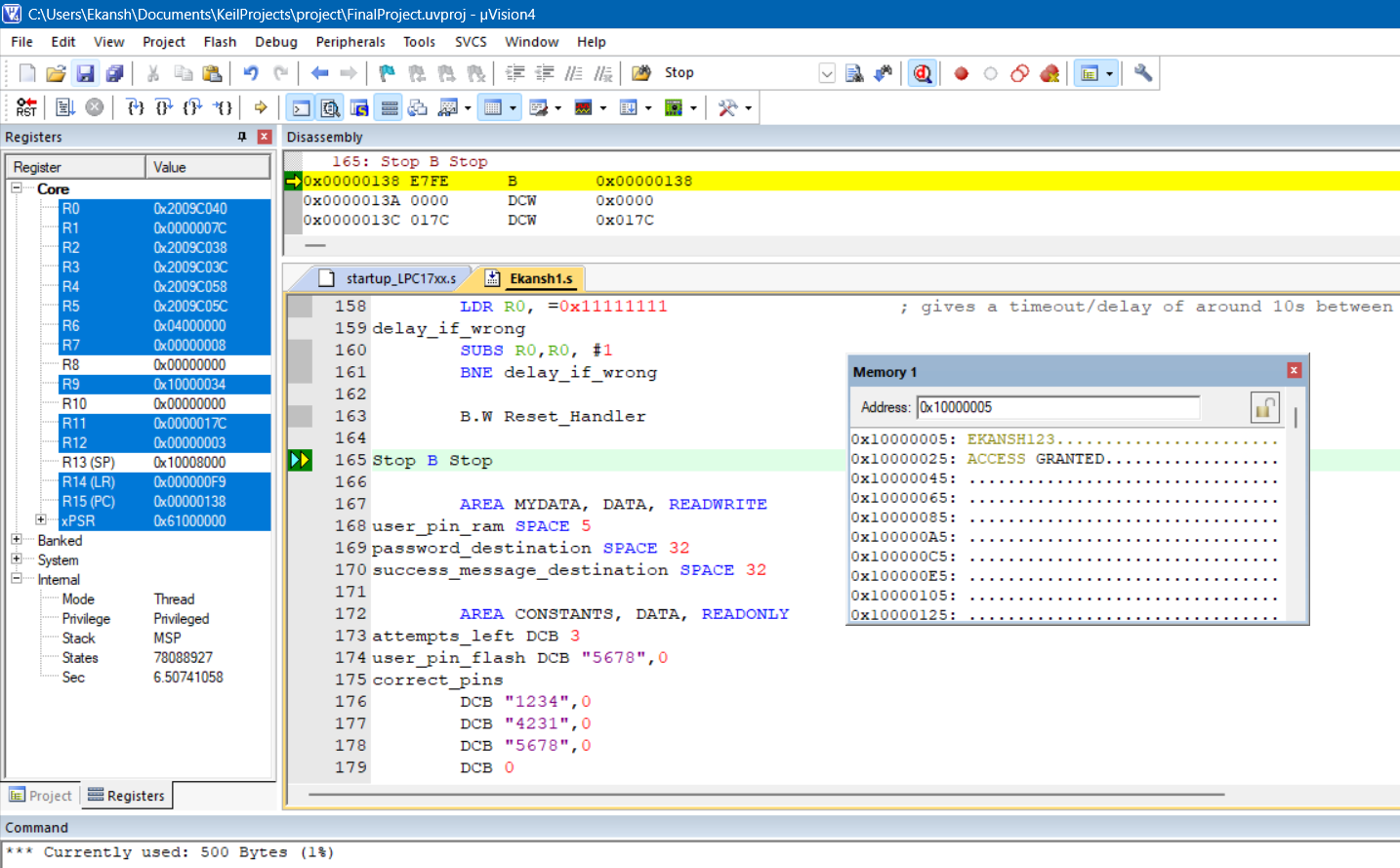
1. When input pin = 1234



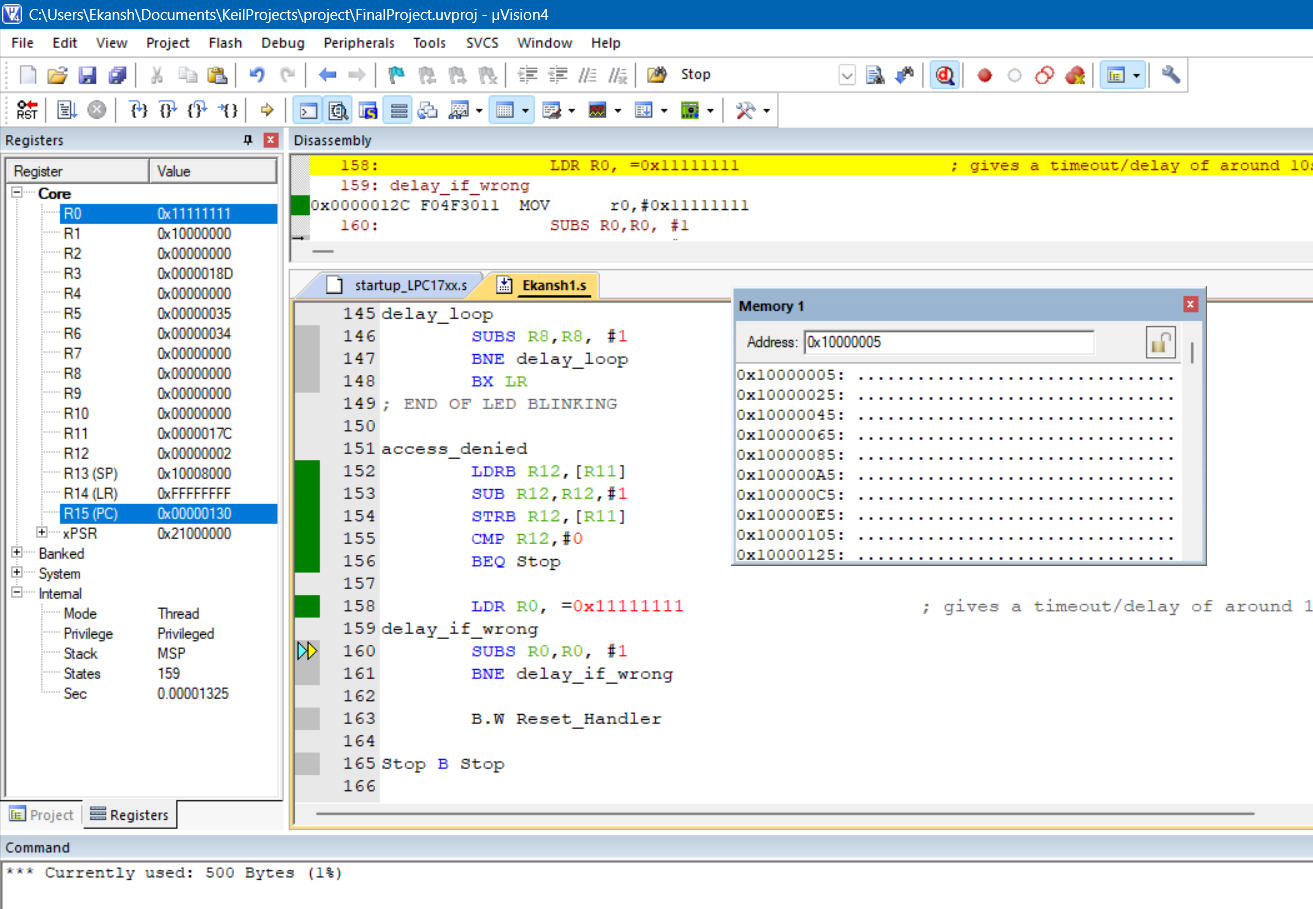
1. When input pin = 4231



1. When input pin = 5678



1. When input pin = 4232



* When the user input pin does not match with any of the correct pins, there is a timeout of nearly 10s. The user cannot input a pin for the next 10s and a silent loop runs to ensure this. This is shown in the picture.
* Silent loop indicates that there is no output to the user, the secure password is not displayed and there is no “access denied” message either, which ensures maximum security.

Demo:

(Video of LEDs blinking)

1. When input pin = 1234

<https://github.com/Ekansh345/lpc176-pin_lock-project/blob/main/videos/input-1234.mp4>

1. When input pin = 4231

<https://github.com/Ekansh345/lpc176-pin_lock-project/blob/main/videos/input-4231.mp4>

1. When input pin = 5678

<https://github.com/Ekansh345/lpc176-pin_lock-project/blob/main/videos/input-5678.mp4>

1. When input pin = 4232

There is no output when input pin is wrong.

<https://github.com/Ekansh345/lpc176-pin_lock-project/blob/main/videos/input-4232.mp4>

Note: The password has been written to memory instead of being uploaded to a register.