ISEL
DEETC
Departamento de Engenharia Electrónica e de Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

Sistema de Controlo de Acessos (*Access Control System*)

Bernardo Pereira - 50493
António Paulino - 50512

Projeto
de
Laboratório de Informática e Computadores
2022 / 2023 verão

7 de Julho de 2023

1 Introdução

Neste projeto foi implementado um sistema de controlo de acessos (*Access Control System*), que permite controlar o acesso a zonas restritas através de um número de identificação de utilizador (*User Identification Number – UIN*) e um código de acesso (*Personal Identification Number - PIN*). O sistema permite o acesso à zona restrita após a inserção correta de um par *UIN* e *PIN*. Após o acesso válido o sistema permite a entrega de uma mensagem de texto ao utilizador.

O sistema de controlo de acessos é constituído por: um teclado de 12 teclas; um ecrã *Liquid Cristal Display* (LCD) de duas linhas de 16 caracteres; um mecanismo de abertura e fecho da porta (designado por *Door Mechanism*); uma chave de manutenção (designada por M) que define se o sistema de controlo de acessos está em modo de Manutenção; e um PC responsável pelo controlo dos outros componentes e gestão do sistema. O diagrama de blocos do sistema de controlo de acessos é apresentado na Figura 1.

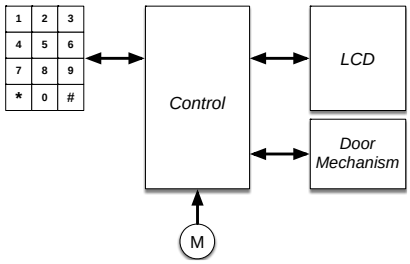


Figura 1 – Sistema de controlo de acessos (*Access Control System*)

Sobre o sistema podem-se realizar as seguintes ações em modo Acesso:

- **Acesso -** Para acesso às instalações, o utilizador deverá inserir os três dígitos correspondentes ao *UIN* seguido da inserção dos quatro dígitos numéricos do *PIN*. Se o par *UIN* e *PIN* estiver correto o sistema apresenta no LCD o nome do utilizador e a mensagem armazenada no sistema se existir, acionando a abertura da porta. A mensagem é removida do sistema caso seja premida a tecla '*' durante a apresentação desta. Todos os acessos são registados com a informação de data/hora e *UIN* num ficheiro de registos (um registo de entrada por linha), designado por *Log File*.
- **Alteração do PIN –** Esta ação é realizada se após o processo de autenticação for premida a tecla '#'. O sistema solicita ao utilizador o novo *PIN*, este deverá ser novamente introduzido de modo a ser confirmado. O novo *PIN* só é registado no sistema se as duas inserções forem idênticas.

Nota: A inserção de informação através do teclado tem o seguinte critério: se não for premida nenhuma tecla num intervalo de cinco segundos, o comando em curso é abortado; se for premida a tecla '*' e o sistema contiver dígitos, elimina todos os dígitos, se não contiver dígitos, aborta o comando em curso.

Sobre o sistema, podem-se realizar também as seguintes ações em modo Manutenção. Ao contrário das ações em modo Acesso, as ações em modo Manutenção são realizadas através do teclado e ecrã do PC. As ações disponíveis neste modo são:

- **Inserção de utilizador -** Tem como objetivo inserir um novo utilizador no sistema. O sistema atribui o primeiro *UIN* disponível, e espera que seja introduzido pelo gestor do sistema o nome e o *PIN* do utilizador. O nome tem no máximo 16 caracteres.
- **Remoção de utilizador -** Tem como objetivo remover um utilizador do sistema. O sistema espera que o gestor do sistema introduza o *UIN* e pede confirmação depois de apresentar o nome.
- **Inserir mensagem -** Permite associar uma mensagem de informação dirigida a um utilizador específico a ser exibida ao utilizador no processo de autenticação de acesso às instalações.
- **Desligar –** Permite desligar o sistema de controlo de acessos. Este termina após a confirmação do utilizador e reescreve o ficheiro com a informação dos utilizadores. Esta informação é armazenada num ficheiro de texto (com um utilizador por linha) que é carregado no início do programa e reescrito no final do programa. O sistema armazena até 1000 utilizadores, que são inseridos e suprimidos através do teclado do PC pelo gestor do sistema.
- **Listar Utilizadores –** Permite ao gestor do sistema obter uma lista com todos os utilizadores atualmente presentes no sistema.
- **Desbloquear Utilizadores** – Permite ao gestor do sistema desbloquear utilizadores.

Nota: Durante a execução das ações em modo manutenção, não podem ser realizadas ações no teclado do utilizador e no LCD consta a mensagem "Out of Service".

2 Arquitetura do sistema

O controlo (designado por *Control*) do sistema de acessos foi implementado numa solução híbrida de *hardware* e *software*, como apresentado no diagrama de blocos da Figura 2. A arquitetura é constituída por quatro módulos principais: *i*) um leitor de teclado, designado por *Keyboard Reader*; *ii*) um módulo de interface com o *LCD*, designado por *Serial LCD Controller* (*SLCDC*); *iii*) um módulo de interface com o mecanismo da porta (*Door Mechanism)*, designado por *Serial Door Controller* (*SDC*); e *iv*) um módulo de controlo, designado por *Control*. Os módulos *i*), *ii*) e *iii*) foram implementados em *hardware* e o módulo de controlo foi implementado em *software* a executar num PC.

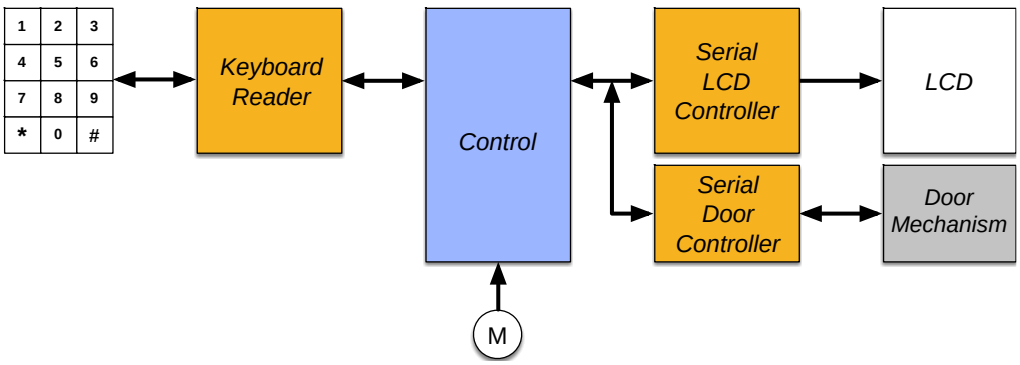


Figura 2 – Arquitetura do sistema que implementa o Sistema de Controlo de Acessos (*Access Control System*)

O módulo *Keyboard Reader* é responsável pela descodificação do teclado matricial de 12 teclas, determinando qual a tecla pressionada e disponibilizando o código desta em quatro bits ao *Control*, caso este esteja disponível para o receber. Caso este não esteja disponível para o receber imediatamente, o código da tecla é armazenado até ao limite de nove códigos. O *Control* processa e envia para o *SLCDC* a informação contendo os dados a apresentar no *LCD*. A informação para o mecanismo da porta é enviada através do *SDC*. Por razões de ordem física, e por forma a minimizar o número de sinais de interligação, a comunicação entre o módulo *Control* e os módulos *SLCDC e SDC* é realizada através de um protocolo série.

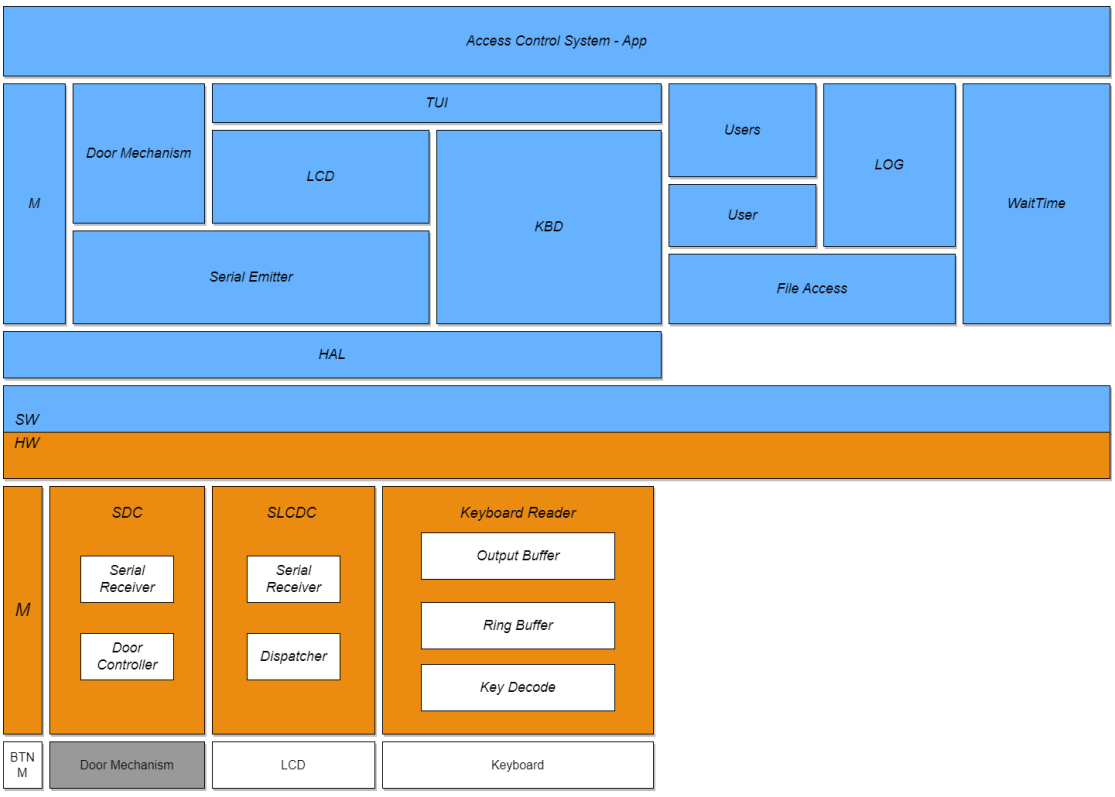


Figura 3 – Diagrama lógico do Sistema de Controlo de Acessos (*Access Control System*)

3 Control

O módulo *Control* foi implementado em software com recurso à linguagem *Kotlin*, seguindo a arquitetura lógica apresentada na figura 3 (SW).
HAL e KBD são descritos na documentação do módulo *Keyboard Reader*, enquanto *Serial Emitter*, *LCD* e *Door Mechanism* são descritos nas documentações dos módulos SDC e SLCDC. Todos os módulos se encontram no anexo A.

3.1 TUI

A classe TUI é responsável pela interação e exibição do texto no LCD. Providencia funcionalidade para escrever no LCD e ler inputs do utilizador, com seleção de linhas e tipo de alinhamento.

O enumerado `LINES` define as linhas presentes no LCD.

O enumerado `ALIGN` define o alinhamento do texto exibido no LCD.

O enumerado `ENTRY` define um tipo específico de informação a ser exibido no LCD e o seu tamanho.

A constante `KBDTIMEOUT` representa o tempo de espera em milissegundos para o utilizador premir uma tecla.

A constante `TIMEOUTCODE` representa um código de erro para quando a operação de leitura de input atingiu o valor `KBDTIMEOUT` de espera para o utilizador inserir o valor.

A constante `ABORTCODE` representa um código de erro para quando a operação de leitura é abortada pelo utilizador.

A lista `CMD_ABORT_CODES` representa uma lista de códigos de erro com os valores `TIMEOUTCODE` e `ABORTCODE`

A função `getColOffset(align: ALIGN, text:String)` calcula a posição inicial de escrita do cursor do LCD com base no tamanho do texto a ser inserido e no alinhamento do mesmo.

A função `clearEntryDigits(line: Int, col:Int, len: Int)` limpa o campo de entrada da linha especificada por `line`, começando na coluna `col`. Reescreve no LCD o campo de entrada, sendo que o seu tamanho é especificado por `len`.

A função `write(text: String, col: Int, line: Lines)` escreve o texto especificado por `text` no LCD na linha especificada por `line` começando na coluna especificada por `col`.

A função `read(line: Lines, startcol: Int, entry: ENTRY)` Lê e retorna os valores inseridos pelo utilizador no teclado, escrevendo os mesmos no LCD na linha especificada por `line` e começando na coluna especificada por `col`. Quando a quantidade de valores inseridos corresponder ao tamanho da entrada especificada por `entry`, a função retorna os valores escritos.

A função `clearline(line:LINES)` limpa a linha do LCD.

A função `writeLine(str: String, align: ALIGN, line: LINES, clear: Boolean = true)` Limpa a linha e escreve a `str` indicado na linha `line`, com o alinhamento especificado por `align`. O parâmetro `clear` indica se a linha deve ser limpa antes de escrever no LCD, por definição limpa, mas pode ser definido pelo programador.

A função `writeLines (str1: String, align1: ALIGN, str2: String, align2: ALIGN, clear: Boolean = true)` escreve 2 linhas no LCD chamando a função `writeLine` 2 vezes. A string `str1` é escrita na primeira linha e a string

`str2` é escrita na segunda linha. O parâmetro `clear` indica se devem ser limpas as duas linhas antes de escrever no LCD. Caso seja necessário limpar uma linha e não outra, deve ser utilizada a função `writeLine` duas vezes.

A função `query(text: String, align : ALIGN, line: LINES, entry: ENTRY)` Consulta o utilizador para que este insira um valor no LCD com o tamanho especificado pelo parâmetro `entry`.

3.2 File Access

A classe File Access providencia funcionalidade para leitura e escrita em ficheiros.

A função `createReader(fileName : String)` cria um leitor de ficheiro no ficheiro especificado por `fileName`.

A função `createWriter(fileName: String)` cria um escritor de ficheiro no ficheiro especificado por `fileName`

A função `inFromFile(fileName: String)` lê as linhas do ficheiro especificado e retorna uma lista de `Strings` em que cada elemento é uma linha.

A função `outToFile(lines: List<String>, fileName : String)` escreve no ficheiro especificado por `fileName` as linhas especificadas por `lines`.

A função `appendFile(line : String, fileName : String)` adiciona ao ficheiro especificado por `fileName` a linha especificada por `line`.

3.3 LOG

A classe LOG é responsável por registar num ficheiro sempre que há uma autenticação o utilizador e quando entrou.

A função `add(uin : Int)` é responsável por adicionar uma nova log ao ficheiro, sendo que o utilizador a registar é especificado pelo seu `UIN`.

A função `getDate(display : Boolean)` retorna a data corrente do sistema, sendo que o parâmetro display especifica o tipo de formatação para a data, se `display` for `true` retorna a formatação de display ao LCD, se for `false` retorna a formatação de escrita no ficheiro LOG.

3.4 User

A classe User implementa a classe de utilizador do sistema, com o seu `UIN`, `PIN` nome de utilizador, mensagem, e flag de controlo de bloqueio de autenticação.

3.5 Users

A classe Users providencia funcionalidade para gerir múltiplos utilizadores no sistema.

A constante `SIZE` representa a quantidade máxima de utilizadores no sistema.
A variável `userlist` representa o `array` em que os utilizadores são armazenados durante o tempo de execução do programa.
A constante `key` representa a chave para codificar e descodificar o pin do utilizador, para este poder ser armazenado.

A função `init(fileName: String)` inicializa a `userlist` com os utilizadores presentes no ficheiro `fileName`.

A função `Int.encode()` é responsável por codificar um número inteiro com a chave key.

A função `addUser(username : String, pin: Int)` adiciona a userlist um novo utilizador com o nome de utilizador e pin especificados pelos parâmetros username e pin. Atribui ao novo utilizador o primeiro UIN disponível.

A função `removeUser(uin : Int)` remove o utilizador especificado por `uin` da `userlist`.

A função `getUser(uin : Int)` retorna o utilizador da `userlist` especificado por `uin`

A função `blockUser(uin : Int)` bloqueia o utilizador especificado pelo `uin` de se autenticar no sistema.

A função `unlockUser(uin : Int)` desbloqueia o utilizador especificado pelo `uin` de se autenticar no sistema.

A função `isValidLogin(user: User, pin : Int)` verifica se o pin especificado por `pin` é o `pin` do utilizador especificado por `user`.

A função `close(filename: String)` escreve no ficheiro especificado por `filename` os novos utilizadores.

3.6 M

A Classe M implementa o modo de manutenção para o sistema.

A constante `MAINTENANCEMASK` representa a máscara do bit M da Usb Port
A variável `maintenance` indica se o sistema está em modo de manutenção.

A função `init()` inicializa o modo de manutenção ao verificar se a flag `maintenance` está ativa. Se o estiver, entra no modo de manutenção.

A função `run()` é responsável por correr o modo de manutenção enquanto a flag `maintenance` estiver ativa e processa os inputs de comando dados pelo utilizador.

A função `printHelp()` é responsável por colocar no standard output todos os comandos disponíveis no modo de manutenção e as suas descrições

3.7 Access Control System – App

O *Access Control System* é responsável pelo controlo do acesso ao sistema por meios de um UIN e um PIN. O sistema permite o acesso a zonas restritas após ser inserido um par UIN-PIN correto. Após um acesso autorizado, entrega uma mensagem com destino ao utilizador se esta existir. Também após um acesso autorizado, o utilizador tem a opção de alterar o seu PIN, ou de limpar a mensagem que lhe foi entregue.

A constante `CMD_WAIT_TIME` representa o tempo de espera em milissegundos entre diferentes comandos do sistema.
A constante `DOOR_OPEN_SPEED` representa a velocidade de abertura da porta.
A constante `DOOR_CLOSE_SPEED` representa a velocidade de fecho da porta.
A variável `on` indica se o sistema está ligado ou desligado.

A função `init()` inicializa todos os componentes necessários para o funcionamento do sistema, tendo em conta a hierarquia representada na figura 3, e ativa a flag `on`.

A função `run()` é responsável por correr o sistema enquanto a flag `on` está ativa. Processa a validação do utilizador e os comandos de mudança de PIN, limpeza de mensagem e abertura/fecho de porta.

A função `validateUser(user : User)` é responsável por validar o PIN dado pelo utilizador. O utilizador é dado 3 tentativas para validar o seu PIN. Retorna true quando o PIN dado pelo utilizador corresponde ao do utilizador especificado por user.

A função `changePin(user : User)` é responsável pela alteração do PIN do utilizador após este se ter autenticado. O sistema inicialmente pede confirmação para alteração do PIN, e caso haja confirmação pergunta ao utilizador qual é o novo PIN duas vezes. Caso as duas respostas sejam iguais, o PIN é alterado.

A função `clearMessage(user : User)` é responsável pela limpeza da mensagem atribuída ao utilizador após este se ter autenticado. O sistema inicialmente pede confirmação para limpeza da mensagem, e caso haja confirmação a mensagem é apagada.

A função `hello(user : User)` é responsável por saudar o utilizador após este se ter autenticado, mostrando a mensagem que lhe foi atribuída caso esta exista.

A função `openCloseDoor(user : User)` é responsável por abrir a porta ao utilizador após este se ter autenticado, permitindo o acesso à zona restrita. Após a porta ser aberta, é fechada novamente para impedir acesso a utilizadores não autorizados.

A função `printUsersCommand()` escreve no standard output todos os utilizadores atualmente presentes no sistema.

A função `addUserCommand()` é responsável por adicionar um novo utilizador ao sistema, sendo que processa os inputs de `username` e `pin` pelo gestor do sistema.

A função `removeUserCommand()` é responsável por remover um utilizador do sistema, sendo que o processa o input de `uin` pelo gestor do sistema.

A função `addMessageCommand()` é responsável por entregar uma mensagem a um utilizador. Esta é apresentada depois de o utilizador se ter autenticado corretamente.

A função `unlockUserCommand()`é responsável por desbloquear um utilizador após este ter sido bloqueado por falhar na autenticação 5 vezes seguidas.

A função `closeCommand()` permite ao gestor do sistema encerrar o sistema, ao atualizar a lista de utilizadores. Após execução deste comando, o programa deixa de ser executado assim que o gestor do sistema sair do modo de manutenção.

A função `getIntEntry(entry : ENTRY)` é responsável por processar inputs do gestor do sistema do tipo Integer, como por exemplo o UIN ou o PIN.

A função `getStrEntry(entry : ENTRY)` é responsável por processar inputs do gestor do sistema do tipo String, como por exemplo a mensagem ou o nome de utilizador.

A função `close()` desliga o sistema ao atualizar o ficheiro onde são armazenados os utilizadores, e a colocar a flag `on` a false.

4. Conclusões

Neste projeto, foi implementado um sistema de controlo de acessos que utiliza um número de identificação de utilizador (UIN) e um código de acesso (PIN) para permitir ou negar o acesso a zonas restritas. O sistema é composto por um teclado, um ecrã LCD, um mecanismo de abertura e fecho da porta, uma chave de manutenção e um PC responsável pelo controlo e gestão do sistema.

Durante o modo de acesso, os utilizadores podem inserir o par UIN PIN correto para aceder às instalações. O sistema verifica a autenticação e, se for bem-sucedida, exibe o nome do utilizador no LCD e entrega uma mensagem de texto, se existir. Também são registados os acessos no Log File.

Além disso, o sistema permite a alteração do PIN, caso o utilizador pressione a tecla '#' após a autenticação, sendo que o novo PIN só é registado se as duas inserções forem idênticas.

No modo de manutenção, o gestor do sistema pode realizar várias ações, como inserir ou remover utilizadores, associar mensagens de informação a utilizadores específicos, desligar o sistema e obter uma lista de utilizadores presentes no sistema.

A arquitetura do sistema é dividida em quatro módulos principais: Keyboard Reader, Serial LCD Controller (SLCDC), Serial Door Controller (SDC) e Control. O Keyboard Reader decodifica as teclas pressionadas e envia os códigos para o Control. O Control processa a informação e envia dados para o SLCDC e SDC. A comunicação entre os módulos é realizada através de um protocolo série.

Nesta implementação, foram utilizados na placa FPGA DE10-LITE da Intel 437/49,670 elementos lógicos(0,88%), 215 registos e 83/360 pinos (23%).

Em conclusão, o sistema de controlo de acessos implementado demonstra a capacidade de autenticar utilizadores e controlar o acesso a zonas restritas. A integração de hardware e software permite uma interação eficiente entre o utilizador, o sistema e os dispositivos de interface. O sistema fornece também recursos de manutenção, garantindo a gestão adequada dos utilizadores e das mensagens associadas.

A. Módulos implementados

SLCDC

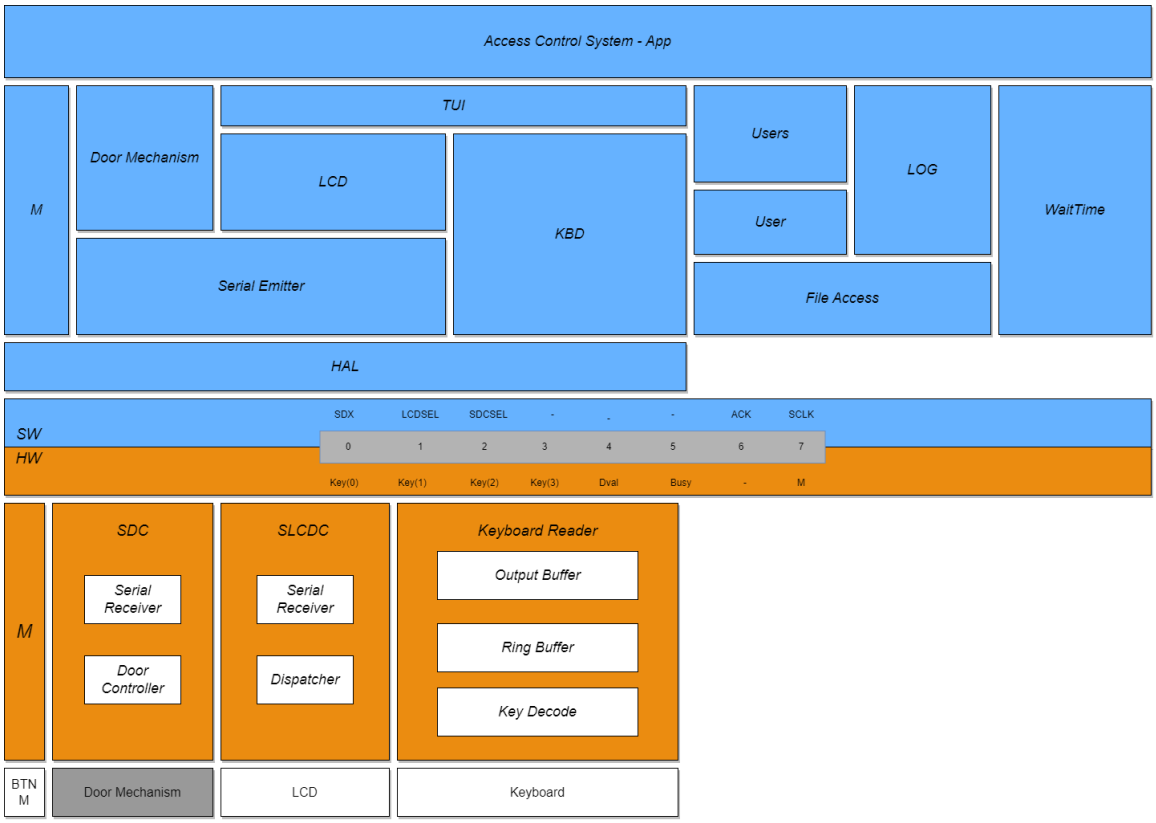
SDC

Keyboard Reader

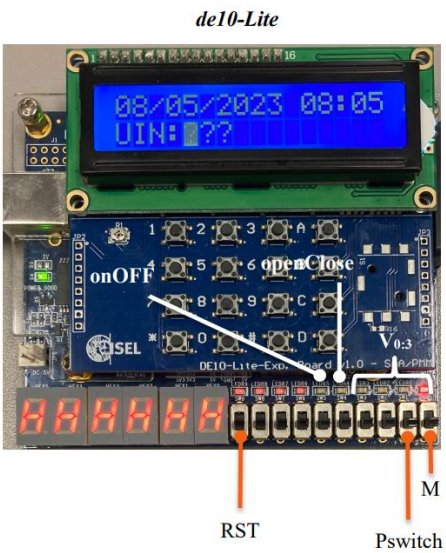
B. Interligações entre o HW e SW

As ligações entre o hardware e software foram realizadas de acordo com a seguinte tabela e o seguinte esquema:

UsbPort	0	1	2	3	4	5	6	7
Input Port	Key(0)	Key(1)	Key(2)	Key(3)	DVal	Busy	-	M
Output Port	SDX	LCDSEL	SDCSEL	-	-	-	ACK	SCLK



Os LEDS e switches foram mapeados da seguinte forma:



C. Código *Kotlin - TUI*

```
import LCD.COLS
import TUI.ABORTCODE
import TUI.CMD_ABORT_CODES
import TUI.KBDTIMEOUT
import TUI.TIMEOUTCODE

/**
 * 22/5/2023
 *
 * Text User Interface (TUI) for displaying and interacting with text on an LCD
screen.
 * Provides functionality for writing and querying text on the LCD with line
selection and alignment type.
 *
 * @property KBDTIMEOUT the max amount of time to wait for a user input. Used
when querying for information.
 * @property TIMEOUTCODE value to return when [KBDTIMEOUT] is reached during a
query.
 * @property ABORTCODE value to return when a query operation is aborted.
 * @property CMD_ABORT_CODES list of possible abort codes for query operations.
 *
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 * @see LCD
 */
object TUI {


    /**
     * The LCD lines.
     *
     */
    enum class LINES { First, Second }

    /**
     * The text alignment type. None means keep writing without repositioning
the cursor
     */
    enum class ALIGN { None, Left, Center, Right }

    /**
     * Refers to a specific type of information to display on the LCD and its
length.
     *
     * @see User
     */
    enum class ENTRY(val len: Int) { UIN(3), PIN(4), MSG(16), USERNAME(16) }

    const val KBDTIMEOUT = 5000.toLong()

    const val TIMEOUTCODE = -1

    const val ABORTCODE = -2

    val CMD_ABORT_CODES = listOf(ABORTCODE, TIMEOUTCODE)
```

```
    /**
     * Calculates the LCD start column position based on text length and
alignment type.
     * @param alignment the alignment type
     * @param text the text to write on the LCD
     *
     * @return the column position
     */
    private fun getColOffset(align: ALIGN, text: String): Int {

        return when (align) {
            ALIGN.None -> 0
            ALIGN.Center -> (COLS - text.length) / 2 + 1
            ALIGN.Left -> 1
            ALIGN.Right -> COLS - text.length + 1
        }

    }


    /**
     * Rewrites the input field on the LCD when querying. Used for clearing the
input field.
     * @param line The line to re-write on
     * @param column The first column to re-write
     * @param len The number of columns to re-write
     */
    private fun clearEntryDigits(line: Int, col: Int, len: Int) {

        LCD.cursor(line, col) //Position the cursor on the first col of the
input field

        for (i in 0 until len) {
            LCD.write("_") //writes '_' on the input field of the LCD.
        }

        LCD.cursor(line, col) //Reposition the cursor on the first col of the
input field for user input

    }


    /**
     * Writes the given text on the LCD.
     * @param text The text to write
     * @param col The starting col position
     * @param line the line to write on
     */
    private fun write(text: String, col: Int, line: LINES) {

        if (col > 0) LCD.cursor(line.ordinal + 1, col) //reposition the cursor
if the type of alignment is not none
        LCD.write(text)

    }
```

```
    /**
     * Reads user input from the LCD.
     * @param line The line where the input is on
     * @param col The starting col position of the input
     * @param entry The type of entry (UIN/PIN) to determine length.
     * @return The user input as integer, [TIMEOUTCODE] if the timeout is
reached, or [ABORTCODE] if the command is aborted.
     */
    private fun read(line: LINES, startcol: Int, entry: ENTRY): Int {

        val linepos = line.ordinal + 1

        var userInput = 0 //stores the user input
        var inputCount = 0

        LCD.cursor(linepos, startcol)   //position the cursor at the start of
the input

        while (inputCount < entry.len) {   //get the user input

            val key = KBD.waitKey(KBDTIMEOUT)

            if (key == KBD.NONE.toChar()) return TIMEOUTCODE  //abort if no key
is pressed

            else if (key == '*') {

                if (userInput == 0) return ABORTCODE    //abort if the input
field is empty

                clearEntryDigits(linepos, startcol, entry.len) //if the input
field is not empty clear the input field
                userInput = 0
                inputCount = 0

            } else if (key in '0'..'9') {
                inputCount++
                userInput = userInput * 10 + (key - '0')
                if (entry == ENTRY.PIN) LCD.write('*') //show the user input on
the input field of the LCD. If the entry is PIN don't show it.
                else LCD.write(key)

            }

        }

        return userInput
    }

    /**
     * Clears a line on the LCD
     * @param line the line to clear
     */
    private fun clearline(line: LINES) {
        write("                ", 1, line)
        LCD.cursor(line.ordinal + 1, 1)
```

```
    }


    /**
     * Writes a line on the LCD. Gets the column offset needed to align the text
     * and writes it on the LCD using [write]. Clears the line on the LCD before
writing by default.
     * @param str the string to be written
     * @param align the alignment type
     * @param line the line to write on
     * @param clear whether to clear the line before writing. Default is yes but
can be user set.
     *
     */
    fun writeLine(str: String, align: ALIGN, line: LINES, clear: Boolean = true)
{
        if (clear) clearline(line)
        val coloffset = getColOffset(align, str)
        write(str, coloffset, line)
    }

    /**
     * Writes two lines on the LCD by calling [writeLine] two times.
     * @param str1 the string to write on the first line
     * @param str2 the string to write on the second line
     * @param align1 the alignment type for the first line
     * @param align2 the alignment type for the second line
     * @param clear whether to clear both lines before writing. Default is yes
but can be user set.
     * If one line should be cleared and the other should not, use [writeLine]
instead.
     */
    fun writeLines(str1: String, align1: ALIGN, str2: String, align2: ALIGN,
clear: Boolean = true) {

        if (clear) LCD.clear()

        writeLine(str1, align1, LINES.First, false)
        writeLine(str2, align2, LINES.Second, false)

    }

    /**
     * Queries the user for an input using the LCD. Calls [read] to get the user
input.
     * @param text the text prompt to be displayed
     * @param align the alignment of the text prompt
     * @param line the line to display the prompt on
     * @param entry the type of input entry
     * @return The user input as integer, [TIMEOUTCODE] if the timeout is
reached, or [ABORTCODE] if the command is aborted.
     */
    fun query(text: String, align: ALIGN, line: LINES, entry: ENTRY): Int {

        var entrytext = text
        for (i in 0 until entry.len) {
            entrytext += '_'
```

```
        }   //creates the input field of the given entry

        val coloffset = getColOffset(align, entrytext)
        clearline(line)
        write(entrytext, coloffset, line)


        return read(line, text.length + coloffset, entry) //text length + col
offset = first col of the input field.

    }

    fun getKey() = KBD.waitKey(KBDTIMEOUT / 2)


}
```

D. Código *Kotlin - FileAccess*

```
import java.io.BufferedReader
import java.io.FileReader
import java.io.FileWriter
import java.io.PrintWriter


/**
 * 22/5/2023
 *
 * Provides functionality for reading and writing to a file.
 *
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 */
object FileAccess {

    /**
     * Creates a BufferedReader for reading from the specified file
     * @param fileName The name of the file to read from
     * @return A BufferedReader associated with the file.
     */
    private fun createReader(fileName: String): BufferedReader =
BufferedReader(FileReader(fileName))

    /**
     * Creates a PrintWriter for writing to the specified file
     * @param fileName The name of the file to write to
     * @return A PrintWriter associated with the file.
     */
    private fun createWriter(fileName: String): PrintWriter =
PrintWriter(fileName)


    /**
     * Creates a FileWriter for appending to the specified file
     * @param fileName The name of the file to append to
     * @return A FileWriter associated with the file.
     */

    private fun createWriterAppend(fileName: String): FileWriter =
FileWriter(fileName, true)


    /**
     * Writes the provided lines to the specified file
     * @param lines The lines to write
     * @param fileName The name of the file to write to
     */
    fun outToFile(lines: List<String>, fileName: String) {
        val outFile = createWriter(fileName)

        for (line in lines) {
            outFile.println(line)
        }

        outFile.close()
```

```
    }

    /**
     * Reads all the lines from the specified file
     * @param fileName The name of the file to read from
     * @return A list of strings with all the lines from the file.
     */
    fun inFromFile(fileName: String): List<String> {
        val inFile = createReader(fileName)
        val lines = inFile.readLines()
        inFile.close()
        return lines
    }

    /**
     * Appends the provided line to the specified file
     * @param fileName The name of the file to append to
     * @param line The line to append
     *
     */
    fun appendFile(line: String, fileName: String) {
        val outFile = createWriterAppend(fileName)
        outFile.write(line)
        outFile.close()
    }

}
```

E. Código *Kotlin - User*

```
/**
 * 22/5/2023
 *
 * Represents a user in the system.
 *
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 *
 *
 * @property UIN The User Identification Number.
 * @property PIN The Personal Identification Number of the user.
 * @property username The username of the user.
 * @property message A message stored for the user to be displayed when
authenticated.
 *
 *
 *
 * @see Users
 */
data class User(val UIN: Int, var pin: Int, var username: String, var message:
String, var blocked: Boolean)
```

F. Código *Kotlin - Users*

```
import Users.SIZE

/**
 * 22/5/2023
 *
 * Collection of user data, provides functionality for managing users.
 *
 * @property SIZE the max amount of users
 * @see User
 * @see FileAccess
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 */
object Users {
    /**
     * The amount of users that can be stored
     */
    const val SIZE = 1000

    /**
     * The array where users are stored during program run time
     */
    private var userlist = arrayOfNulls<User>(SIZE)

    /**
     * The key for pin encryption
     */
    private const val key = 0b0100101100111011


    /**
     * Initializes the list of users.
     * Reads from the file where users are stored.
     * @param fileName the file where users are stored
     */
    fun init(fileName: String) {
        val users = FileAccess.inFromFile(fileName)
        for (user in users) {

            val userargs = user.split(";") //User parameters are separated by
';'
            val (uin, pin, username, message, blocked) = userargs
            val uinInt = uin.toInt()
            val pinInt = pin.toInt()
            userlist[uinInt] = User(uinInt, pinInt, username, message,
blocked.toBoolean())
        }
    }


    /**
     * Encodes the user PIN so it can be stored.
     */
    private fun Int.encode(): Int = (this.xor(key) * key)

    /**
```

```
     * Adds a user to the current user list by
     * iterating through the user list and atributting the first available
     * UIN to the new user.
     *
     * @param username Username of the new user.
     * @param pin PIN of the new user.
     *
     * @return the new user's UIN or -1 if the user list is currently full.
     */
    fun addUser(username: String, pin: Int): Int {
        for (uin in userlist.indices) {
            if (userlist[uin] == null) {
                userlist[uin] = User(uin, pin.encode(), username, "", false)
                return uin
            }
        }
        return -1
    }

    /**
     * Removes a user from the current user list.
     * @param uin The UIN of the user to remove.
     */
    fun removeUser(uin: Int) {
        userlist[uin] = null
    }

    /**
     * Adds a message to the specified user.
     * @param uin The UIN of the user to message.
     * @return the UIN of the user if the message was added successfully or -1
if the user does not exist.
     */
    fun setMsg(message: String, uin: Int): Int {
        if (userlist[uin] == null) return -1
        userlist[uin] = userlist[uin]!!.copy(message = message)
        return uin
    }

    /**
     * Gets a user from the user list.
     * @param uin The UIN of the user
     * @return The user or null if the user does not exist
     */
    fun getUser(uin: Int): User? = userlist[uin]

    /**
     * Blocks a user
     * @param user The user to block
     * @return -1 if the user does not exist or the uin of the user if the
change was successful
     */
    fun blockUser(uin: Int) : Int {
        if (userlist[uin] == null) return -1
        userlist[uin] = userlist[uin]!!.copy(blocked = true)
        return uin
    }
```

```
    /**
     * Unlocks a user
     * @param user The user to unlock
     * @return -1 if the user does not exist or the uin of the user if the
change was successful
     */
    fun unlockUser(uin: Int) : Int {
        if (userlist[uin] == null) return -1
        userlist[uin] = userlist[uin]!!.copy(blocked = false)
        return uin
    }

    /**
     * Checks if the given pin is the correct pin for the given user
     * @param user The user being validated
     * @param pin The pin to compare with the user pin
     * @return true if the given pin is the user pin, false otherwise
     */
    fun isValidLogin(user: User, pin: Int): Boolean = user.pin == pin.encode()


    /**
     * Stores the user list that was altered during program run time in the
specified file.
     * One user per line, with each parameter separated by a semicolon.
     * @param fileName the name of the file to store users in.
     */

    fun close(filename: String) {
        val usersToStore = userlist.filterNotNull().map {
"${it.UIN};${it.pin};${it.username};${it.message};${it.blocked}" }
        FileAccess.outToFile(usersToStore, filename)
    }

}
```

G. Código *Kotlin - Log*

```
import java.text.SimpleDateFormat
import java.util.*

/**
 * 22/5/2023
 *
 * Registers the authenticated user and their time of entry to a file.
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 * @see Users
 * @see FileAccess
 */
object LogFile {


    /**
     * Adds a log entry to the log file
     * @param uin the UIN (Universal Identification Number) of the authenticated
user
     */
    fun add(uin: Int) {
        val time = getDate(false)
        val user = Users.getUser(uin)!!
        FileAccess.appendFile("$time -> ${user.UIN}, ${user.username} \n",
"LOG.txt")
    }

    /**
     * Gets the current system date.
     *
     * @param display Whether to return the date to display on the LCD (true) or
the date to write to the log file (false).
     * @return The formatted date to display on the LCD or write to the log
file.
     */

    fun getDate(display: Boolean): String {
        val date = Calendar.getInstance().time
        val dateformat = if (display) SimpleDateFormat("MMM dd HH:mm")
        else SimpleDateFormat("dd/MM/yyyy HH:mm:ss")
        return dateformat.format(date)
    }
}
```

H. Código *Kotlin* da classe *M*

```
fun main() {
    M.run()
}


/**
 * 22/5/2023
 *
 * Maintenance mode for the [AccessControlSystem].
 * Handles maintenance operations and commands.
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 *
 * @see AccessControlSystem
 * @see Users
 * @see User
 * @see TUI
 */
object M {
    /**
     * Bit mask for the M flag on the USB input port.
     */
    private const val MAINTENANCEMASK = 0b10000000

    /**
     * Stores the current M flag value, if true, the maintenance mode runs, if
false, it stops.
     */
    private var maintenance = false

    /**
     * initializes the maintenance mode if the system is in maintenance mode,
indicated by the M input.
     */
    fun init() {
        maintenance = HAL.isBit(MAINTENANCEMASK)
        if (maintenance) run()
    }

    /**
     * Executes the maintenance mode commands.
     * Displays the maintenance mode interface on the LCD.
     * Available commands: HELP, ADDUSER, REMOVEUSER, ADDMSG, CLOSE.
     */
    fun run() {
        AccessControlSystem.enterMaintenance()
        println("Maintenance mode. Write help for a list of commands")
        while (maintenance) {
            print("Maintenance> ")
            val command = readln().trim().lowercase()

            if (command.isNotEmpty()) {
                when (command) {
                    "help" -> printHelp()
                    "adduser" -> AccessControlSystem.addUserCommand()
                    "removeuser" -> AccessControlSystem.removeUserCommand()
```

```
                    "addmsg" -> AccessControlSystem.addMessageCommand()
                    "close" -> AccessControlSystem.closeCommand()
                    "listusers" -> AccessControlSystem.printUsersCommand()
                    "unlockuser" -> AccessControlSystem.unlockUserCommand()
                    else -> println("Invalid command")
                }
            }
            maintenance = HAL.isBit(MAINTENANCEMASK)
        }
        println("Exiting maintenance mode...")
    }
}


/**
 * Prints the help information for available maintenance commands.
 */
fun printHelp() {
    println("ADDUSER                                                    | Adds
a user to the system.")
    println("REMOVEUSER                                                 |
Removes a user from the system.")
    println("ADDMSG                                                     | Adds
a message to a specified user.")
    println("CLOSE                                                      |
Updates the system, allowing it to be shut down.")
    println("LISTUSERS                                                  |
Prints the system users.")
    println("UNLOCKUSER                                                 |
Unlocks a blocked user.")
}
```

I. Código *Kotlin - Access Control System - App*

```
import AccessControlSystem.CMD_WAIT_TIME
import AccessControlSystem.on
import TUI.ALIGN
import TUI.CMD_ABORT_CODES
import TUI.ENTRY
import TUI.KBDTIMEOUT
import TUI.LINES

fun main() {
    AccessControlSystem.init()
    AccessControlSystem.run()
}


/**
 *
 * 22/5/2023
 *
 *
 * Responsible for controlling access to restrict zones by means of a User
Identification Number (UIN)
 * and a Personal Identification Number (PIN).
 *
 * The system allows access to restrict zone after correct input of a UIN and
PIN pair. After a valid access,
 * the system allows the delivery of a text message addressed to the user.
 *
 * The AccessControlSystem is composed of a [LCD] for display, a [KBD] for
reading user inputs,  a [DoorMechanism] for opening and closing the door,
 * as well as [M] for maintenance operations.
 * @property on Indicates whether the Access Control System is on or off. To
turn off the system set to false.
 * @see HAL
 * @see SerialEmitter
 * @see LCD
 * @see TUI
 * @see KBD
 * @see LogFile
 * @see M
 * @see User
 * @see Users
 * @see DoorMechanism
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 *

 */
object AccessControlSystem {

    /**
     * The time to wait between successive commands from the Access Control
System.
     */
    private const val CMD_WAIT_TIME = 2000

    /**
```

```
     * The speed at which the door is opened after user authorized access.
     */
    const val DOOR_OPEN_SPEED = 0b1000

    /**
     * The speed at which the door is closed after user authorized access.
     */
    const val DOOR_CLOSE_SPEED = 0b0010


    /**
     * The max amount of login attempts before user gets locked
     */
    private const val MAX_ATTEMPTS = 5


    var on = false

    /**
     * Initializes the [AccessControlSystem] by initializing all the required
components and loading user data.
     */
    fun init() {
        HAL.init()
        KBD.init()
        SerialEmitter.init()
        LCD.init()
        DoorMechanism.init()
        Users.init("USERS.txt")
        on = true
        M.init()
    }


    /**
     * Runs the [AccessControlSystem].
     *
     * Validates the user and processes the Open/Close door commands, as well as
the Change PIN and Clear Msg commands.
     *
     */
    fun run() {

        while (on) { // run if on

            TUI.writeLine(LogFile.getDate(true), ALIGN.Center, LINES.First)

            val uin = TUI.query("UIN:", ALIGN.Center, LINES.Second, ENTRY.UIN)

            if (uin !in CMD_ABORT_CODES) { // If there wasn't a timeout or input
abort by the user

                val user = Users.getUser(uin)

                if (user != null) { // If the user exists

                    val isValidUser = acsValidateUser(user)
```

```
                    if (isValidUser) {
                        LogFile.add(uin)
                        acsHello(user) // Greets the user
                        val key = TUI.getKey()
                        if (key == '#') acsChangePin(user) // Change PIN after
authorized access
                        if (key == '*') acsClearMessage(user) // Clear MSG if it
exists
                        openCloseDoor(user)
                    }
                } else {
                    TUI.writeLine("INVALID USER", ALIGN.Center, LINES.Second) //
User doesn't exist
                    waitTimeMilli(CMD_WAIT_TIME)
                }
            }
            M.init() // checks the M key to enter maintenance mode
            // if after exiting maintenance mode on is set to false via the
close command, then the program will close.

        }
        //Shuts down the system if on is false
        TUI.writeLines("Shutting Down", ALIGN.Center, "...", ALIGN.Center)

        waitTimeMilli(CMD_WAIT_TIME)

        LCD.clear()

    }

    fun enterMaintenance() = TUI.writeLines("OUT OF SERVICE", ALIGN.Center,
"|MAINTENANCE|", ALIGN.Center)


    fun printUsersCommand() {
        for (i in 0 until Users.SIZE) {
            val user = Users.getUser(i)
            if (user != null) println(user)
        }
    }



    /**
     * Adds a user to the system if the user list is not full.
     * Assigns the first available UIN and requests the username and PIN of the
user from the system manager.
     *
```

```
     * The name of the user must be at most 16 characters long.
     *
     * The command is aborted if no input is given.
     *
     * Gets the pin and username inputs by calling [getIntEntry] and
[getStrEntry]
     */
    fun addUserCommand() {
        val username = getStrEntry(ENTRY.USERNAME)
        if (username == null) {
            println("Command aborted.")
            return
        }

        val pin = getIntEntry(ENTRY.PIN)
        if (pin == null) {
            println("Command aborted.")
            return
        }

        val uin = Users.addUser(username, pin)
        if (uin >= 0) {
            println("User $uin $username added successfully.")
        } else {
            println("The user list is full.")
        }
    }


    /**
     * Removes a user from the user list if there is a user for the given UIN.
     *
     * The system requests the system manager to input the
     * UIN, shows the name of the corresponding user, and asks for confirmation.
     *
     * The command is aborted if no input is given.
     * Gets the UIN input by calling [getIntEntry]
     */
    fun removeUserCommand() {
        val uin = getIntEntry(ENTRY.UIN)

        if (uin == null) {
            println("Command aborted.")
            return
        }

        val user = Users.getUser(uin)

        if (user == null) {
            println("User does not exist.")
            return
        }

        print("${user.username}. Are you sure you want to remove this user?
(Y/N) ")

        val confirmation = readln().trim().lowercase()
```

```
        if (confirmation in "sy" && confirmation.isNotEmpty()) {
            Users.removeUser(uin)
            println("User $uin deleted successfully.")
        } else {
            println("User was not removed.")
        }
    }


    /**
     * Unlocks a user
     *
     * @param user the user to unlock
     */
    fun unlockUserCommand() {
        val uin = getIntEntry(ENTRY.UIN)

        if (uin == null) {
            println("Command aborted.")
            return
        }

        val res = Users.unlockUser(uin)

        if (res >= 0)
            println("User unlocked successfully")
        else
            println("User does not exist.")
    }

    /**
     * Allows associating an information message addressed to a particular user
that is presented
     * during the process of authentication to the restricted zone.
     *
     * The command is aborted if no input is given.
     *
     * Gets the UIN and message inputs by calling [getIntEntry] and
[getStrEntry]
     */
    fun addMessageCommand() {
        val uin = getIntEntry(ENTRY.UIN)

        if (uin == null) {
            println("Command aborted.")
            return
        }

        val msg = getStrEntry(ENTRY.MSG)

        if (msg == null) {
            println("Command aborted.")
            return
        }

        val res = Users.setMsg(msg, uin)
```

```
        if (res >= 0)
            println("Message delivered successfully")
        else
            println("User does not exist.")
    }

    /**
     * Allows shutting down the [AccessControlSystem]. The system asks the user
to confirm the command and
     * writes the user information to a text file (one user per line).
     *
     * Updates the users using the [Users.close] function
     *
     *
     *
     */
    fun closeCommand() {
        print("Shut down the Access Control System? (Y/N): ")

        val confirmation = readln().trim().lowercase()

        if (confirmation in "sy" && confirmation.isNotEmpty()) {
            close()
            println("System updated. You can now shut the system down by exiting
maintenance mode (Turn M off).")
        } else
            println("The system was not shut down.")
    }


    /**
     * Retrieves a string entry from the system manager.
     *
     * The entry must not exceed 16 characters, since that is the maximum length
supported by the LCD.
     *
     * @param entry The type of entry (USERNAME, MSG)
     * @return The system manager provided string entry, or null if the entry
was aborted.
     *
     *
     */
    private fun getStrEntry(entry: ENTRY): String? {
        var str = "                  "
        while (str.length > entry.len) {

            print("$entry : ")
            str = readln().trim()

            if (str.isEmpty()) return null

            if (str.length > entry.len)
                println("The $entry must not exceed ${entry.len} chars.")
        }

        return str
```

```
    }


    /**
     * Retrieves an integer entry from the system manager.
     *
     * The number value must not be larger than the number implicitly defined by
entry length (e.g. 999 for UIN length 3).
     *
     * @param entry The type of entry (UIN, PIN).
     * @return The system manager provided integer entry, or null if the entry
was aborted.
     *
     *
     */
    private fun getIntEntry(entry: ENTRY): Int? {
        var maxNum = 1
        for (i in 0 until entry.len) {
            maxNum *= 10
        }
        var num = -1
        while (num !in 0 until maxNum) {
            try {
                print("$entry: ")
                val numstr = readln()

                if (numstr.isEmpty()) return null
                num = numstr.toInt()

                if (num < 0) println("Value must be positive")
                else if (num >= maxNum) println("The $entry must not exceed
${entry.len} digits")

            } catch (e: NumberFormatException) {
                println("Value must be a number.")
            }
        }
        return num
    }


    /**
     * Validates the user's PIN for authentication.
     *
     * The user is given 3 attempts.
     *
     * @param user The user to validate.
     * @return true if the user is validated, false if all the 3 attempts fail
or the command is aborted.
     *
     */
    fun acsValidateUser(user: User): Boolean {

        if(user.blocked) {
            TUI.writeLine("USER BLOCKED", ALIGN.Center, LINES.First)
            TUI.writeLine("CONTACT SYS MAN", ALIGN.Center, LINES.Second)
            waitTimeMilli(CMD_WAIT_TIME * 2)
```

```
            return false
        }

        var attempts = 0

        while (attempts < MAX_ATTEMPTS) {

            val pin = TUI.query("PIN:", ALIGN.Center, LINES.Second, ENTRY.PIN)

            if (Users.isValidLogin(user, pin)) return true
            else {

                if (pin in CMD_ABORT_CODES) return false //Return false if
aborted

                TUI.writeLine("INVALID PIN (${attempts + 1})", ALIGN.Center,
LINES.Second)

                attempts++

                waitTimeMilli(CMD_WAIT_TIME)

            }

        }
        Users.blockUser(user.UIN)
        TUI.writeLines("Too many tries", ALIGN.Center, "User Blocked",
ALIGN.Center)
        waitTimeMilli(CMD_WAIT_TIME)

        return false //Return false if all attempts failed

    }



    /**
     * Changes the pin for the specified user after authorized access via the
[AccessControlSystem]
     *
     * The [AccessControlSystem] queries the user two times, one for the new PIN
and another for the confirmation of the new PIN.
     *
     * If both PIN inputs are the same, and the command was not aborted, the PIN
is changed.
     *
     * @param user the user to change the pin for
     */
    private fun acsChangePin(user: User) {

        TUI.writeLines("Change PIN?", ALIGN.Center, "* to confirm ",
ALIGN.Center)

        if (KBD.waitKey(KBDTIMEOUT) == '*') {

            TUI.writeLine("New PIN?", ALIGN.Center, LINES.First)
```

```
            val newPin = TUI.query("PIN:", ALIGN.Center, LINES.Second,
ENTRY.PIN)

            TUI.writeLine("Confirm PIN:", ALIGN.Center, LINES.First)

            val confirmPin = TUI.query("PIN:", ALIGN.Center, LINES.Second,
ENTRY.PIN)

            if (newPin == confirmPin && newPin !in CMD_ABORT_CODES) {
                user.pin = newPin
                TUI.writeLines("PIN", ALIGN.Center, "Changed", ALIGN.Center)
            } else {

                TUI.writeLines("PIN", ALIGN.Center, "Kept", ALIGN.Center)

            }

        } else {

            TUI.writeLines("PIN", ALIGN.Center, "Kept", ALIGN.Center)

        }

        waitTimeMilli(CMD_WAIT_TIME)

    }


    /**
     * Clears the set user message if it exists after authorized access via the
[AccessControlSystem].
     *
     * The [AccessControlSystem] queries the user for confirmation to delete the
message. If the user confirms
     * the deletion of the message, the message is deleted.
     *
     */
    private fun acsClearMessage(user: User) {

        if (user.message == "") return

        TUI.writeLines("Clear Message?", ALIGN.Center, "* To confirm ",
ALIGN.Center)

        if (KBD.waitKey(KBDTIMEOUT) == '*') {

            user.message = ""

            TUI.writeLines("Message", ALIGN.Center, "Cleared", ALIGN.Center)

            waitTimeMilli(CMD_WAIT_TIME)

        } else {

            TUI.writeLines("Message", ALIGN.Center, "Kept", ALIGN.Center)
```

```
            waitTimeMilli(CMD_WAIT_TIME)
        }

    }

    /**
     * Displays a greeting message for the user after authentication via the
[AccessControlSystem].
     *
     * Displays the set user message if there is one after waiting
[CMD_WAIT_TIME].
     */
    private fun acsHello(user: User) {

        TUI.writeLines("Hello", ALIGN.Center, user.username, ALIGN.Center)

        waitTimeMilli(CMD_WAIT_TIME)

        if (user.message != "") {
            TUI.writeLines(user.username, ALIGN.Center, user.message,
ALIGN.Center)
            waitTimeMilli(CMD_WAIT_TIME)
        }

    }


    /**
     * Opens the door for the authorized user after authentication via the
[AccessControlSystem].
     *
     * The [AccessControlSystem] starts by informing the user that the door is
being opened, and opens the door.
     * After the door is opened, the [AccessControlSystem] waits [CMD_WAIT_TIME]
* 2 for the user to enter.
     *
     * After waiting, the [AccessControlSystem] informs the user that the door
is being closed and closes the door.
     *
     * @param user The user the door is being opened for
     *
     */

    private fun openCloseDoor(user: User) {

        TUI.writeLines(user.username, ALIGN.Center, "Opening Door",
ALIGN.Center)
        waitTimeMilli(CMD_WAIT_TIME / 2)

        DoorMechanism.open(DOOR_OPEN_SPEED)

        TUI.writeLine("Door Open", ALIGN.Center, LINES.Second)
        waitTimeMilli(CMD_WAIT_TIME * 2)

        TUI.writeLine("Door Closing", ALIGN.Center, LINES.Second)
        DoorMechanism.close(DOOR_CLOSE_SPEED)
```

```
            TUI.writeLine("Door Closed", ALIGN.Center, LINES.Second)
            waitTimeMilli(CMD_WAIT_TIME)

        }

        private fun close() {
            Users.close("USERS.txt")
            on = false
        }


}
```