Payment Application Project

(1)Card Module:

This module is used to deal with card data when it is inserted to the payment application,here we implemented three functions with their test functions.

1)Implementing getCardHolderName function:

```
  EN_cardError_t getCardHolderName(ST_cardData_t* cardData)
{
    //Start with card ok state
    EN_cardError_t errorStateHolderName = CARD_OK;
    uint8_t inputLen;
    uint8_t iterate = 0;
    *inputFromUser = ' ';

    //Get card holder name from the user
    printf("Enter Card Holder Name: ");
    fgets(inputFromUser, sizeof(inputFromUser), stdin);

     inputLen = strlen(inputFromUser);

    //Remove the newline from the holder name
    if (inputFromUser[inputLen - 1] == '\n')
    {
        inputFromUser[inputLen - 1] = 0;
    }
    //Check if the input is null
    if (strlen(inputFromUser) == 0)
    {
        errorStateHolderName = WRONG_NAME;
    }
    //Check the length of the name to be min 20 and max 24 characters
    if (strlen(inputFromUser) < 20 || strlen(inputFromUser) > 24)
    {
        errorStateHolderName = WRONG_NAME;
    }
    //Check if the length is in the acceptable range
    if (strlen(inputFromUser) >= 20 && (strlen(inputFromUser) <= 24))
    {
        //This condition to check if the name is in the right format
contains only alphabetic characters and spaces
        for ( iterate  = 0; iterate < strlen(inputFromUser); iterate++)
        {
            if (!isalpha(inputFromUser[iterate]) &&
!isspace(inputFromUser[iterate]))
            {
                errorStateHolderName = WRONG_NAME;
                break;
            }
        }
    }
    if (errorStateHolderName == CARD_OK)
    {
        //To store the input only if it is right
        strcpy_s(cardData->cardHolderName, sizeof(inputFromUser),
inputFromUser);
    }
    return errorStateHolderName;
}
```

This function is used to get card holder name by asking the user to enter the name ang gets it using **fgets** function and returns the error state specified for it in the error state enum for this module depending on the input of each case.
inputFromUser is a global array used to get the input from the user.
It has some constraints such as this name must be between 20 and 24 characters as it is stored in an array of 25 element and not equal to **NULL**,so **"if statements"** are used to check over these conditions.
At the end if the entered name is in the acceptable range and the right format (contains only alphabetic characters and spaces) it is stored in **cardHolderName** in the **cardData** struct using **strcpy** function to copy the input into the struct data and the return of the function is **"CARD_OK"** which is equal to 0 and if not it doesn't store the input and returns **"WRONG_DATE"** which is equal to 1 .

2)Implementing getCardExpiryDate function:

```
EN_cardError_t getCardExpiryDate(ST_cardData_t *cardData)
{
     uint8_t inputLen;
     EN_cardError_t errorStateExpireDate = WRONG_EXP_DATE;
     *inputFromUser = ' ';
     uint8_t iterate = 0;

     //Get the expiration date from user
     printf("Enter Card Expiry Date: the format (MM/YY), e.g (05/25): ");
     fgets(inputFromUser , sizeof(inputFromUser) , stdin);

      inputLen = strlen(inputFromUser);

     //Remove the newline from InputExp to calculate the length
     if(inputFromUser[inputLen-1] == '\n')
     {
        inputFromUser[inputLen-1] = 0;
     }

     //Check the length of Exp date
     if(strlen(inputFromUser) !=5)
     {
         errorStateExpireDate = CARD_OK;
     }
     //Check the format
     for(iterate = 0 ; iterate < 5 ; iterate++)
     {
        if( (isdigit(inputFromUser[iterate])) == 0  && iterate != 2  )
        {
            errorStateExpireDate = CARD_OK;
        }
        if(iterate == 2 && inputFromUser[iterate] != 47)
        {
            errorStateExpireDate = CARD_OK;
        }
     }

     //Check the return status
     if(errorStateExpireDate == CARD_OK)
     {
         errorStateExpireDate = WRONG_EXP_DATE;

     }
     else
     {
       strcpy_s(cardData->cardExpirationDate , sizeof(inputFromUser) ,
inputFromUser );
       errorStateExpireDate =  CARD_OK;
     }
     return errorStateExpireDate;
}
```

This function is used to deal with expiry date of the card, using the **inputFromUser** array to get the input using **fgets** function
The logic of this function is to check the length and format of the entered date .It must be like this **"DD/YY"**,like this example**: "05/25".** So a **"for loop"** is used to iterate over the digits of the input to check that it is a number and that the third digit (second as we start from zero in arrays,strings and arrays) is equal to the **"/"** character.
If this check is right the input data is stored in **cardExpirationDate** variable in the **cardData** struct and returns **CARD_OK** state which is equal to 0, if anything is false in the length or the format the function returns **WRONG_EXP_DATE** which is equal to 2 and stores nothing in the struct.

-Third function is:

```
EN_cardError_t getCardPAN(ST_cardData_t *cardData)
{
    /*To Make Error State For PAN And Return This Error State*/
    EN_cardError_t ErrorStatePAN = CARD_OK;
    *inputFromUser = ' ';
    uint8_t iterate = 0;

    printf("Enter Primary Account Number: ");
    gets(inputFromUser);
    //Remove the newline from InputExp to calculate the length
    uint8_t inputLen = strlen(inputFromUser);
    if (inputFromUser[inputLen - 1] == '\n')
    {
        inputFromUser[inputLen - 1] = 0;
    }

    /*This Condition If User Input Null (Dont Input Any Thing*/
    if (strlen(inputFromUser) == NULL || strlen(inputFromUser) < 16 ||
strlen(inputFromUser) > 19)
    {
        ErrorStatePAN = WRONG_PAN;
    }

    //Numeric chars check
    for (iterate = 0; iterate < strlen(inputFromUser); iterate++) {
        if (isdigit(inputFromUser[iterate]) == 0)
        {
            if (isspace(inputFromUser[iterate]) == 0)
            {

                ErrorStatePAN = WRONG_PAN;
            }
        }
    }
    if (ErrorStatePAN == CARD_OK)
    {
        strcpy_s(cardData->primaryAccountNumber,
sizeof(cardData->primaryAccountNumber), inputFromUser);
    }
    return ErrorStatePAN;
}
```

"getCardPAN" function checks the Primary Account Number to be stored in our struct.it gets the PAN from the user.
First it checks the length of this PAN that must be between 16 and 19 number and not equal to zero using "if statements" to check this constraint .
Then implements a "for loop" to iterate over the length of this input and checks that they are only numbers (digits) and there are no spaces if all is right, it stores this PAN in the **primaryAccountNumber** variable in **cardData** struct and returns a **CARD_OK** state which is equal to 0.
If there is any error in the length or the specified format ,it returns **WRONG_PAN** which is equal to 3.

*There is a piece of code used in all these three function which is:

```
 uint8_t inputLen = strlen(inputFromUser);
    if (inputFromUser[inputLen - 1] == '\n')
    {
        inputFromUser[inputLen - 1] = 0;
    }
```
We used a new variable named It is used inputLen of the type uint8_t to store the length of the input and then checks if "\n" is entered which means a new line (Enter key is pressed) so the input is done , it removes this new line and adds a "0" at the end to assign a null terminator so this string is done.
***All these functions take a pointer to cardData struct to fill it if the input is in a right format.**

<u>(2)Terminal Module:</u>

This module works on transaction date,transaction amount,maximum transaction amount and checks if the card id expired,the transaction is allowed.
We have implemented five functions in this module that will be discussed below.

1)Implementing getTransactionDate function:

```
EN_terminalError_t getTransactionDate(ST_terminalData_t *termData)
{
    //Transaction date format -> 10 digits  (29/09/2023).
    time_t currentTime;
    struct tm localTime;
    uint8_t inputLen;
    uint8_t iterate = 0;
    uint8_t dateString[11];  // "DD/MM/YYYY" plus null terminator.
    *inputFromUser = ' ';

    //Error state with initial state as TERMINAL_OK.
    EN_terminalError_t errorStateTransactionDate = TERMINAL_OK;

    // Get the current time
    time(&currentTime);
    // Convert to local time
    localtime_s(&localTime, &currentTime);
    //store Date in dateString variable
    sprintf_s(dateString, sizeof(dateString), "%02d/%02d/%04d",
localTime.tm_mday, localTime.tm_mon + 1, localTime.tm_year + 1900);
    printf("system date %s \n" , dateString);


    //Get the transaction date from user
     printf("Enter the transaction date in a this format , e.g (DD/MM/YYYY)
: ");
     fgets(inputFromUser , sizeof(inputFromUser) , stdin);
     inputLen = strlen(inputFromUser);

     //Remove the newline from InputExp to calculate the length.
     if(inputFromUser[inputLen-1] == '\n')
     {
        inputFromUser[inputLen-1] = 0;
     }

     //Checking the length of inputFromUser.
     if( strlen(inputFromUser) != 10 || inputFromUser == NULL)
     {
         errorStateTransactionDate = WRONG_DATE;
     }

     //Checking the format.
     for(iterate ; iterate < strlen(inputFromUser) ; iterate++)
     {
         // iterate = 2 or 5 that the '/'
         if( (isdigit(inputFromUser[iterate])) == 0  && iterate != 2 &&
iterate != 5 )
         {
             errorStateTransactionDate = WRONG_DATE;
         }
          if((iterate == 2 || iterate == 5 )&& inputFromUser[iterate] !=
47)
        {
            errorStateTransactionDate = WRONG_DATE;
        }
     }

     //printf("Wrong transaction date - Use system date ");
     strcpy_s(termData->transactionDate, sizeof(termData->transactionDate)
,dateString);

     //return error state
     return errorStateTransactionDate;
}
```

First of all there are two variables used to get the system's date,a **currentTime** variable of type **"time_t"** which is a date type used to represent time values, particularly in the context of the C standard library. It can store the number of seconds elapsed since a specific point in time.
Then we have a localTime **variable of** type "**struct.tm"** which is a C structure used to represent a calendar time broken down into its components, such as year, month, day, hour, minute, and second. It's commonly used for manipulating and formatting time-related information.These two are used to get the system data.

This function takes the transaction date from the user in the format **"DD/MM/YYYY"** and uses iterations in the **"for loop"** to check the length and format of this input data, if there is any error it returns **WRONG_DATE** and stores the system's date instead.
If the input is right it returns **TERMINAL_OK** and stores the transaction date .

2)Implementing isCardExpired function:

```
EN_terminalError_t isCardExpired(ST_cardData_t* cardData,
ST_terminalData_t* termData)
{
    EN_terminalError_t errorStateCardExpired = TERMINAL_OK;
    uint8_t* endptr;

    //convert string to long integer using strtol and casting its return to
uint8_t
    long expMonth = strtol(cardData->cardExpirationDate, &endptr, 10);
    long expYear = strtol(&cardData->cardExpirationDate[3], &endptr, 10);
    long transactionMonth = strtol(&termData->transactionDate[3], &endptr,
10);
    long transactionYear = strtol(&termData->transactionDate[8], &endptr,
10);

    //Check if the expiry year is larger than year of transaction date so
it's not expired
    if (expYear > transactionYear)
    {
        errorStateCardExpired = TERMINAL_OK;
    }
    else if (expYear == transactionYear)
    {
        //If the years are the same compare the months
        if (expMonth >= transactionMonth)
        {
            errorStateCardExpired = TERMINAL_OK;
        }else {
            errorStateCardExpired = EXPIRED_CARD ;
        }
    }
    else
    {
        errorStateCardExpired = EXPIRED_CARD;
    }

    return errorStateCardExpired;
}
```

The logic behind this function is to compare the transaction date (month and year) with the card expiry date to check if the card is expired or not.
First we convert the string containing the date into long integers as the function **strtol** returns a long integer after converting it from string

Long int strtol(char* str,char* endptr,int base);
Enptr is a pointer used by the **strtol** function that points to the first non-integer character signaled to stop conversion and base is 10.
In simple words we compare years if the expiry year is larger this means it is not expired and if the years are equal we check the month to see if it’s expired or not.
If not it returns **TERMINAL_OK** ,but if the expiry date is smaller than transaction date this means that the card is expired and the function returns the error state **EXPIRED_CARD.**

3)Implementing getTransactionAmount function:

```
EN_terminalError_t getTransactionAmount(ST_terminalData_t *termData)
{
    *inputFromUser = 0;
    /*This is The Error State To Return In The End Of Function*/
    EN_terminalError_t errorStateTransactionAmount = INVALID_AMOUNT;
    uint8_t iterate = 0;
    uint8_t* storeFirstInvalidChar;

    /*Ask User To Input Amount Transaction*/
    printf("Enter The Transaction Amount: ");
    fgets(inputFromUser, sizeof(inputFromUser), stdin);

    /*Check The User Input Is Numbers Only and Not Spaces*/
    for (iterate = 0; iterate < strlen(inputFromUser) - 1; iterate++)
    {
        if (isdigit(inputFromUser[iterate]))
        {
            errorStateTransactionAmount = TERMINAL_OK;
        }
        if (isspace(inputFromUser[iterate]) && inputFromUser[iterate] !=
'.')
        {
            errorStateTransactionAmount = INVALID_AMOUNT;
            break;
        }
    }
    /*
    The &storeFirstInvalidChar argument is used to store the address of the
first invalid character
    encountered during the conversion. After the conversion,
    storeFirstInvalidChar will point to the first character
    that couldn't be converted to a float.
    This can be useful if you need to check for errors
    or process any remaining characters in the input string that were not
converted.
    */
    termData->transAmount = strtof(inputFromUser , &
storeFirstInvalidChar);

    if (termData->transAmount <= 0)
    {
        errorStateTransactionAmount =  INVALID_AMOUNT;
    }

    return errorStateTransactionAmount;
}
```

getTransactionAmount function checks the input transaction amount to be not less than or equal to zeros and checks also that the user input is only numbers and doesn’t contain any spaces.

Function **strtof** is used here to convert this string to a float number and in the function implementation and **storeFirstInvalidChar** will point to the first character that couldn't be converted to a float.

4)Implementing is belowMaxAmount function:

```
EN_terminalError_t isBelowMaxAmount(ST_terminalData_t *termData)
{
    EN_terminalError_t errorStateBelwMax = TERMINAL_OK;
   if(termData->transAmount > termData->maxTransAmount )
   {
       // transaction amount is larger than the terminal max allowed amount
       errorStateBelwMax = EXCEED_MAX_AMOUNT;
   }
   return errorStateBelwMax;
}
```

The **isBelowMaxAmount** function checks whether a transaction amount is below the maximum allowed amount for a terminal and returns an appropriate error state. Here's an explanation of the function:
It takes a pointer to a ST_terminalData_t structure as an argument and returns an EN_terminalError_t value, which represents the error state.
An if statement is used that checks if the transAmount (transaction amount) stored in the termData structure is greater than the maxTransAmount (maximum allowed transaction amount) stored in the same structure. it means that the transaction amount exceeds the maximum allowed amount and the error state is assigned to **EXCEED_MAX_AMOUNT.**
If not, then the transaction can be done and the error state is **TERMINAL_OK.**

5)Implementing setMaxAmount function:

```
EN_terminalError_t setMaxAmount(ST_terminalData_t *termData, float
maxAmount)
{
    /*This is The Error State To Return In The End Of Function*/
    EN_terminalError_t errorStateMaxAmount;

    /*Make This With Initial Value*/
    termData->maxTransAmount;

    /*Check If Max Amount Less Than 0 Or Equal*/
    if (maxAmount <= 0)
    {
        errorStateMaxAmount = INVALID_MAX_AMOUNT;
    }
    else
    {
        errorStateMaxAmount = TERMINAL_OK;
    }
    termData->maxTransAmount = maxAmount;
    return errorStateMaxAmount;
}
```

This function is another simple function used to set a maximum transaction amount and checks that can not be less than or equal to zero
If a valid amount is entered it is stored in the termData struct as the maxAmount and then returned error state is **TERMINAL_OK**.
If the entered value is not valid, the function returns an error state equal to **INVALID_MAX_AMOUNT.**

(3)Server Module:

This module is used mainly in implementing the database of the system.

First, a global array of **ST_accountsDB_t** struct is created to store the valid accounts database.
ST_accountsDB_t accountsDB[255]. It contains the PAN number associated with the account,its balance and its state if **RUNNING** or **BLOCKED** if stolen

Another global array of **ST_transaction_t** struct of maximum size 255 is created to store the transactions database.

1)Implementing recieveTransactionData function:

```
EN_transState_t recieveTransactionData(ST_transaction_t* transData)
{
    ST_accountsDB_t accountRefrence = { 0 };
    if (isValidAccount(&transData->cardHolderData, &accountRefrence) == 3)
    {
        return FRAUD_CARD;
    }
    if (isBlockedAccount(&accountRefrence) == 5)
    {
        return DECLINED_STOLEN_CARD;
    }
    if (isAmountAvailable(&transData->terminalData, &accountRefrence) == 4)
    {
        return DECLINED_INSUFFECIENT_FUND;
    }
    //update the transaction status ( APPROVED ).
    transData->transState = APPROVED;

    if (saveTransaction(transData) == 1) //saveTransaction(transData) ==
SAVING_FAILED.
    {
        return INTERNAL_SERVER_ERROR;
    }

    //update the database with the new balance.
    for (int i = 0; i <= 255; i++) {
        if (strcmp(accountsDB[i].primaryAccountNumber,
accountRefrence.primaryAccountNumber) == 0)
        {
            accountsDB[i].balance -= transData->terminalData.transAmount;
            break;
        }
    }
    return transData->transState;
}
```

This function handles the transaction data and determines its approval status by checking various conditions related to the card, account, and transaction data. Here's an explanation of the function:
It takes a pointer to a **ST_transaction_t** structure (transData) as an argument and returns an **EN_transState_t** value, which represents the transaction's approval status.
The function proceeds to check several conditions with **"if statements"** one by one, and if any of them fail, it sets the **errorStateRecieveTrans** variable to an appropriate error code.
It first checks if the card is a **fraud card** by calling the **isValidAccount** function and looking for a return value of **3**, which corresponds to **FRAUD_CARD**.
If the card is not a **fraud card**, it checks if there are sufficient funds available for the transaction by calling the **isAmountAvailable** function and looking for a return value of 4, which corresponds to **DECLINED_INSUFFECIENT_FUND**.
If the transaction amount is within the available balance, it checks if the account is **BLOCKED** by calling the **isBlockedAccount** function and looking for a return value of **5**, which corresponds to **DECLINED_STOLEN_CARD**.
If none of the above conditions are met, it proceeds to save the transaction by calling the **saveTransaction** function. If the return value of saveTransaction is **1**, which corresponds to **SAVING_FAILED**, it sets **errorStateRecieveTrans** to **INTERNAL_SERVER_ERROR**.

After checking all the conditions, if none of the conditions fail, it updates the transaction status to **APPROVED** and sets **errorStateRecieveTrans** to **APPROVED**.
Finally, the function updates the account balance in the database by subtracting the transaction amount.
The function returns **errorStateRecieveTrans**, which represents the approval status of the transaction. It can have values like **APPROVED, FRAUD_CARD**, **DECLINED_INSUFFECIENT_FUND, DECLINED_STOLEN_CARD,** or **INTERNAL_SERVER_ERROR.**
This function essentially processes a transaction by checking various conditions related to the card and account, updates the transaction status, and updates the account balance in the database accordingly.

2)Implementing isValidAccount function:

```
EN_serverError_t isValidAccount(ST_cardData_t* cardData, ST_accountsDB_t*
accountRefrence)
{
    EN_serverError_t errorState = ACCOUNT_NOT_FOUND;
    for (int iterate = 0; iterate <= 255; iterate++)
    {
        /*Check The User Enterd PAN ANd Compare It To The PANs In Data
Base*/
        if (strcmp(accountsDB[iterate].primaryAccountNumber,
cardData->primaryAccountNumber) == 0)
        {
            *accountRefrence = accountsDB[iterate];
            errorState = SERVER_OK;
        }
    }
    accountRefrence = NULL;
    return errorState;
}
```

The **isValidAccount** function is designed to validate a card's primary account number (**PAN**) by comparing it with the PANs stored in the database (presumably accountsDB). Here's an explanation of the function:
 It takes a pointer to a **ST_cardData_t** structure (**cardData**) representing card information and a pointer to a **ST_accountsDB_t** structure (**accountRefrence**) where the account details will be stored if a match is found. It returns a **EN_serverError_t** value, which represents the error state of the function.

cmpPAN variable will be used to store the result of string comparison between the provided **PAN** and the PANs in the database.

The function then enters a **"for loop"** that iterates through the accounts in the database. The loop continues until it reaches an account with a PAN that matches the provided card's PAN or until it reaches the end of the **accountsDB** array.
Inside the loop, it compares the provided card's PAN (**cardData->primaryAccountNumber**) with the PAN of the current account in the database (**accountsDB[iterate].primaryAccountNumber**) using **strcmp**.

If the **PAN** match (i.e., cmpPAN == 0), it copies the details of the account from the database to the **accountRefrence** structure using ***accountRefrence = accountsDB[iterate]**,

indicating that the account is found, and sets **errorStateValidAccount** to **SERVER_OK**. It then breaks out of the loop.
If the **PAN**s do not match, it sets errorStateValidAccount to **ACCOUNT_NOT_FOUND** to indicate that the account was not found and sets accountRefrence to **NULL**.
Finally, the function returns the **errorStateValidAccount**, whether **SERVER_OK** or **ACCOUNT_NOT_FOUND**.

3)Implementing isBlockedAccount function:
```
EN_serverError_t isBlockedAccount(ST_accountsDB_t* accountRefrence)
{
    EN_serverError_t errorState = BLOCKED_ACCOUNT;
    /*Check Account State If Runnig Return Server OK Else Is Block And
Return Block*/
    if (accountRefrence->state == RUNNING)
    {
        errorState = SERVER_OK;
    }
    return errorState;
}
```

The **isBlockedAccount** function is designed to check a card's state if **RUNNING** or **BLOCKED** by comparing its state with the state stored in the database associated with the input reference (presumably accountsDB).
If the account's state is **RUNNING** it sets the **errorStateBlockedAccount** to **SERVER_OK,** if not,then the **errorStateBlockedAccount** is set to **BLOCKED_ACCOUNT.**
Finally, the function returns the **errorStateBlockedAccount** value.

4)Implementing isAmountAvailable function:

```
EN_serverError_t isAmountAvailable(ST_terminalData_t* termData,
ST_accountsDB_t* accountRefrence)
{
    EN_serverError_t errorState = SERVER_OK;
    /*Check And Compare Between Trans Amount in terminal and The Balance in
database */
    if (termData->transAmount > accountRefrence->balance)
    {
        errorState = LOW_BALANCE;
    }
    return errorState;
}
```

The **isAmountAvailable** function is designed to check if the transaction amount is available in the account by comparing terminal data **transAmount** with the balance stored in the database associated with the input reference .

If the transaction amount is larger than the balance stored then the **errorStateAmountAvilable** is set to **LOW_BALANCE** which is equal to 4 and if the transaction can be done as the transaction amount is smaller than the stored balance this means that the **errorStateAmountAvilable** is now set to **SERVER_OK** which is equal to 0.

At the end the function returns the **errorStateAmountAvilable** which declares the case of the transaction.

5)Implementing saveTransaction function:

```
EN_serverError_t saveTransaction(ST_transaction_t* transData)
{
    EN_serverError_t errorState = SAVING_FAILED;
    /*Loop On All Data Base*/
    for (int iterate = 0; iterate < 255; iterate++)
    {
        /*If Seq Num IS 0 it Mean No Trans Is Happen*/
        if (transactionDB_t[iterate].transactionSequenceNumber == 0)
        {
            /*Store*/
            transactionDB_t[iterate].cardHolderData =
transData->cardHolderData;
            transactionDB_t[iterate].terminalData =
transData->terminalData;
            /*If Seq Num Is Zero IS No Trans And Begin With Init Value*/
            if (iterate == 0)
            {
                transactionDB_t[iterate].transactionSequenceNumber = 1000;
            }
            /*Here To Increment in All Transaction*/
            if (iterate > 0)
            {
                transactionDB_t[iterate].transactionSequenceNumber =
transactionDB_t[iterate - 1].transactionSequenceNumber + 1;
            }
            transactionDB_t[iterate].transState = transData->transState;
            listSavedTransactions();

            errorState = SERVER_OK;
        }
    }
    return errorState;
}
```

In this function, first, we iterate over the database if the sequence number is equal to zero and also the iteration is zero ,then initialise the sequence number with a certain value (i.e:404) .

If the iteration is larger than zero, which means that this is not the first transaction, in this case the sequence number is incremented.

6)Implementing listSaveTransactions function:

```
void listSavedTransactions(void)
{
   uint8_t iterate = 0;
   uint8_t trans[5][30] = {
                        {"APPROVED"},
                        {"DECLINED_INSUFFECIENT_FUND"},
                        {"DECLINED_STOLEN_CARD"},
                        {"FRAUD_CARD"},
                        {"INTERNAL_SERVER_ERROR"} };

   for (iterate = 0; iterate <= 255; iterate++)
   {
      if (transactionDB_t[iterate].transState < 0 || transactionDB_t[iterate].transState > 4)
      {
         break;
      }
      if (transactionDB_t[iterate].transactionSequenceNumber == 0)
      {
         break;
      }

      printf("#######################################################\n");
      printf("Transaction Sequence Number: %d\n",
transactionDB_t[iterate].transactionSequenceNumber);
      printf("Transaction Date: %s\n",
&transactionDB_t[iterate].terminalData.transactionDate);
      printf("Transaction Amount: %f\n",
transactionDB_t[iterate].terminalData.transAmount);
      printf("Transaction State: %s\n", trans[transactionDB_t[iterate].transState]);
      printf("Terminal Max Amount: %f\n",
transactionDB_t[iterate].terminalData.maxTransAmount);
      printf("Cardholder Name: %s\n",
transactionDB_t[iterate].cardHolderData.cardHolderName);
      printf("PAN: %s\n", transactionDB_t[iterate].cardHolderData.primaryAccountNumber);
      printf("Card Expiration Date: %s\n",
transactionDB_t[iterate].cardHolderData.cardExpirationDate);
      printf("#######################################################\n");
   }
}
```

This is the last function of this module and it is used to print the saved transactions in an appropriate format to view all transaction's data such as its date,amount, sequence number and also the cardholder data like his name,PAN and expiration date.
It is a void function that returns nothing as it is only used for printing.