COLLEGE OF COMPUTING & DATA SCIENCE

NANYANG TECHNOLOGICAL UNIVERSITY

SC4015: Cyber Physical System Security

Smart Card Lab Report

Hashil Jugjivan

U2120599F

1 Introduction

A smart card is a credit card-sized plastic card with an embedded computer chip. Smart cards play a critical role in modern payment and access control systems. Compared to older magnetic stripe cards, chip-based cards such as MIFARE Ultralight and MIFARE Classic incorporate cryptographic elements and dedicated memory structures to ensure the confidentiality and integrity of data. In this lab, we explored the features of these smart cards, including their memory organisation, methods for secure reading and writing of data, and integration with smart card readers with both a GUI tool and Python scripts.

2 Brief Background

2.1 Importance of Chip-Based Smart Cards vs Magnetic Stripe Cards

1. Enhanced Security: Magnetic stripe cards store data in a static format on the stripe, with no encryption. In contrast, chip-based cards rely on cryptographic features, making it far harder to clone or tamper with data.
2. Reduced Cloning Risk: Due to widespread backwards compatibility with various Point of Sale (POS) terminals globally that might not follow standard security standards and protocols, and lack of encryption, it is relatively easy to clone data from magnetic stripe cards. This allows criminals to clone metadata from the magnetic stripe smart card and duplicate the card onto another magnetic stripe smart card. Malicious actors can then use the cloned card to perform transactions which will be accepted by most banks. Chip-based cards, on the other hand, require knowledge of cryptographic keys, which significantly raises the difficulty for malicious duplication or cloning.

2.2 Features of Cards

1. MIFARE Ultralight Smart Card
    a. Contactless: Operates over a radio frequency interface, removing the need for direct physical contact with a reader.
    b. Limited Memory: Used in single-journey tickets for public transport or day passes for events. It has about 64 bytes of memory and has 48 bytes for storing data
    c. Password Authentication: A basic protection mechanism to secure write operations.
    d. Memory Organisation: Organised into pages or blocks.
2. MIFARE Classic Smart Card
    a. Contactless: Similar to Ultralight, but with a larger memory size.
    b. Sector-based Memory: Memory is divided into sectors, each consisting of 4 blocks, where each sector has its own keys and access conditions.
    c. Security Features: Requires authentication before reading or writing. Has Key A and Key B for controlling read or write access to each sector. The default keys can be changed to enhance security.

3 Lab Proceedings

3.1 MIFARE Ultralight Smart Card Experiments

To connect to the MIFARE Ultralight card, a computer-based smart card reader was used, which was connected to the computer using its USB port. To access the card, the ACS Smart Card and Reader Tool was used to perform various experiments on the card. First, we establish a connection with the card by placing the card on the reader and selecting a new connection in the tool. Next, we selected the appropriate interface: ACS ACR 1281 1S Dual Reader PICC 0.

3.1.1 Steps, Procedures and Explanations

1 Answer to Reset (ATR)

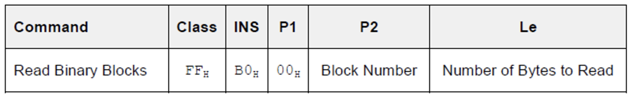
Upon presenting the card to the reader, the card returns an ATR, which is a string of bytes indicating protocol parameters and capabilities, but does not uniquely identify the card. It also indicates what language the card supports. The ATR confirms that the card is responding and indicates which communication protocols and data rates it supports. This helps software determine if the card can handle specific commands or advanced security features. The ATR obtained for the card was: 3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 01 00 00 00 00 6A. This can be seen in Figure 1 of the appendix.

2 Reading Card UID

To read the card's unique identifier (UID), we sent an Application Protocol Data Units (APDU) command such as FF CA 00 00 00, which requests the card's UID. This shows how each card can be uniquely identified, which is important for access control or tracking usage in real-world scenarios. The UID obtained was: 04 62 0E C2 98 11 94, which can be seen in Figure 2 of the appendix.

3 Reading and Writing Data (Pages/Blocks)

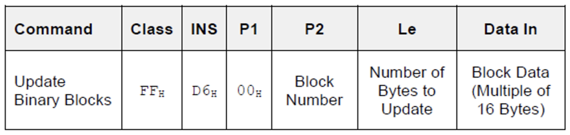
Memory in MIFARE Ultralight is organised into pages, where each page holds 4 bytes. To read data stored in the card, we need to use an APDU command of the following format:



Command	Class	INS	P1	P2	Le
Read Binary Blocks	FF$_H$	B0$_H$	00$_H$	Block Number	Number of Bytes to Read

For instance, to read 4 bytes from page/block 4, we used the following command: FF B0 00 04 04. This can be seen in Figure 3 of the appendix.

To write data to a particular page/block in the card, we need to use an APDU command of the following format:

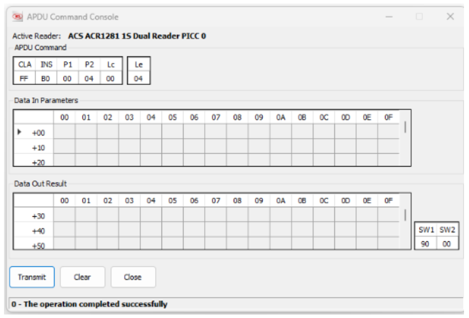


Command	Class	INS	P1	P2	Le	Data In
Update Binary Blocks	FF$_H$	D6$_H$	00$_H$	Block Number	Number of Bytes to Update	Block Data (Multiple of 16 Bytes)

For instance, to write 4 bytes to page/block 4, we used the following command: FF D6 00 04 04 00 10 00 11. To ensure the data was written, we read block 4 again to see if the data had changed. This can be seen in Figure 4 of the appendix.

4 Reading and Writing Data using APDU Command Console

Another method of reading and writing data to the card is using the APDU console to send read and write commands. The APDU command console looks as follows:



Active Reader: ACS ACR1281 1S Dual Reader PICC 0

APDU Command

CLA	INS	P1	P2	Lc	Le
FF	B0	00	04	00	04

SW1	SW2
90	00

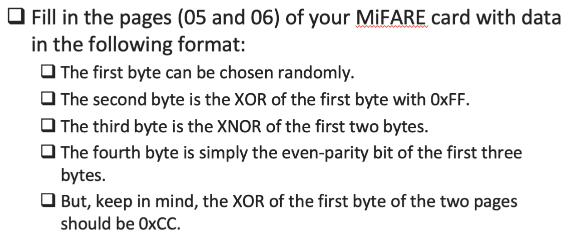
0 - The operation completed successfully

The APDU command has the following format: <CLA>< INS><P1><P2><Lc><Le>. INS is where you indicate if you want to write data (D6) or read data (B4). Lc is the write command, indicating how many bytes you want to write. Le is the read command, indicating how many bytes you want to write to a block. When writing to a block, we fill in the data we want to write to it in the “Data in Parameters” section.

The execution of the APDU Command Console can be seen in Figure 5 in the appendix.

5 Error Detection Demonstration

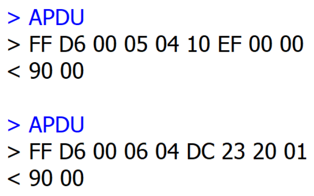
In this activity, we had to write specific byte patterns to certain pages of the card to illustrate how XOR and XNOR checks and detect a single bit flip, and hence detect errors or tampering attempts. The activity is described as follows:



- ❑ Fill in the pages (05 and 06) of your MiFARE card with data in the following format:
  - ❑ The first byte can be chosen randomly.
  - ❑ The second byte is the XOR of the first byte with 0xFF.
  - ❑ The third byte is the XNOR of the first two bytes.
  - ❑ The fourth byte is simply the even-parity bit of the first three bytes.
  - ❑ But, keep in mind, the XOR of the first byte of the two pages should be 0xCC.

The write instruction for Page 5 is as follows:

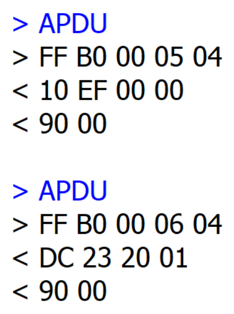
- Our Implementation:
  - APDU Command: FF D6 00 05 04 00
  - Data In: 10 EF 00 00
  - Final Command: FF D6 00 05 04 10 EF 00 00
- Guided Implementation:
  - First Byte: 0xAA
  - Second Byte: 0xAA XOR 0xFF = 0x55 (Inverse of the First Byte)
  - Third Byte: 0xAA XNOR 0x55 = 0x00 (Always result in a 0)
  - Fourth Byte: 0x00 because the total number of 1s is even
  - First Byte: x number of 1s
  - Second Byte: x - x number of 1s
  - Add First Byte and Second Byte: x number of 1s



```
> APDU
> FF D6 00 05 04 10 EF 00 00
< 90 00

> APDU
> FF D6 00 06 04 DC 23 20 01
< 90 00
```

The write instruction for Page 6 is as follows:

- Our Implementation:
    - APDU Command: FF D6 00 06 04 00
    - Data In: DC 23 20 01
    - Final Command: FF D6 00 06 04 DC 23 20 01
- Guided Implementation:
    - Y = 0xAA XOR 0xCC = 0x66
    - Second Byte: 0x99
    - Third Byte: 0x00
    - Fourth Byte: 0x00



```
> APDU
> FF B0 00 05 04
< 10 EF 00 00
< 90 00

> APDU
> FF B0 00 06 04
< DC 23 20 01
< 90 00
```

Key Lessons Learnt from Error Detection:

- Everything is dependent on the first byte of page 5.
- Can be used for error detection!
    - ie 1 bit actually flipped on the first byte on Page 5 from AA to AB.
    - XOR the first 2 bytes, see if it is equal to 0 on the third byte.
        - If not 0x00, some corruption has happened.
    - The first byte of Page 5 and Page 6 is then XORed to get 0xCC. If it is not 0xCC also means Error Detected on the first byte of either Page 5 or Page 6.
    - Error detection is different from error correction; recovering from that error is different.

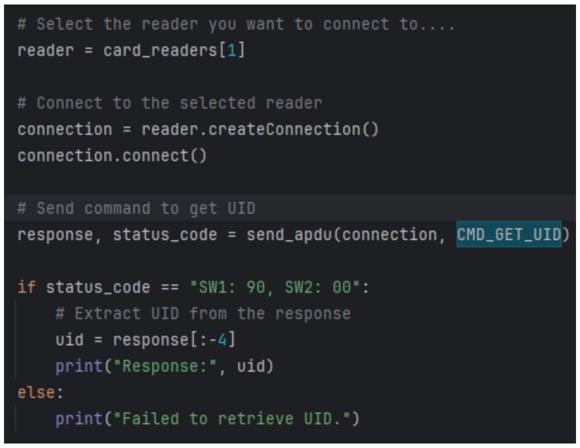
3.2 Reading Purse Balance and Transaction Logs on NETS FlashPay (CEPAS)

For Contactless E-Purse Application Specification (CEPAS) cards like NETS Flashpay in Singapore, specialised commands exist to retrieve the stored balance and transaction history. We used the APDU Plus tool to send commands using a GUI. To read the purse balance, we used the following APDU command: <u>90 32 03 00 00</u>. The details of the commands can be seen in Figure 6. The response returned is a 62-byte response shown in Figure 7 of the Appendix, where the wallet purse balance is contained in bytes 3 to 5. To read the transaction logs, we used the following APDU command: <u>90 32 03 00 01 00 00.</u> Each transaction returned is 16 bytes. The full byte representation of the response can be seen in Figure 8 of the Appendix.

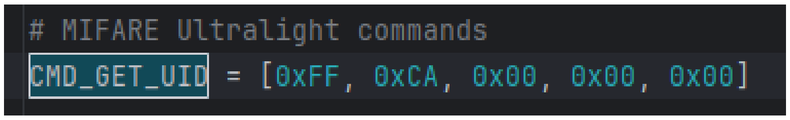
While the APDU Plus GUI tool provided a user-friendly interface for sending APDU commands, we used Python scripts for automation and deeper customisation. This included converting the bytes of the purse balance and transaction logs to human-readable data for analysis.

<u>1 Python Script to Connect to NETS FlashPay Card</u>

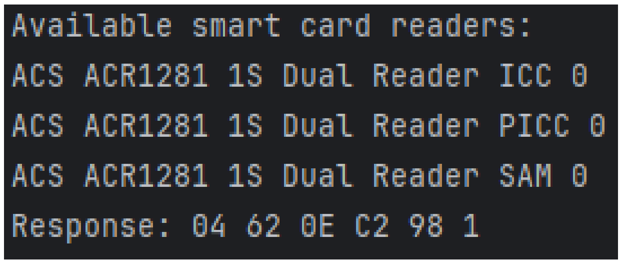
We used the pyscard library to locate available smart card readers and establish a connection with them. We implemented to following Python scripts to get the card UID and establish a connection with the card:



```
# Select the reader you want to connect to....
reader = card_readers[1]

# Connect to the selected reader
connection = reader.createConnection()
connection.connect()

# Send command to get UID
response, status_code = send_apdu(connection, CMD_GET_UID)

if status_code == "SW1: 90, SW2: 00":
    # Extract UID from the response
    uid = response[:-4]
    print("Response:", uid)
else:
    print("Failed to retrieve UID.")
```

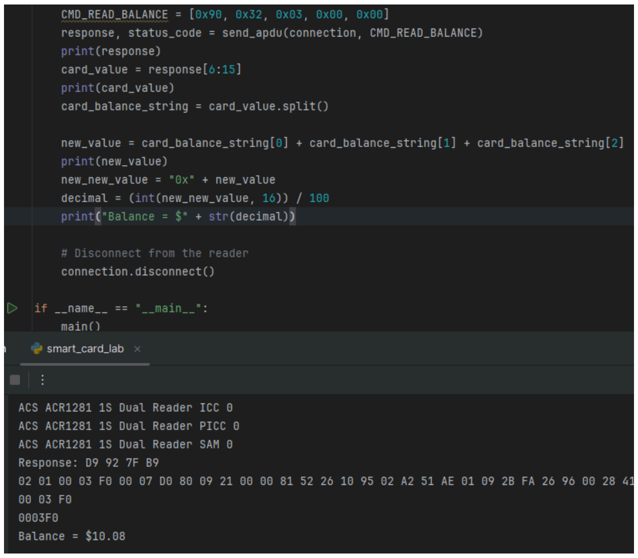


```
# MIFARE Ultralight commands
CMD_GET_UID = [0xFF, 0xCA, 0x00, 0x00, 0x00]
```



```
Available smart card readers:
ACS ACR1281 1S Dual Reader ICC 0
ACS ACR1281 1S Dual Reader PICC 0
ACS ACR1281 1S Dual Reader SAM 0
Response: 04 62 0E C2 98 1
```

2 Python Scripts to Read NETS FlashPay Card Balance



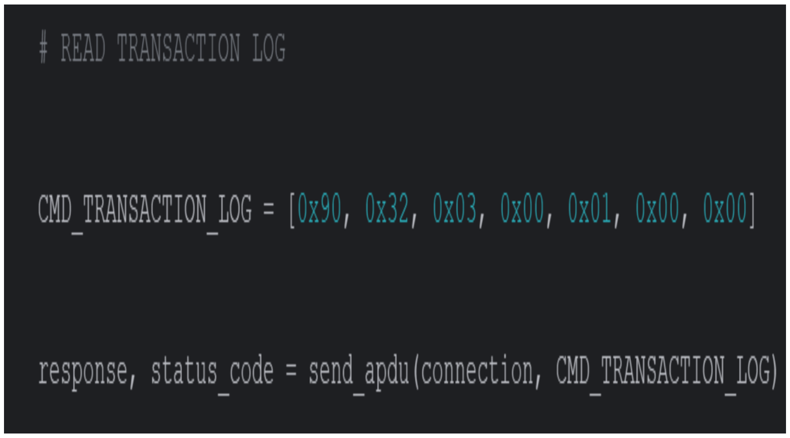
```
    CMD_READ_BALANCE = [0x90, 0x32, 0x03, 0x00, 0x00]
    response, status_code = send_apdu(connection, CMD_READ_BALANCE)
    print(response)
    card_value = response[6:15]
    print(card_value)
    card_balance_string = card_value.split()

    new_value = card_balance_string[0] + card_balance_string[1] + card_balance_string[2]
    print(new_value)
    new_new_value = "0x" + new_value
    decimal = (int(new_new_value, 16)) / 100
    print("Balance = $" + str(decimal))

    # Disconnect from the reader
    connection.disconnect()

if __name__ == "__main__":
    main()
```

```
ACS ACR1281 1S Dual Reader ICC 0
ACS ACR1281 1S Dual Reader PICC 0
ACS ACR1281 1S Dual Reader SAM 0
Response: D9 92 7F B9
02 01 00 03 F0 00 07 D0 80 09 21 00 00 81 52 26 10 95 02 A2 51 AE 01 09 2B FA 26 96 00 28 41
00 03 F0
0003F0
Balance = $10.08
```

To read the balance, we used the send_apdu function, where the argument of the command to be sent was defined as follows: CMD_READ_BALANCE = [0x90, 0x32, 0x03, 0x00, 0x00]. We then printed the response from the command and converted the byte response into a human-readable format. We found out that the balance of the card was $10.08.

Data object	Length	Security
CEPAS version	1 byte	None
Purse status	1 byte	None
Purse balance	3 bytes	None

3 Python Scripts to Read NETS FlashPay Card Transaction Logs



```
# READ TRANSACTION LOG

CMD_TRANSACTION_LOG = [0x90, 0x32, 0x03, 0x00, 0x01, 0x00, 0x00]

response, status_code = send_apdu(connection, CMD_TRANSACTION_LOG)
```

To read the transaction log, we used the send_apdu function, where the argument of the command to be sent was defined as follows: CMD_TRANSACTION_LOG = [0x90, 0x32, 0x03, 0x00, 0x01, 0x00, 0x00]. Refer to Figure 9 in the Appendix for full code implementation.

We also converted the byte response of the transaction log to a human-readable format, illustrated in Figure 10 in the Appendix.

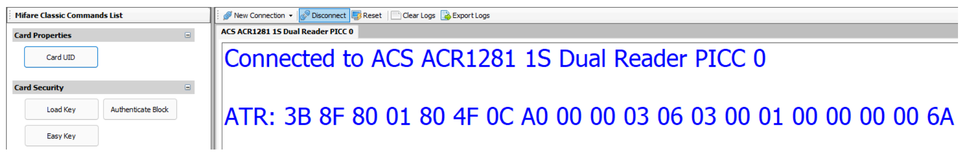
3.3 MIFARE Classic Smart Card

Information:

- 16 Sectors of 4 Blocks, each block of 16 Bytes (Figure 11 in Appendix)
- Total Bytes: 16*4*16 = 1024 Bytes (1KB Memory)
- Can only write data to the first three blocks (48 Bytes)
- The 4th block (trailer block) contains the key for authentication (16 Bytes)
- Key A is 6 Bytes (48 Bits), needs to brute force 2^48 times (a few years)

3.3.1 Steps, Procedures and Explanations

1 Answer-To-Reset (ATR)



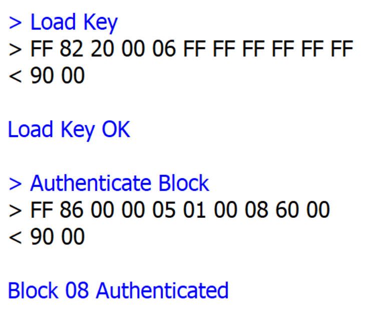
```
Connected to ACS ACR1281 1S Dual Reader PICC 0

ATR: 3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 01 00 00 00 00 6A
```

2 Read & Write from Sector 1

1. Need to authenticate first
   1.1. Load keys to the card reader first
      1.1.1. Key Structure: 20
      1.1.2. Key Number: 00 (store up to 32 different keys)
      1.1.3. Key to Load: FF FF FF FF FF FF(default keys)
   1.2. Authenticate Block 4 (aka Sector 1, Block 0)
      1.2.1. ie Blocks 4, 5, 6, 7 all have the same key because they belong to the same Sector
      1.2.2. Target Block: 04
      1.2.3. Key Number: 00
      1.2.4. Key Type: Key A
   1.3. Authenticate Block 8 (aka Sector 2, Block 0)
      1.3.1. Permission to Read Blocks 8, 9, 10, 11
      1.3.2. Target Block: 08
      1.3.3. Key Number: 00
      1.3.4. Key Type: Key A
   1.4. Each time want to access a new Sector (of 4 Blocks), you need to first authenticate that sector.
   1.5. Authenticating 1 sector allows you to access the corresponding 4 Blocks
2. Read Block

2.1. Block to Read: 08
2.2. Length: 10 (0x10 means can only read 16 bytes at a time)

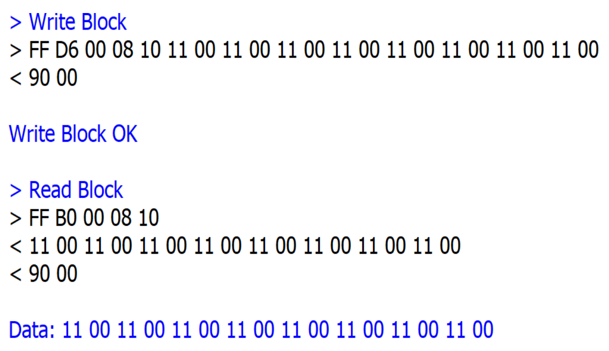
3 Load key and Authenticate Block



```
> Load Key
> FF 82 20 00 06 FF FF FF FF FF FF
< 90 00

Load Key OK

> Authenticate Block
> FF 86 00 00 05 01 00 08 60 00
< 90 00

Block 08 Authenticated
```

Load the key into the reader's memory using the APDU command to store the default key: FF FF FF FF FF FF. Then, we authenticate a given block using key A or B. This demonstrates how the trailer block of each sector contains the keys and access bits. It also illustrates how critical it is to authenticate each sector before reading or writing data, unlike MIFARE Ultralight, MIFARE Classic enforces sector-based access control.

4 Write to block (store 16-byte value)



```
> Write Block
> FF D6 00 08 10 11 00 11 00 11 00 11 00 11 00 11 00 11 00 11 00
< 90 00

Write Block OK

> Read Block
> FF B0 00 08 10
< 11 00 11 00 11 00 11 00 11 00 11 00 11 00 11 00
< 90 00

Data: 11 00 11 00 11 00 11 00 11 00 11 00 11 00 11 00
```

After authenticating a sector, we can read and write to and from its blocks. To read, we use B,0 and to write, we use D6 in the APDU command as the third and fourth bytes.

5 Store as value (storing 0000 as value) and Increment values (by 1)

In this section, we convert a standard data block into a value block by following a specific format. This helps to introduce atomic operations in the smart card, which reduces the risk of partial updates. The procedure used to convert a data block into a value block can be seen in Figure 12 in the appendix.

6 Trailer Block and Access Bits

Access Bits: Access Information determines R/W Access Permissions to that particular sector

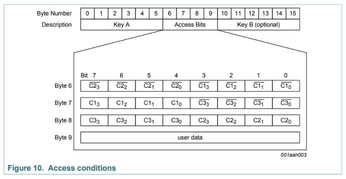
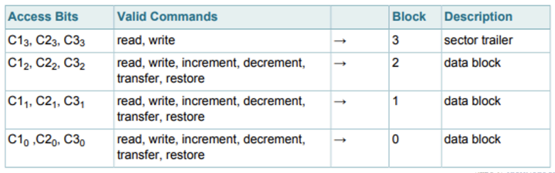
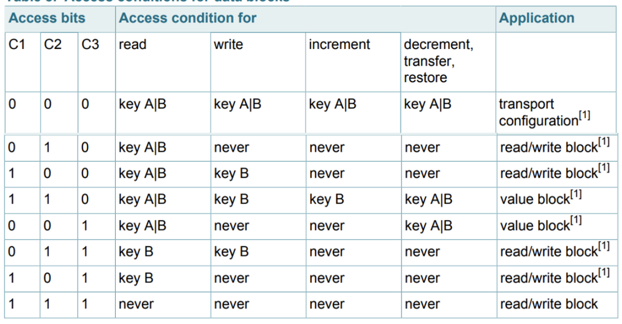


Figure 10. Access conditions



Access Bits	Valid Commands		Block	Description
$C1_3$, $C2_3$, $C3_3$	read, write	→	3	sector trailer
$C1_2$, $C2_2$, $C3_2$	read, write, increment, decrement, transfer, restore	→	2	data block
$C1_1$, $C2_1$, $C3_1$	read, write, increment, decrement, transfer, restore	→	1	data block
$C1_0$,$C2_0$, $C3_0$	read, write, increment, decrement, transfer, restore	→	0	data block

Table 8. Access conditions for data blocks



Access bits			Access condition for				Application
C1	C2	C3	read	write	increment	decrement, transfer, restore	
0	0	0	key A|B	key A|B	key A|B	key A|B	transport configuration[1]
0	1	0	key A|B	never	never	never	read/write block[1]
1	0	0	key A|B	key B	never	never	read/write block[1]
1	1	0	key A|B	key B	key B	key A|B	value block[1]
0	0	1	key A|B	never	never	key A|B	value block[1]
0	1	1	key B	key B	never	never	read/write block[1]
1	0	1	key B	never	never	never	read/write block[1]
1	1	1	never	never	never	never	read/write block

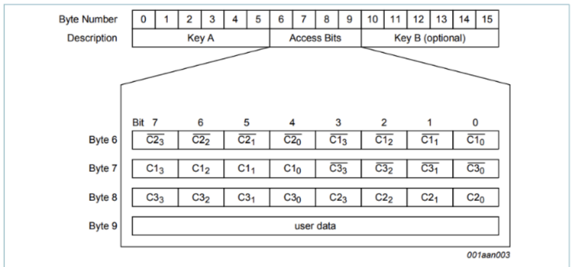
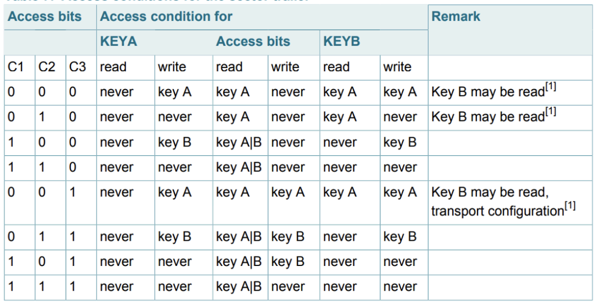


Figure 10. Access conditions

Table 7. Access conditions for the sector trailer



Access bits			Access condition for						Remark
			KEYA		Access bits		KEYB		
C1	C2	C3	read	write	read	write	read	write	
0	0	0	never	key A	key A	never	key A	key A	Key B may be read[1]
0	1	0	never	never	key A	never	key A	never	Key B may be read[1]
1	0	0	never	key B	key A|B	never	never	key B	
1	1	0	never	never	key A|B	never	never	never	
0	0	1	never	key A	key A	key A	key A	key A	Key B may be read, transport configuration[1]
0	1	1	never	key B	key A|B	key B	never	key B	
1	0	1	never	never	key A|B	key B	never	never	
1	1	1	never	never	key A|B	never	never	never	

Calculate Access Bits for the Following Condition:

- ❑ **Data Block 0**: Read by Key A/B, Cannot be written. Cannot be incremented. But, can be decremented using key A/B
- ❑ **Data Block 1**: Read by Key A/B, Cannot be written, incremented or decremented
- ❑ **Data Block 2**: Read and written only by Key B, cannot be incremented or decremented
  - ❑ **Key A**: Never Read, Write using Key B
  - ❑ **Key B**: Never Read, Write using Key B
  - ❑ **Access Bits**: Read using Key A/B, Write using Key B

Data Block 0:

- C1, C2, C3 = 0, 0, 1

Data Block 1:

- C1, C2, C3 = 0, 1, 0

Data Block 2:

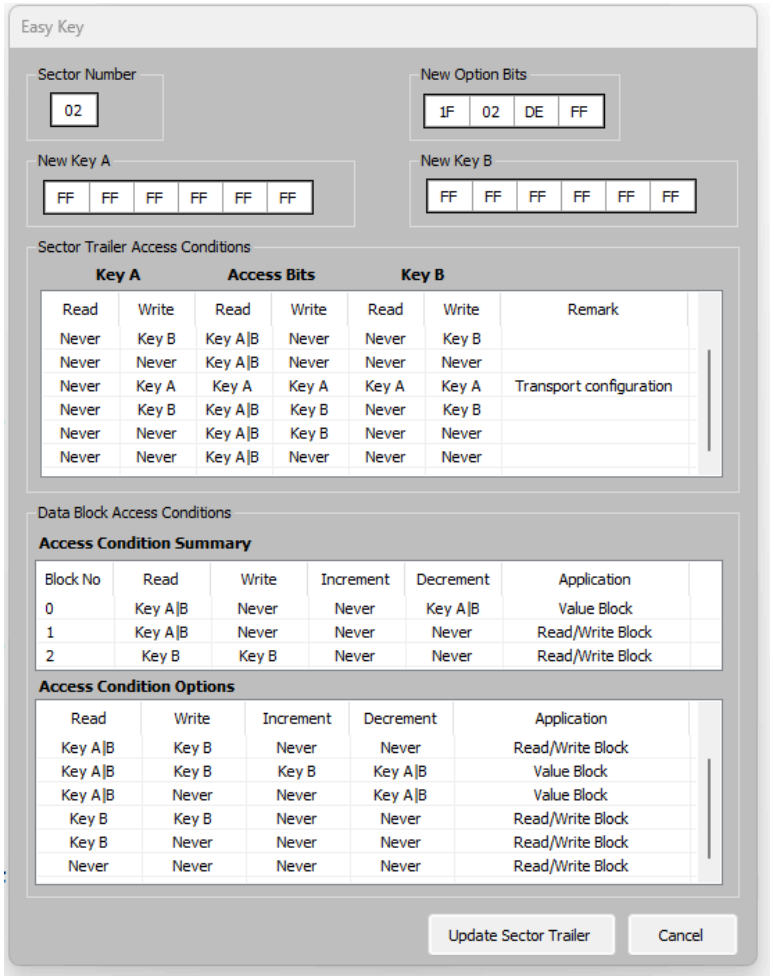
- C1, C2, C3 = 0, 1, 1

Data Block 3 (Sector Trailer):

- C1, C2, C3 = 0, 1, 1

Overall:
C13, C23, C33 = 0, 1, 1
C12, C22, C32 = 0, 1, 1
C11, C21, C31 = 0, 1, 0
C10, C20, C30 = 0, 0, 1

Manual Calculation: Same as the "New Option Bits" in the Easy Key
Byte 6 = 0 0 0 1 1 1 1 1 = 1F
Byte 7 = 0 0 0 0 0 0 1 0 = 02
Byte 8 = 1 1 0 1 1 1 1 0 = DE



Sector Number: 02

New Option Bits: 1F 02 DE FF

New Key A: FF FF FF FF FF FF

New Key B: FF FF FF FF FF FF

Sector Trailer Access Conditions

Key A		Access Bits		Key B		
Read	Write	Read	Write	Read	Write	Remark
Never	Key B	Key A|B	Never	Never	Key B	
Never	Never	Key A|B	Never	Never	Never	
Never	Key A	Key A	Key A	Key A	Key A	Transport configuration
Never	Key B	Key A|B	Key B	Never	Key B	
Never	Never	Key A|B	Key B	Never	Never	
Never	Never	Key A|B	Never	Never	Never	

Data Block Access Conditions

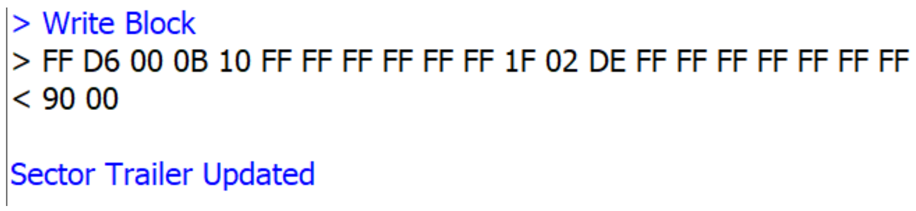
Access Condition Summary

Block No	Read	Write	Increment	Decrement	Application
0	Key A|B	Never	Never	Key A|B	Value Block
1	Key A|B	Never	Never	Never	Read/Write Block
2	Key B	Key B	Never	Never	Read/Write Block

Access Condition Options

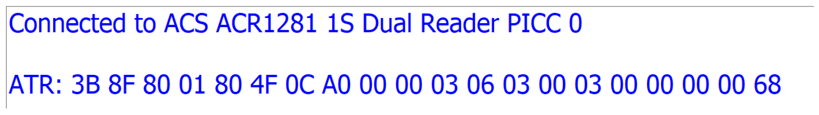
Read	Write	Increment	Decrement	Application
Key A|B	Key B	Never	Never	Read/Write Block
Key A|B	Key B	Key B	Key A|B	Value Block
Key A|B	Never	Never	Key A|B	Value Block
Key B	Key B	Never	Never	Read/Write Block
Key B	Never	Never	Never	Read/Write Block
Never	Never	Never	Never	Read/Write Block

Sector trailer


```
> Write Block
> FF D6 00 0B 10 FF FF FF FF FF FF 1F 02 DE FF FF FF FF FF FF FF
< 90 00

Sector Trailer Updated
```

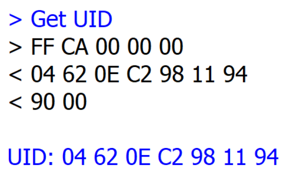
Appendix

Figure 1: ATR of Mifare Ultralight Smart Card



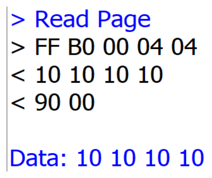
```
Connected to ACS ACR1281 1S Dual Reader PICC 0

ATR: 3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 03 00 00 00 00 68
```

Figure 2: Reading Mifare Ultralight Card UID



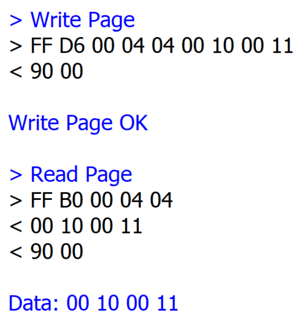
```
> Get UID
> FF CA 00 00 00
< 04 62 0E C2 98 11 94
< 90 00

UID: 04 62 0E C2 98 11 94
```

Figure 3: Reading 4 Bytes in Block 4 of Mifare Ultralight



```
> Read Page
> FF B0 00 04 04
< 10 10 10 10
< 90 00

Data: 10 10 10 10
```

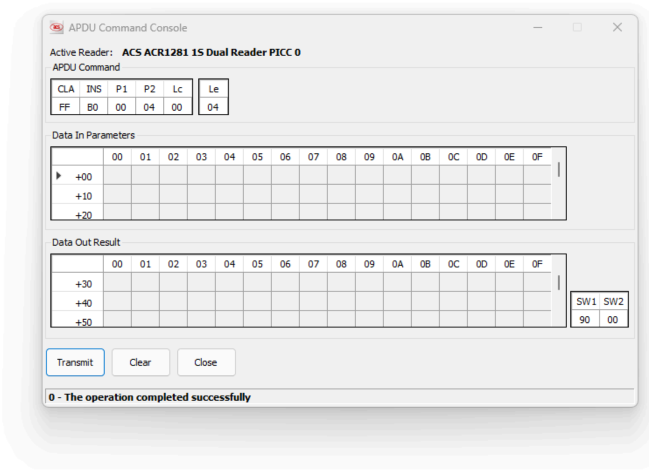
Figure 4: Writing 4 Bytes of Data to Block 4 of Mifare Ultralight



```
> Write Page
> FF D6 00 04 04 00 10 00 11
< 90 00

Write Page OK

> Read Page
> FF B0 00 04 04
< 00 10 00 11
< 90 00

Data: 00 10 00 11
```

Figure 5: Reading and Writing to Card using APDU Console

```
> APDU
> FF B0 00 04 04
< 00 10 00 11
< 90 00

> APDU
> FF D6 00 04 04 11 00 11 00
< 90 00

> APDU
> FF B0 00 04 04
< 11 00 11 00
< 90 00
```



Active Reader: **ACS ACR1281 1S Dual Reader PICC 0**

APDU Command

CLA	INS	P1	P2	Lc	Le
FF	B0	00	04	00	04

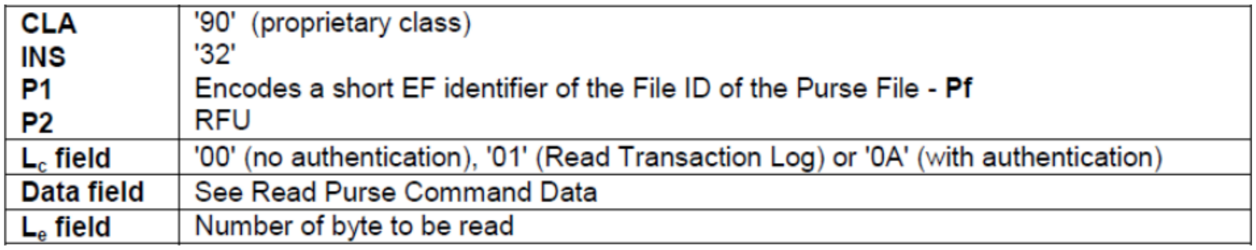
Data In Parameters

Data Out Result

SW1	SW2
90	00

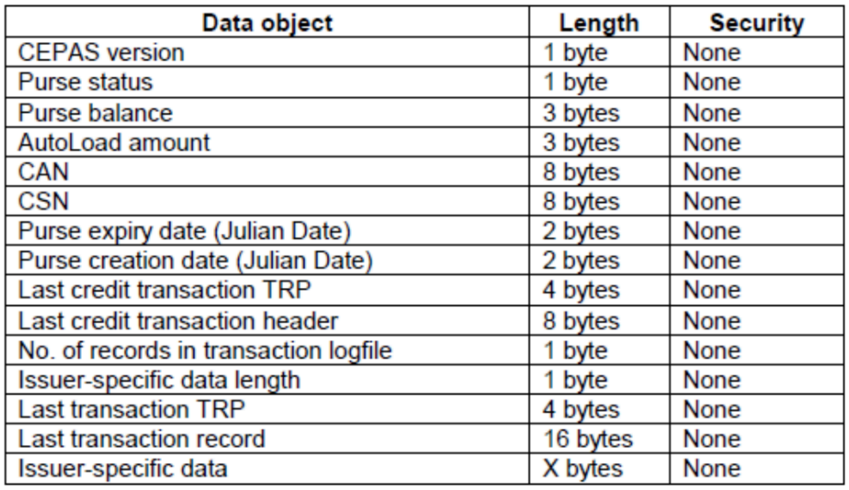
0 - The operation completed successfully

Figure 6: APDU Command for Reading CEPAS Purse Balance



CLA **INS** **P1** **P2**	'90' (proprietary class) '32' Encodes a short EF identifier of the File ID of the Purse File - **Pf** RFU
L_c field	'00' (no authentication), '01' (Read Transaction Log) or '0A' (with authentication)
Data field	See Read Purse Command Data
L_e field	Number of byte to be read

Figure 7: APDU Response Format for Purse Balance



Data object	Length	Security
CEPAS version	1 byte	None
Purse status	1 byte	None
Purse balance	3 bytes	None
AutoLoad amount	3 bytes	None
CAN	8 bytes	None
CSN	8 bytes	None
Purse expiry date (Julian Date)	2 bytes	None
Purse creation date (Julian Date)	2 bytes	None
Last credit transaction TRP	4 bytes	None
Last credit transaction header	8 bytes	None
No. of records in transaction logfile	1 byte	None
Issuer-specific data length	1 byte	None
Last transaction TRP	4 bytes	None
Last transaction record	16 bytes	None
Issuer-specific data	X bytes	None

Figure 8: APDU Response Format for Transaction Logs

Transaction log record format			
Transaction header			Transaction user data
Transaction type	Transaction amount	Transaction DateTime	
1 byte	3 bytes	4 bytes	8 bytes

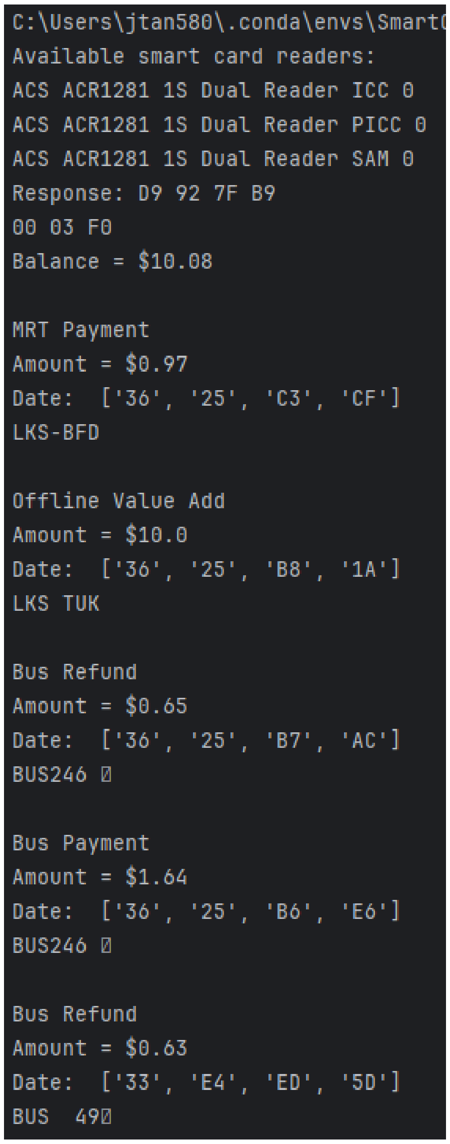
Figure 9: Python Code Implementation for NETS Flashpay Card

```
import re
from smartcard.System import readers
from smartcard.util import toHexString
# MIFARE Ultralight commands

CMD_GET_UID = [0xFF, 0xCA, 0x00, 0x00, 0x00]

CMD_READ_BLOCK = [0xFF, 0xB0, 0x00, 0x04, 0x04]

CMD_GET_TRANSACTION_LOG = None

# Function to send APDU commands to the card

def send_apdu(connection, apdu_cmd):

   data, sw1, sw2 = connection.transmit(apdu_cmd)

   response = toHexString(data)

   status_code = "SW1: {:02X}, SW2: {:02X}".format(sw1, sw2)

   return response, status_code

def main():

   # Get all available smart card readers

   card_readers = readers()

   if not card_readers:
       print("No smart card readers found.")

       return

   print("Available smart card readers:")

   for reader in card_readers:

       print(reader)

   # Select the reader you want to connect to....

   reader = card_readers[1]

   # Connect to the selected reader

   connection = reader.createConnection()

   connection.connect()

   # Send command to get UID

   response, status_code = send_apdu(connection, CMD_GET_UID)

   if status_code == "SW1: 90, SW2: 00":

       # Extract UID from the response

       uid = response[:]

       print("Response:", uid)

   else:

       print("Failed to retrieve UID.")

   # READ BALANCE

   CMD_READ_BALANCE = [0x90, 0x32, 0x03, 0x00, 0x00]
```

```
   response, status_code = send_apdu(connection, CMD_READ_BALANCE)

   # print(response)

   card_value = response[6:15]

   print(card_value)

   card_balance_string = card_value.split()

   new_value = card_balance_string[0] + card_balance_string[1] +
card_balance_string[2]

   # print(new_value)

   new_new_value = "0x" + new_value

   decimal = (int(new_new_value, 16)) / 100

   print("Balance = $" + str(decimal))

   print("")

   # READ TRANSACTION LOG

   CMD_TRANSACTION_LOG = [0x90, 0x32, 0x03, 0x00, 0x01, 0x00, 0x00]

   response, status_code = send_apdu(connection, CMD_TRANSACTION_LOG)

   # print(response)

   transaction_data_string = response.split()

   chunk_size = 16

   transactions = []

   for i in range(0, len(transaction_data_string), 16):
       transaction = transaction_data_string[i:i+16]

       transactions.append(transaction)

   # print(transactions)

   for i in transactions:

       transaction_type = i[0]

       transaction_amount = i[1:4]

       # print(transaction_amount)

       if transaction_type == "75":

           print("Offline Value Add")

           transaction_amount_combined = transaction_amount[0] +
transaction_amount[1] + transaction_amount[2]

           transaction_amount_hex_string = "0x" + transaction_amount_combined

           decimal = (int(transaction_amount_hex_string, 16)) / 100

           print("Amount = $" + str(decimal))

       elif transaction_type == "76":

           print("Bus Refund")

           transaction_amount_combined = transaction_amount[0] +
transaction_amount[1] + transaction_amount[2]

           transaction_amount_hex_string = "0x" + transaction_amount_combined

           decimal = (int(transaction_amount_hex_string, 16)) / 100

           print("Amount = $" + str(decimal))
       elif transaction_type == "31":
           print("Bus Payment")
           transaction_amount_combined = transaction_amount[0] +
transaction_amount[1] + transaction_amount[2]
           transaction_amount_string = "" + transaction_amount_combined
           num = int(transaction_amount_string, 16)
           bit_length = len(transaction_amount_string) * 4
           mask = (1 << bit_length) - 1
           twocomp = ((~num) & mask) + 1
           result = format(twocomp, f'0{len(transaction_amount_string)}X')
           decimal = (int(result, 16)) / 100
           print("Amount = $" + str(decimal))
       elif transaction_type == "30":
           print("MRT Payment")
           transaction_amount_combined = transaction_amount[0] +
transaction_amount[1] + transaction_amount[2]
           transaction_amount_string = "" + transaction_amount_combined
           num = int(transaction_amount_string, 16)
           bit_length = len(transaction_amount_string) * 4
           mask = (1 << bit_length) - 1
           twocomp = ((~num) & mask) + 1
```

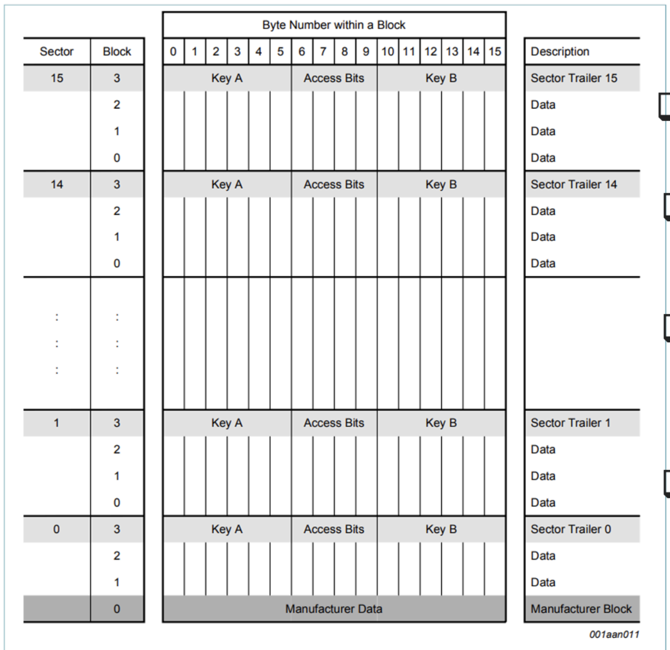
```python
            result = format(twocomp, f'0{len(transaction_amount_string)}X')
            decimal = (int(result, 16)) / 100
            print("Amount = $" + str(decimal))
        transaction_datetime = i[4:8]
        print("Date: ", transaction_datetime)
        transaction_userdata = i[8:17]
        text = []
        for i in transaction_userdata:
            character_string = "" + i
            new_data = bytes.fromhex(character_string)
            ascii_text = new_data.decode('ascii')
            text.append(ascii_text)
        final_string = ''.join(text)
        print(final_string)
        text = []
        # print(transaction_userdata)
        print()
    # Disconnect from the reader
    connection.disconnect()
if __name__ == "__main__":
  main()
```

Figure 10: Transaction Log Output of NETS Flashpay Card



```
C:\Users\jtan580\.conda\envs\Smart
Available smart card readers:
ACS ACR1281 1S Dual Reader ICC 0
ACS ACR1281 1S Dual Reader PICC 0
ACS ACR1281 1S Dual Reader SAM 0
Response: D9 92 7F B9
00 03 F0
Balance = $10.08

MRT Payment
Amount = $0.97
Date:  ['36', '25', 'C3', 'CF']
LKS-BFD

Offline Value Add
Amount = $10.0
Date:  ['36', '25', 'B8', '1A']
LKS TUK

Bus Refund
Amount = $0.65
Date:  ['36', '25', 'B7', 'AC']
BUS246 ▯

Bus Payment
Amount = $1.64
Date:  ['36', '25', 'B6', 'E6']
BUS246 ▯

Bus Refund
Amount = $0.63
Date:  ['33', 'E4', 'ED', '5D']
BUS  49▯
```

Figure 11: MIFARE Classic Memory Organisation/Layout



Sector	Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
		Byte Number within a Block																
15	3	Key A						Access Bits				Key B						Sector Trailer 15
	2																	Data
	1																	Data
	0																	Data
14	3	Key A						Access Bits				Key B						Sector Trailer 14
	2																	Data
	1																	Data
	0																	Data
:	:																	
:	:																	
:	:																	
1	3	Key A						Access Bits				Key B						Sector Trailer 1
	2																	Data
	1																	Data
	0																	Data
0	3	Key A						Access Bits				Key B						Sector Trailer 0
	2																	Data
	1																	Data
	0	Manufacturer Data																Manufacturer Block

Figure 5. Memory organization

- ❑ Each sector has four blocks
- ❑ First three blocks within a sector are data blocks.
- ❑ Fourth block in a sector is the trailer block.
- ❑ Trailer block stores two keys and the access rights of the block.

Figure 12: Convert Data Block Into a Value Block

```
> Write Block
> FF D6 00 09 10 00 00 00 00 FF FF FF FF 00 00 00 00 09 F6 09 F6
< 90 00

Write Block OK

> Read Value
> FF B1 00 09 04
< 00 00 00 00
< 90 00

Value Read: 00 00 00 00

> Increment Value Block
> FF D7 00 09 05 01 00 00 00 01
< 90 00

Increment Value Block OK

> Read Value
> FF B1 00 09 04
< 00 00 00 01
< 90 00

Value Read: 00 00 00 01
```