NXP MIFARE DESFire Protocol

This document is reverse engineered from various sources and may not be 100% correct. It seems the official protocol is subject to an NDA - if this is not the case, please do let me know! Obviously there is no warranty for any error or mistake I have made.

Sources

Various testing and experimentation by me (@TheRealRevK), plus blogs and code including :-

- https://github.com/nfc-tools/libfreefare/blob/master/libfreefare/mifare_desfire.c (This has actual code that works, and so is a primary source of this manual).
- https://scancode.ru/upload/iblock/de4/mifare_application_programming_guide_for_desfire_rev.e.pdf (This has lots of details but different command codes, useful clues though)
- https://hack.cert.pl/files/desfire-9f122c71e0057d4f747d2ee295b0f5f6eef8ac32.html (some useful examples)
- https://ridrix.wordpress.com/tag/desfire-protocol/ (more useful examples)
- http://www.ti.com/lit/an/sloa213/sloa213.pdf (explains more on authentication)
- https://www.nxp.com/docs/en/data-sheet/MF3ICDQ1_MF3ICDHQ1_SDS.pdf

Overview

It is useful to understand what these cards/fobs can do for you!

The card itself has some top level settings, but the idea is that the card can be used with one or more *applications*. This means the same card can be used for travel, access control, currency, all sorts. Each application is independent of the others on the card and is identified by a 3 byte application id (AID). So you select the application you want, and then you can access *files* that are stored relating to that application. In practice this sharing of applications may not happen generally as someone will *own* the card, but it can in theory. With access control I have each site as an application and so one card can work multiple sites independently, which is convenient.

The files within an application are small data storage areas, and there are actually several types of file. The card itself has a total capacity (e.g. 4k), but typically the data stored is small, such as a value or identity. The different types of files include values (e.g. points or currency), or simple binary data, or records of fixed length. One of the clever features is a record file that automatically overwrites the oldest entry when full, ideal for some sort of log of events. There is also a system to ensure integrity (e.g. backup files, and a commit stage after setting up changes).

The files themselves have access permissions controlling read and write and change. The access control is done using *keys*. Each application can have one or more keys. You select the application and then authenticate with a key. Having done this, the access rights you have depend on what key you used. There are also application level access controls and controls on the top level card itself. Each application can have unto 14 keys (a master key and up to 13 others), and the top level card has a master key as well. These keys can use DES, 3DES, or AES encryption. This manual only covers AES. Note, only symmetric authentication exists, i.e. card and reader need same key, and there is no support for public key where one side can check the other without knowing its secret key. This is a security consideration, as extracting a key from a reader or code means you can make cards that match that key, so consider not storing keys in flash/EEPROM, etc.

The card also allows encryption of the communications, so you cannot snoop data passively when a card is being used.

There are also a number of different types of cards with different storage and features. I am testing using the DESFire EV1 4k cards.

Communications

I am assuming you have a way to talk to the card. I am using a PN532 which handles the RATS and anti collision and selecting the card and so on, and gives me a 4 or 7 byte card ID. I can then do an `InDataExchange` which allows me to send a command sequence of bytes and get a response sequence of bytes.

Each command starts with a one byte command code, and may have a number of data bytes following depending on the command. The simplest command is one byte only, e.g. `60` gets version details of the card. I will be quoting bytes using hex within this document, so that is `0x60`.

Each response starts with a one byte status code, and may have a number of data bytes following depending on the command and response. The simplest response is one byte, and `00` means success.

There is a way to wrap these commands in an ISO message format, and to perform ISO equivalent versions of commands, which may be better. But this document is trying to cover the native DESFire commands and how they work.

It is important to also realise that there is a whole process to authenticate or encrypt the message and response payloads, explained later. This is important to ensure communications cannot be snooped on.

Command List

As you can see, many of these command codes are designed to be mnemonic, albeit somewhat convoluted in some cases.

Security related commands		
AA	Authenticate (AES)	Start the authentication process for a key, using AES
1A	Authenticate (ISO)	Start the authentication process for a key, using 3DES or 3K3DES
0A	Authenticate (Legacy)	Start the authentication process for a key, using simple DES
54	Change KeySettings	Change the settings for a key
5C	Set Configuration	Card level configuration
C4	Change Key	Change a key
64	Get Key Version	Returns a key version byte.

Card level commands		
CA	Create Application	Create a new application
DA	Delete Application	Delete an application
6A	Get Applications IDs	Get a list of application IDs
6E	Free Memory	Get free memory details
6D	GetDFNames	Get the data file names
45	Get KeySettings	Get details of a keys settings
5A	Select Application	Select application
FC	FormatPICC	Format the card
60	Get Version	Get version details for card
51	GetCardUID	Get the read ID for the card (can be set so a random ID is used as part of collision detection, rather than the real ID).

Application level commands		
6F	Get FileIDs	Get a list of file IDs
61	Get FileIDs (ISO)	Get a list of ISO file IDs
F5	Get FileSettings	Get file settings for a specific existing file
5F	Change FileSettings	Change file settings for a specific existing file
CD	Create StdDataFile	Creates a file for arbitrary binary data
CB	Create BackupDataFile	Creates a file for arbitrary binary data but with a commit process so changes apply reliably all in one go

Application level commands		
CC	Create ValueFile	Creates a file to hold a 32 bit value
C1	Create LinearRecordFile	Create a file to allow records of fixed size to be added until full
C0	Create CyclicRecordFile	Create a file to allow records of fixed size to be added, clearing the oldest record automatically - ideal for a history or a log
DF	DeleteFile	Delete a file

Data manipulations commands		
BD	Read Data	Read data from standard or backup file
3D	Write Data	Write data to standard or backup file (write to backup only happens when commit is done)
6C	Get Value	Get the value from a value file
0C	Credit	Increase the value in a value file
DC	Debit	Decrease the value in a value file
1C	Limited Credit	Increase the value in a value file without having full permissions to that file, up to a limit
3B	Write Record	Write a record to a linear or cyclic record file
BB	Read Records	Read records from a linear or cyclic record file
EB	Clear RecordFile	Clear a linear or cyclic record file
C7	Commit Transaction	Commit writes to backup, value, or record files
A7	Abort Transaction	Invalid writes to backup, value, or record files

Status codes

Status codes	
`00`	Success
`0C`	No change
`0E`	Out of EEPROM
`1C`	Illegal command
`1E`	Integrity error
`40`	No such key
`6E`	Error (ISO?)
`7E`	Length error
`97`	Crypto error
`9D`	Permission denied
`9E`	Parameter error
`A0`	Application not found
`AE`	Authentication error
`AF`	Additional frame (more data to follow before final status code)
`BE`	Boundary error
`C1`	Card integrity error
`CA`	Command aborted
`CD`	Card disabled
`CE`	Count error
`DE`	Duplicate error
`EE`	EEPROM error
`F0`	File not found
`F1`	File integrity error

Detailed commands

Note that data values such as offsets and record counts and access rights, etc, are all sent low byte first.

AA: Authenticate (AES)

This allows you to authenticate for a specified key number using AES. If the key is not AES you get an AE error back. The key number is the key in the current application. If top level (AID 0) selected or no application selected yet then this is the master key for the card (key 0).

AA	Key ID

The response is AF with 16 bytes of data (random).

AF	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX

The reader then responds with AF and 32 bytes of data

AF	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX

The card then responses with a final status code and 16 bytes of data.

00	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX

The process establishes that both parties have the same AES key, and creates a session key for further communications.

The way the response is calculated and checked is explained in a later chapter.

1A: Authentication (ISO)

This covers 3DES and 3K3DES. This type of encryption is not covered in this manual.

1A	Key ID

The response is AF and a number of bytes depending if 3DES or 3K3DES. There is then a reply command (AF) and a final response with status code in a similar way to the AES authentication.

0A: Authenticate (Legacy)

This covers DES. This type of encryption is not covered in this manual.

0A	Key ID

The response is `AF` and a number of bytes. There is then a reply command (`AF`) and a final response with status code in a similar way to the AES authentication.

54: Change KeySettings

This allows the settings for the current key to be changed.

Encrypted command		
54	Setting byte	CRC

The key settings depend if you are setting the master key for AID 0 (the card level), or for an application.

Bit	Card master key	Application key
0	The master key (i.e. key 0) can be changed	
1	List applications is possible without master key, otherwise card master key is needed.	Get FID list, Get File Settings and Get Key settings are possible, otherwise application master key is needed
2	Create applications is permitted without the master key. Delete needs card master key or app master key. If not set then both need card master key.	Create and Delete file are permitted without master key authentication, otherwise application master key is needed.
3	This setting can be changed. If unset, then that freezes this config.	
4-7	Not used	0-13: Key number required to change any key. 14: Auth with the key to be changed is needed to change. 15: All keys (except master) are frozen (master by bit 0)

5C00: Set Configuration (card config)

This sets top level card settings.

Encrypted command			
5C	00	Card config byte	CRC

The card config byte contains:-

Bit	Meaning
0	Disable card formatting
1	Enable random UID (note, this is a 4 byte random ID not 7)

Note, you can only set these bits! Once set they are set, tough!

5C01: Set Configuration (default key)

Not sure what this does, sorry. Perhaps it means the card is logged in with this by default?

Encrypted command				
5C	01	Key data, padded with 00 to 24 bytes	Key version	CRC

5C02: Set Configuration (Set ATS)

I guess this sets the ATS bytes.

Encrypted command				
5C	02	ATS len	ATS data	CRC

C4: Change Key

This allows a key to be changed.

Encrypted command					
C4	Key no	16 bytes key data (See note below)	Key version byte	CRC	CRC of new key (if changing different key)

If setting the master key for the card level, the key no has bit 7 set to indicate AES.

If changing a different key to the current key, the new key data is the new XOR'd with the old key. A CRC of the new key is included at the end of the message.

This message format is slightly different if not using AES.

64: Get Key Version

This gets the version of the key, for AES this is a version byte which can be set when setting the key. This allows you to then know which key to use if there have been different versions of keys in use.

64	Key ID

CA: Create Application

Create a new application. Depending on settings this may be possible without authenticating as the card master key.

CA	AID (3 bytes)	Key setting byte (see command 54)	App setting byte

The key setting byte is described in command 54.

The app setting byte contains:

Bit	Meaning
0-3	Number of keys, 1 to 14
6	Use 3K3DES (if not AES), else DES/3DES (if not AES)
7	Use AES (set bit 6 to 0)

DA: Delete Application

This allows an application to be deleted. The permission to allow deletion depends on settings, but it always requires authentication.

DA	AID (3 bytes)

6A: Get Application IDs

This one byte command gets a list of application IDs, each 3 bytes.

6A	

6E: Free Memory

This gets the free memory, the response is 3 bytes (low byte first).

6E	

6D: GetDFNames

Not sure, sorry.

6D	

45: Get KeySettings

Returns the settings for the specified key - the response is the settings, and number of keys.

45	Key ID

5A: Select Application

This selects an application.

`5A`	AID (3 bytes)

`FC`: FormatPICC

This formats the card - you need to be authenticated with the card master key for this.

`FC`	

`60`: Get Version

This gets the card version details.

`60`	

The response is several parts, and uses `AF` status code on each. Send an AF command to get the next part.

The first part if hardware version

Hardware version	
Vendor ID	`04` for NCP
Type	`01`
Sub Type	`01`
Major Version	1 byte
Minor Version	1 byte
Storage Size	`18` means 4k, `16` means 2k
Protocol Type	`05` means ISO 14443-2 and -3

The second part is software version

Software version	
Vendor ID	`04` for NCP
Type	`01`
Sub Type	`01`
Major Version	1 byte
Minor Version	1 byte
Storage Size	`18` means 4k, `16` means 2k
Protocol Type	`05` means ISO 14443-3 and -4

The last part provides other data

General version	
UID	7 byte UID, starting 04 for NXP
Batch	5 byte batch ID
Week	Week number (BCD coded, one byte)
Year	Year number (BCD coded, one byte)

51: GetCardUID

This requests the actual UID for when a card is set to use a random UID.

51	

The response is encrypted so only possible when authenticated, but it seems any key will do.

6F: Get FileIDs

This gets a list of file IDs in the selected application.

6F	

61: Get FileIDs (ISO)

This gets a list of ISO file IDs in the selected application.

61	

F5: Get FileSettings

This gets the file settings for a specific file in the application. These are the settings used when creating the file.

F5	File No

The response depends on the file type

Standard file settings				
00	00	Comms setting byte	Access rights (2 bytes)	File size (3 bytes)

Backup file settings				
00	01	Comms setting byte	Access rights (2 bytes)	File size (3 bytes)

Value file settings							
00	02	Comms setting byte	Access rights (2 bytes)	Min (4 bytes)	Max (4 bytes)	Limited credit (4 bytes)	Limited credit (1 byte)

Linear record file settings						
00	03	Comms setting byte	Access rights (2 bytes)	Record size (3 bytes)	Max records (3 bytes)	Records existing (3 bytes)

Cyclic record file settings						
00	04	Comms setting byte	Access rights (2 bytes)	Record size (3 bytes)	Max records (3 bytes)	Records existing (3 bytes)

5F: Change FileSettings

This allows file settings to be changed for a specific file in the application.

Encrypted command				
5F	File No	Comms setting byte	Access rights (2 bytes)	CRC

This is not encrypted if existing access right for changing settings are set to free access.

Communications settings

Bit	Meaning
0	CMAC communications
1	Encrypted communications (also set bit 0)

Access rights

Bit	Non value files	Value files
15-12	Read access	GetValue, and Debit
11-8	Write Access	GetValue, Debit, and Limited Credit
7-4	Read & Write access	GetValue, Debit, Limited Credit, and Credit
3-0	Change Access Rights	

Each of the access rights defines what key is needed for the specific access. Key 0-13 are key numbers, but 14 means *free access* and 15 means *no access*.

CD: Create StdDataFile

Create a standard data file.

CD	File No	Comms setting byte	Access rights (2 bytes)	File size (3 bytes)

CB: Create BackupDataFile

Create a backup data file.

CB	File No	Comms setting byte	Access rights (2 bytes)	File size (3 bytes)

CC: Create ValueFile

Create a value data file.

CC	File No	Comms setting byte	Access rights (2 bytes)	Lower limit (4 bytes)	Upper limit (4 bytes)	Initial value (4 bytes)	Limited credit available (1 byte)

C1: Create LinearRecordFile

Create a linear record file.

C1	File No	Comms setting byte	Access rights (2 bytes)	Record size (3 bytes)	Number of records (3 bytes)

C0: Create CyclicRecordFile

Create a cyclic record file. The number of records you can have is actually one less as the spare record is the one that is partly written before a commit.

C0	File No	Comms setting byte	Access right (2 bytes)	Record size (3 bytes)	Number of records (3 bytes)

DF: DeleteFile

Delete a file.

DF	File No

BD: Read Data

Read data from a file.

BD	File No	Offset (3 bytes)	Length (3 bytes)

The response will have MAC or be encrypted depending on file comms mode.

3D: Write Data

Write data to a file.

Potentially encrypted command				
3D	File No	Offset (3 bytes)	Length (3 bytes)	Data...

This needs to be sent using plain, CMAC or encrypted (with CRC) depending on file comms mode.

6C: Get Value

Get value from a value file.

6C	File No

0C: Credit

Credit value in a value file. You need to commit the changes.

0C	File No	Amount to credit (4 bytes)

DC: Debit

Debit the value in a file. You need to commit the changes.

DC	File No	Amount to debit (4 bytes)

1C: Limited Credit

Add limited credit. You need to commit the changes.

1C	File No	Amount to credit (4 bytes)

3B: Write Record

Write to a record. The offet/length are within the record. A new record is created with 00 bytes where not written. You need to commit the changes.

Potentially encrypted command				
3B	File No	Offset (3 bytes)	Length (3 bytes)	Data...

BB: Read Records

Read records - record 0 is the oldest/first.

BB	File No	Record number (3 bytes)	Number of records (3 bytes)

EB: Clear RecordFile

Clear all records from a file.

EB	File No

C7: Commit Transaction

Commit updates to value and record files.

C7	

A7: Abort Transaction

Abort changes to value and record files.

61	

Authentication

As explained in the commands, the authentication using AES involves a 3 way handshake. A request is sent for AES authentication for a specific key, a response (16 bytes) is received from the card. This is processed to make a 32 byte response sent back as a command to the card and then there is a 16 byte response from the card with a status code. If the status code is success (00) then we have successfully authenticated.

The process is as follows.

1. The reader asked for AES authentication for a specific key.
2. The card creates a 16 byte random number (B) and encrypts it with the selected AES key. The result is sent to the reader.
3. The reader receives the 16 bytes, and decrypts it using the AES key to get back the original 16 byte random number (B). This is decrypted with an IV of all 00 bytes.
4. The reader generates its own 16 byte random number (A).
5. The reader rotates B one byte to the left.
6. The reader concatenates A and the rotated B together to make a 32 byte value.
7. The reader encrypts the 32 byte value with the AES key and sends to the card. The IV for encrypting this is the 16 bytes received from the card (i.e. before decryption).
8. The card receives the 32 byte value and decrypts it with the AES key.
9. The card checks the second 16 bytes match the original random number B (rotated one byte left). If this fails the authentication has failed. If it matches, the card knows the reader has the right key.
10. The card rotates the first 16 bytes (A) left by one byte.
11. The card encrypts this rotated A using the AES key and sends it to the reader.
12. The reader receives the 16 bytes and decrypts it. The IV for this is the last 16 bytes the reader sent to the card.
13. The reader checks this matches the original A random number (rotated one byte left). If this fails then the authentication fails. If it matches, the reader knows the card has the AES key too.
14. Finally both reader and card generate a 16 byte session key using the random numbers they now know. This is done by concatenating the first 4 bytes of A, first 4 bytes of B, last 4 bytes of A and last 4 bytes of B.

The session key is then used for further communications.

Example, assuming an AES key of 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Cmd	
AA	00

Response from card																
AF	B9	69	FD	EE	56	FD	91	FC	9D	E6	F6	F2	13	B8	FD	1E

Decrypt to give B: C0 5D DD 71 4F D7 88 A6 B7 B7 54 F3 C4 D0 66 E8

Decrypt	
Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IV	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Code	B9 69 FD FE 56 FD 91 FC 9D E6 F6 F2 13 B8 FD 1E
Data	C0 5D DD 71 4F D7 88 A6 B7 B7 54 F3 C4 D0 66 E8

Rotate left to give B': 5D DD 71 4F D7 88 A6 B7 B7 54 F3 C4 D0 66 E8 C0

Create random A: F4 4B 26 F5 68 6F 3A 39 1C D3 8E BD 10 77 22 81

Concatenate A and B': F4 4B 26 F5 68 6F 3A 39 1C D3 8E BD 10 77 22 81 5D DD 71 4F D7 88 A6 B7 B7 54 F3 C4 D0 66 E8 C0

Encrypt: 36 AA D7 DF 6E 43 6B A0 8D 18 61 38 30 A7 0D 5A D4 3E 3D 3F 4A 8D 47 54 1E EE 62 3A 93 4E 47 74

Encrypt	
Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IV	B9 69 FD FE 56 FD 91 FC 9D E6 F6 F2 13 B8 FD 1E
Data	F4 4B 26 F5 68 6F 3A 39 1C D3 8E BD 10 77 22 81 5D DD 71 4F D7 88 A6 B7 B7 54 F3 C4 D0 66 E8 C0
Code	36 AA D7 DF 6E 43 6B A0 8D 18 61 38 30 A7 0D 5A D4 3E 3D 3F 4A 8D 47 54 1E EE 62 3A 93 4E 47 74

Further command to card																
AF	36	AA	D7	DF	6E	43	6B	A0	8D	18	61	38	30	A7	0D	5A
	D4	3E	3D	3F	4A	8D	47	54	1E	EE	62	3A	93	4E	47	74

The card responses with: 80 0D B6 80 BC 14 6B D1 21 D6 57 8F 2D 2E 20 59

Response from card																
00	80	0D	B6	80	BC	14	6B	D1	21	D6	57	8F	2D	2E	20	59

Decrypt to get A': 4B 26 F5 68 6F 3A 39 1C D3 8E BD 10 77 22 81 F4

Decrypt	
Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IV	D4 3E 3D 3F 4A 8D 47 54 1E EE 62 3A 93 4E 47 74
Code	80 0D B6 80 BC 14 6B D1 21 D6 57 8F 2D 2E 20 59
Data	4B 26 F5 68 6F 3A 39 1C D3 8E BD 10 77 22 81 F4

Match to A rotated one: 4B 26 F5 68 6F 3A 39 1C D3 8E BD 10 77 22 81 F4

Work out session key: F4 4B 26 F5 C0 5D DD 71 10 77 22 81 C4 D0 66 E8

Encryption

Commands

There are basically two ways a command can be sent, plain, CMAC, and encrypted. Which is used depends on the circumstances. The commands listed above are mostly *plain* with some listed as encrypted. However, the file access commands can be sent plain, CMAC, or encrypted depending on the file communications settings.

- Plain is sent as is, starting from command byte. However, once authenticated, the command as sent is passed through the CMAC process to update the IV. This means that when checking the CMAC on the response we can be sure no man in the middle changed either the command or the reply.
- Encrypted means that instead of passing through the CMAC process, the command is encrypted (updating the IV). In the case of file access commands a CRC is appended to the command first (others have the CRC explained as listed above). Then padding as normal (`80` then `00`s) to a multiple of 16 bytes and then CBC encrypting. Not all of the command is encrypted, the process starts at an offset (as highlighted above in the command list). For file access this is after the command byte and file number.

Responses

Responses can come back in one of three ways. Before authentication responses are plain, but once authenticated they have an 8 byte CMAC added to the end of the response. One response (get UID) is always an encrypted response. File access responses can be encrypted if the file communications say so. A CRC is appended to the response first (a CRC of the response bytes followed by the final, status byte), then padding. The encryption starts after the status byte.

Whilst the CMAC of the command is simply processed on the command from the command byte to the end, the processing of the response is more complex. The CMAC is calculated over the payload of the response (i.e after the status byte) and then the status byte appended to the end. If the response is multiple parts then the payload of these parts are concatenated (without the `AF` status byte) and the final status byte added to the end.

Authentication stops if there is an error response, or you select another application or change the key you are using.

CMAC

The CMAC process makes use of the session key and session IV, and updates the session IV. Before starting two sub keys need to be generated.

Subkey1 is generated by:-

1. Encrypting 16 bytes of `00` using the session key and an IV of all `00`.
2. Shift the result one bit left.
3. If 1 was shifted off the end (i.e. bit 7 set in first byte before shift) XOR last byte with `87`.

For example :-

Making subkey 1	
Key	2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
IV	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Code	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Encrypted	7D F7 6B 0C 1A B8 99 B3 3E 42 F0 47 B9 1B 54 6F
Shifted	FB EE D6 18 35 71 33 66 7C 85 E0 8F 72 36 A8 DE
XOR 87	Not needed as was 7D does not have top bit set.

Subkey2 is generated by:-

1. Shift subkey 1 left one bit
2. If 1 was shifted off the end (i.e. bit 7 set in first byte before shift) XOR last byte with 87.

Making subkey 2	
Subkey 1	FB EE D6 18 35 71 33 66 7C 85 E0 8F 72 36 A8 DE
Shifted	F7 DD AC 30 6A E2 66 CC F9 0B C1 1E E4 6D 51 BC
XOR 87	F7 DD AC 30 6A E2 66 CC F9 0B C1 1E E4 6D 51 3B

The CMAC process on a block of data involves the following steps.

1. Padding - if zero length, or not a multiple of 16 bytes, add 80 and additional 00 to make a multiple of 16 bytes.
2. If padding was added, XOR the last block (last 16 bytes) with subkey 2
3. If padding was not added, XOR last block (last 16 bytes) with subkey 1
4. Encrypt the data.
5. Update the IV with the result of encryption (the last block). This is the resulting CMAC, the first 8 bytes of which are appended to responses.

For example :- (all assuming these are the first message and so an IV of all 00)

Empty message (i.e. all padding)	
IV	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Padded	80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
XOR'd	77 DD AC 30 6A E2 66 CC F9 0B C1 1E E4 6D 51 3B
Encrypted	BB 1D 69 29 E9 59 37 28 7F A3 7D 12 9B 75 67 46

16 byte message (i.e. no padding)	
IV	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Message	6B C1 BE E2 2E 40 9F 96 E9 3D 7E 11 73 93 17 2A

16 byte message (i.e. no padding)	
XOR'd	90 2F 68 FA 1B 31 AC F0 95 B8 9E 9E 01 A5 BF F4
Encrypted	07 0A 16 B4 6B 4D 41 44 F7 9B DD 9D D0 4A 28 7C

16 byte message (i.e. no padding)	
IV	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Message	6B C1 BE E2 2E 40 9F 96 E9 3D 7E 11 73 93 17 2A AE 2D 8A 57 1E 03 AC 9C 9E B7 6F AC 45 AF 8E 51 30 C8 1C 46 A3 5C E4 11
Padded	6B C1 BE E2 2E 40 9F 96 E9 3D 7E 11 73 93 17 2A AE 2D 8A 57 1E 03 AC 9C 9E B7 6F AC 45 AF 8E 51 30 C8 1C 46 A3 5C E4 11 80 00 00 00 00 00 00 00
XOR'd	6B C1 BE E2 2E 40 9F 96 E9 3D 7E 11 73 93 17 2A AE 2D 8A 57 1E 03 AC 9C 9E B7 6F AC 45 AF 8E 51 C7 15 B0 76 C9 BE 82 DD 79 0B C1 1E E4 6D 51 3B
Encrypted	3A D7 7B B4 0D 7A 36 60 A8 9E CA F3 24 66 EF 97 B1 48 C1 7F 30 9E E6 92 28 7A E5 7C F1 2A DD 49 DF A6 67 47 DE 9A E6 30 30 CA 32 61 14 97 C8 27
CMAC	DF A6 67 47 DE 9A E6 30 30 CA 32 61 14 97 C8 27

CRC

The CRC used is x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1, with an initial value of all 1s. Note that this is encoded low byte first in messages.

Code example :-

```
unsigned int crc(unsigned int len, unsigned char *data)
{
 unsigned int poly=0xEDB88320;
 unsigned int crc=0xFFFFFFFF;
 int n,b;
 for(n=0;n<len;n++)
 {
  crc^=data[n];
  for(b=0;b<8;b++)
   if(crc&1)
    crc=(crc>>1)^poly;
   else
    crc>>=1;
 }
 return crc;
}
```

For example:-

CRC	
Data	00 10 20 30 40 50 60 70 80 90 A0 B0 B0 A0 90 80
CRC	1979E3BF
Coded as	BF E3 79 19

Changing to AES master key

Initially the card will probably be set to use a simple DES master key, and you will probably want to change that to AES.

To do this you have to authenticate using DES, and then set a new key.

The DES authentication is basically the same but using 8 byte A and B values and keys. So start with all `00` key and IV. And send an `1A` for key `00`.

Cmd	
1A	00

The response will be 8 bytes of data, confirming its is DES in use.

Response from card								
AF	C3	27	E0	B3	AE	78	4F	04

Decrypt	
Key	00 00 00 00 00 00 00 00
IV	00 00 00 00 00 00 00 00
Code	C3 27 E0 B3 AE 78 4F 04
Data	8A 9D 09 A4 3D 2D D3 92

Create your own A random value, e.g. `9F 02 17 83 26 DD E5 A2`

Generate the response.

Further command to card																
AF	DC	C7	FB	9A	26	1C	7D	FC	1	20	14	A9	2B	BB	CD	CB

The card responds

Response from card								
00	75	FD	A7	DC	10	07	12	A4

Decoding this confirms the A value is correct. You can then make a session key from the first 4 bytes of the A and B values. Set the IV to all `00`s.

Session key							
9F	02	17	83	8A	9D	09	A4

Then you need to change the key. This is a `C4` command for get `80`. Note `80` is used to indicate key 0 but using AES.

Construct a new message :-

Before encryption																
C4	80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	00	01	1D	D9	EA	C2	00	00	00							

This message has command `C4`, key `80`, AES key all `00`s, key version `01`, and then CRC of everything up to that point, and padded with `00` to multiple of 8 bytes for encryption.

Encrypt with the new session key:-

Before encryption																
C4	80	61	59	2D	C4	0A	D3	58	95	16	52	D8	38	31	A2	73
	CC	E3	EA	31	34	17	83	C4	1E							

You get a 00 and 8 byte CMAC status confirming it worked. You should check the CMAC really, but as this is a one off process to switch to AES you probably don't really need to.