MAX32550 SCP Specifications

Ref: SPEC25B02
Rev A

MAXIM INTEGRATED PROPRIETARY – CONFIDENTIALITY

Revision History

Rev A	2014-Sep-5	1st release

Disclaimer

To our valued customers

We constantly strive to improve the quality of all our products and documentation. We have spent an exceptional amount of time to ensure that this document is correct. However, we realize that we may have missed a few things. If you find any information that is missing or appears in error, please send us an email via www.maximintegrated.com/support. We appreciate your assistance in making this a better document.

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Maxim Integrated Office.

Maxim Integrated technologies may only be used in life-support devices or systems with the express written approval of Maxim Integrated, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents

List of Figures

List of Tables

References

[1] FIPS 113: Compute Data Authentication, NIST, 30-may-1985

[2] FIPS 180-2: Secure Hash Standard. NIST, August-2002.

[3] FIPS 186-3: ECDSA. NIST.

[4] UG25H05: MAX32550 User Guide.

[5] DS25H01: MAX32550 Datasheet.

Glossary

ECDSA	Elliptic Curve Digital Signature Algorithm
GPIO	General Purpose Input Output
OTP	One Time Programmable
RFU	Reserved for future use
ROM	Read Only Memory
SBL	Secure Boot Loader
SOC	System On Chip
SRAM	Static Random Access Memory
TRNG	True Random Number Generator
USN	104-bit Unique Serial Number

1 INTRODUCTION

This document specifies the secure communication protocol, SCP, used on the ARM-based chips family. It is implemented in the SecureROM code.
Up to now, this SecureROM-based chips list is MAX32550.
The life cycle of the chip is made of 5 phases:

- phase 0:
  - *conception*: software development and hardware design are performed.
  - *wafer*: Wafer manufacturing.
- phase 1:
  - *EWS:* electric wafer sort (Flash Test Mode and Test Mode enabled).
  - *testing:* Final testing, Execution of test applets loaded through GPIOs.
- phase 3: *OTP locked*: Maxim OTP is locked, device is tested. TM is locked.
- phase 4: *in-the-field*: the chip starts and runs customer applications.
- phase 5: *end-of-life*: the chip does not start.

SCP is available in phase 3 (for transition to phase 4 only) and in phase 4, for customer needs.

1.1 Notations

Hexadecimal numbers are represented with a `0x` in front of them and in `Courier New`.
Example: `0xF8DF`
Decimal numbers are indicated by a 10 in index.
Example: 123_{10}

1.2 Endianness

The chosen convention is “big endian”. Hexadecimal numbers are presented most significant byte first and most significant bit first. It means the 32-bit number `0x123456` is 1193046_{10}.

1.3 Cryptographic support

1.3.1 ECDSA

This algorithm is used in SCP public session security, with 256-bit keys, for digital signatures. The public part is used for digital signature verification while the private part is used to sign packets.
The digital signature algorithm is ECDSA (see **Error! Reference source not found.**).
The curve is secp256r1.

1.3.2 SHA-256

This algorithm is used in SCP public session security, with for digital signatures.

1.3.3 AES-CRC

We propose here to use the AES as a CRC-like function, very efficient as being in hardware. The proposed algorithm is a AES-CBC-MAC with the null key.

If we want to compute the digest of the data, made of n blocks $B_{i,i=1..n}$ of 128 bits, the proposed algorithm is the following, the computation is:

Consider h_0 an initial 128-bit value, equal to 0.
Consider *CVK* a 128-bit value, equal to 0.
$h_i=AES_{CVK}(B_i \oplus h_{i-1})$, i=1..n (i.e. take the null key, and h_{i-1} as the IV).

h_n is the 128-bit hash result.

2 SCP Keys management

SCP ECDSA keys are:

- Maxim root key: MRK: ECDSA 256-bit key;
  - owned by Maxim
  - private part managed securely (generation, storage and use under dual control with HSM) at Maxim Secure Micro-controllers BU; used for customers public keys certification,
  - public part stored within the ROM code,
  - used for CRK authentication and download in phase #3,
  - used by ROM code for digital signature verification

- Customer Root key: CRK: ECDSA 256-bit key:
  - owned by the customer
  - private part managed securely (generation, storage and use under dual control with HSM) at Customer premises; used for (phase #4) secure downloads,
  - public part stored within the OTP,
  - used by the secure loader in the ROM for secure data/firmware downloads on the terminal using digital signature verification,
  - Stored with its MRK signature.

Key name	purpose	algorithm-length	owner	generation	location	Life-cycle
MRK	CRK certification	ECDSA-256	Maxim	HSM	Private in HSM, public in OTP	#3
CRK	SCP protocol packets authentication	ECDSA-256	Customer	HSM	Private in HSM, public in OTP	#4

Keys table

3 SECURE COMMUNICATION

In phases #3 and #4 a customer-dedicated protocol is available from the ROM code to allow data and firmware secure loading.

It is important to notice that in phase #3, only one specific command, the customer public key loading, WRITE-CRK, is allowed. This five-layer protocol is available on the serial port and optionally on the USB link.

To allow remote access to SecureROM-based chips, the chip has to be connected to an external machine that communicates with it.

The communication is done through a protocol, called the Secure Communication Protocol a.k.a. SCP, which sets up sessions where data are securely exchanged, accordingly to some configurations.

3.1.1 Terminology

In the following sections, the external machine connected to the chip is called host.

3.1.2 General Architecture

The SCP consists of a stack of layers that can be mapped onto the layers of the Open Systems Interconnection (OSI) model:

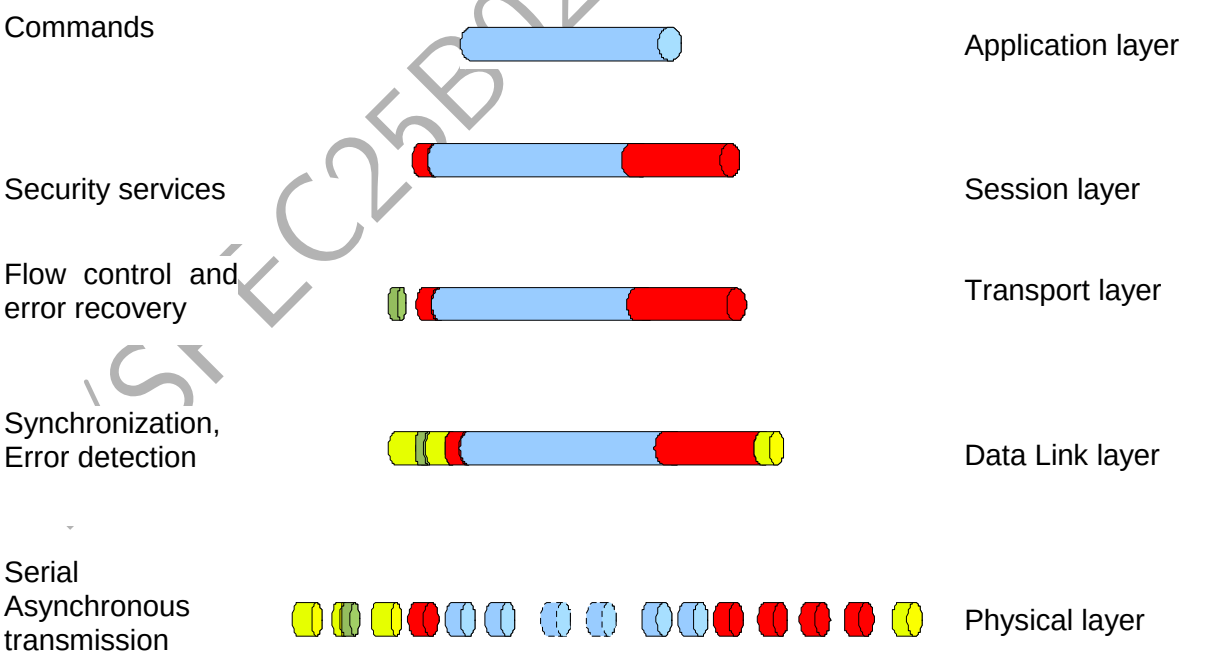


Figure 1 Communication Protocol Layers

The protocol encapsulation is described in the Figure 3. The optional packets are in dotted line. The number indicates the size in bytes.

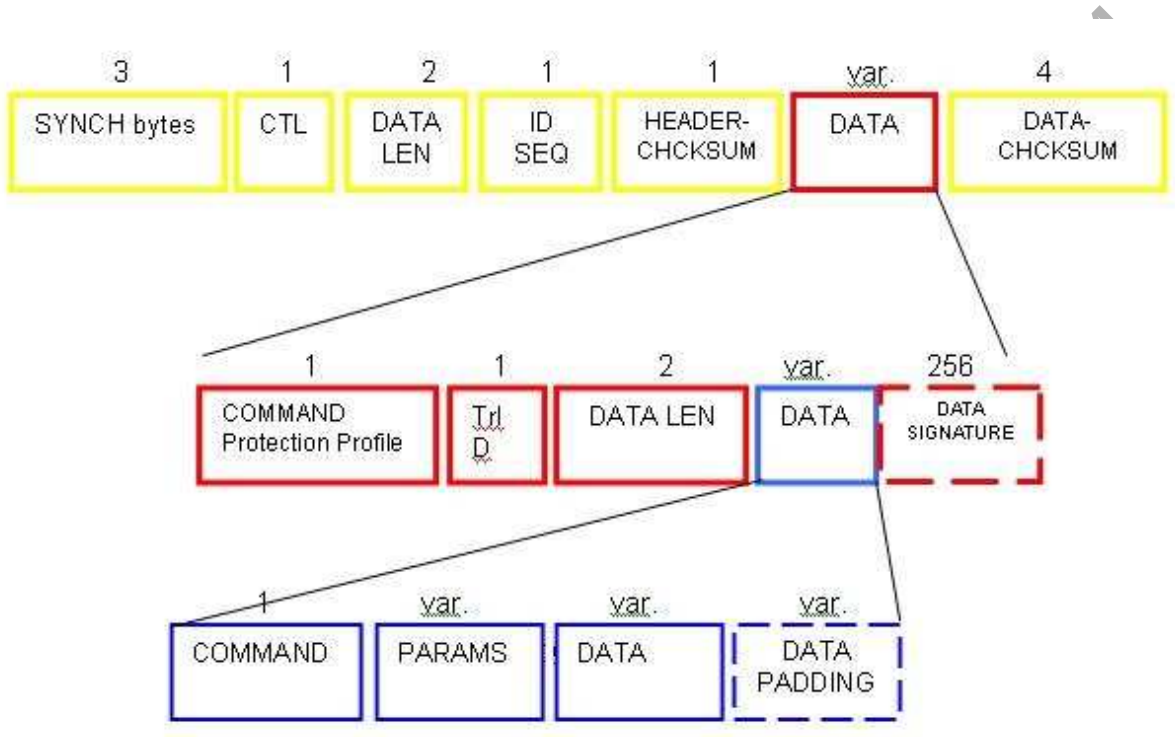


Figure 2 SCP RSA protocol packets

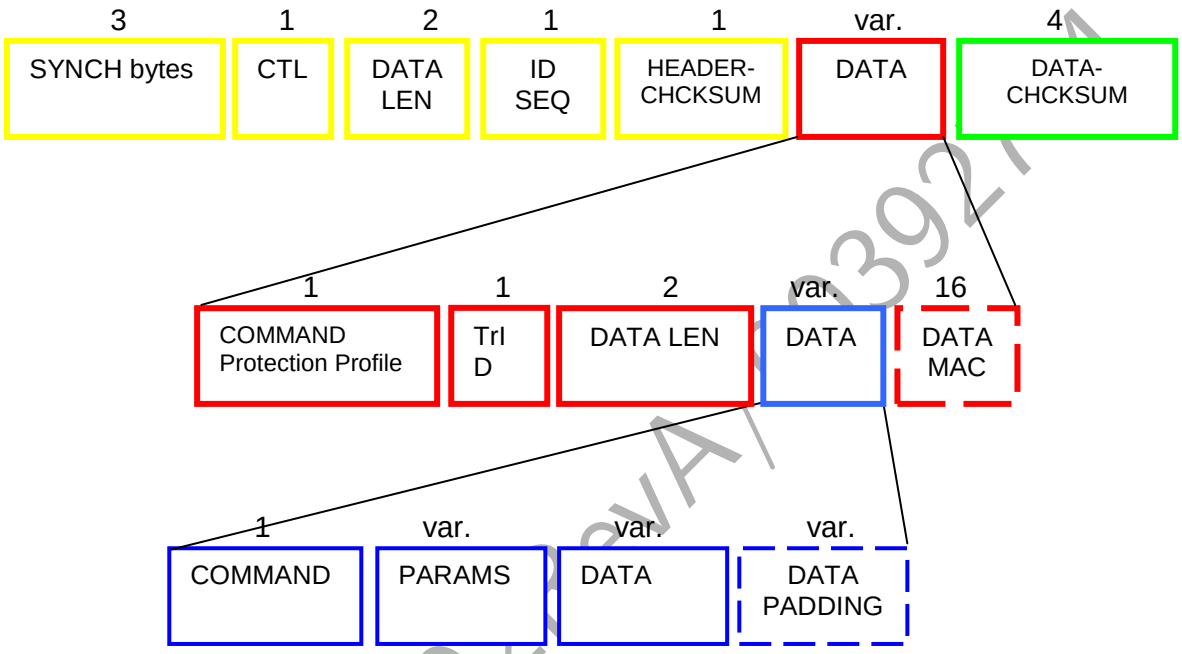


Figure 3 AES protocol packets

3.1.3 Limitations

The secureROM input buffers have a maximum size of 4 KB for the data packets

3.2 Physical Layer

This layer defines the physical (i.e. hardware) implementation for the SCP.

3.2.1 Functional specification

At this layer, data are transmitted over a full-duplex communication link. Data may be transmitted in both directions over the link. A stream of data is communicated by being broken up into bytes.

3.2.1.1 Terminology

At this layer, a Protocol Data Unit (PDU) is called a Byte.

3.2.1.2 Interface

The physical interface used by the chip to communicate with the external world is the UART 0.

3.2.1.3 Configuration

The UART transmitting information is configured as follows:

- 8-bit data character,
- No parity,
- 1 stop bit,
- No flow control

- Speed: 115200 b/s by default (and also configurable)

3.2.2 Framing Error Handling

When a framing error occurs, the data byte is discarded.

3.3 Data Link Layer

The Data Link Layer rule is to package bytes from the Physical layer into frames (logical, structured packets for data) and to check the data integrity.

3.3.1 Functional Specification

At this layer, data bytes are serially accumulated to form a frame. The frame is the unit of data communicated over the link.

Each frame is composed of a header (in yellow in Figure 3) followed by a data portion. The data portion can be empty.

3.3.1.1 Terminology

At this layer, a Protocol Data Unit (PDU) is called a frame.

3.3.1.2 Checksum Algorithm

The chosen checksum algorithm is not a usual CRC but one cryptographic hash function: the AES-CRC (see section 1.3.3.).

3.3.1.3 Header Format

The header is 8-byte long:

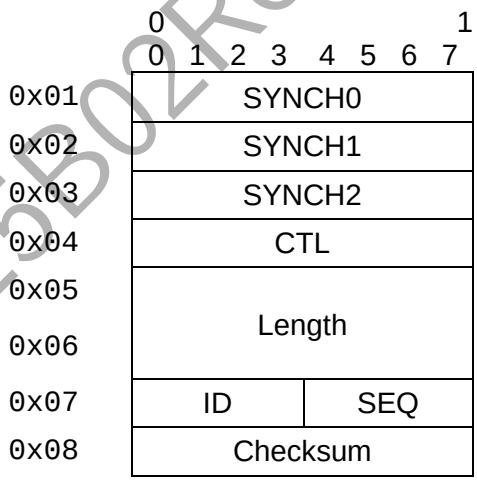


Figure 4 Data link header format

Synchronization pattern ($SYNCH_n$)

The serial asynchronous transmission provides a self-clocking communications medium. The wires used for data transfer are often placed in 'noisy' environments, the latter being able to “add” erroneous data. For this reason, the start of a packet is denoted by a three-octet synchronization pattern. This allows the receiver to discard noise which appears on the connection prior to the reception of a packet.

The synchronization pattern is defined to be the three following hexadecimal bytes :

- $byte_0$: SYNCH0: 0xBE
- $byte_1$: SYNCH1: 0xEF
- $byte_2$: SYNCH2: 0xED

Control byte (CTL)

The “Control field” is one-octet long and codes the higher level protocol used to process the enclosed fragment.

Length bytes

The “Length” field is two-octet long and indicates the length of the client data.

- $byte_4$: Length of the data (MSB)
- $byte_5$: Length of the data (LSB)

Channel Identifier (ID)

The “Channel Identifier” field is one nibble and should be used by the upper protocol level to help in matching responses with requests.

Sequence Numbers (SEQ)

The “Sequence numbers” field is one nibble and should be used by the upper protocol level to help in matching data transfer with acknowledgment.

Header checksum

The “header checksum” field is one-byte long and contains an error-detection code. It is computed as the first byte of the AES-CRC of all the bytes covering the synchronization pattern, the control byte, data length, the channel ID and the sequence number bytes. As the header is 7-byte long while the AES-CRC requires inputs multiple of 16 bytes, nine zeros are added on the right for padding.

The padding zeros are not transmitted.

3.3.1.4 Data Portion Format

The “data portion” of a frame immediately follows the header if the “Length” field is strictly greater than zero. It consists of “Length” client data bytes (in red in Figure 3) immediately followed by four “data checksum” bytes. If present, the “data portion” contains “Length”+4 bytes.

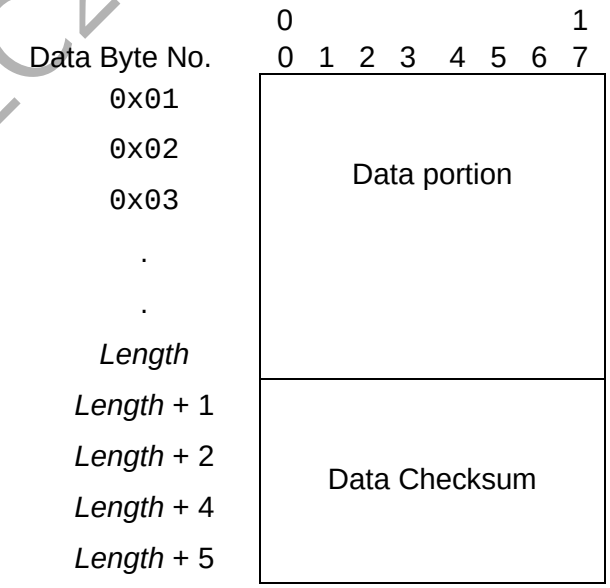


Figure 5 Data link data format

Data checksum

The last four bytes of the data portion of a frame are a “data checksum” (in green in Figure 3). This is used by this protocol to detect incorrectly transmitted data. This has shown

itself to be a reliable method for detecting most categories of bit drop out and bit insertion. The checksum is represented by the 4 first bytes of the AES-CRC on the client data. If the "Length" bytes are not a multiple of 16 bytes, zeros are added on the right for padding until reaching a multiple of 16 bytes.

The padding zeros are not transmitted.

3.3.2 Functional Specification

3.3.2.1 Frame Transmission

The transmission of a frame consists of these steps:

1. Fill in the frame header and compute its checksum.
2. Compute the checksum of the client data.
3. Transmit the frame header, client data, and data checksum.

3.3.2.2 Frame Reception

A frame reception consists of these steps:

1. Scan the serial input until the frame header synchronization pattern is matched.
2. Read in the rest of the frame header and validate its checksum.
3. Read in the client data.
4. Read in and validate the data checksum.

3.3.2.3 Timer

A timeout is associated with the successful packet reception. Considering the situations in which a connection is so noisy that no data are received correctly, or a connection is physically cut. Without an overriding timeout, these situations would result in unbounded data listening. For practical implementation reasons, the timeout is fixed to 10 s.

3.3.2.4 Data Error Handling

Framing errors

The start of a packet is given by the "synchronization pattern". The expected end of a packet can now only be determined by examining the "Length" byte of the header. It is important to know whether or not the "Length" data can be trusted. This is performed by using a one-byte "header checksum" to cover the first bytes following the synchronization pattern. If the header passes the checksum test, the number of bytes expected beyond the header is "Length"+4. If the header fails the checksum test, the length is incorrect so either too small or too large. The following sequence describes how to recover from this error:

1. parse immediately after the synchronization pattern,
2. rescan looking for the next synchronization pattern,
3. throw away all bytes until a synchronization is found,
4. then attempt to reinterpret it as a frame.

The sender's retransmission timeout guarantees that a new copy of the frame will be transmitted. This ensures that in discarding the initial synchronization pattern, the synchronization pattern from the beginning of the retransmitted frame will eventually be found.

Missing synchronization pattern

Any valid frame must begin with the "synchronization pattern". Any receiver must discard all input bytes until the "synchronization pattern" is seen. The data which immediately follow a "synchronization pattern" is interpreted as a frame. The "header checksum" test is applied, then "Length"+4 bytes are read and the "data checksum" test is applied.

Bad frames

A bad frame is received when it fails either the header or data checksum tests. When this happens, the error is flagged to the upper layer.

Too large frames

Assume a frame arrives which passes its header checksum test but whose Length is larger than the maximum data length of the receiver. In such a case, the sender has violated the protocol or a frame has a data error in the Length octet and has passed the header checksum test. When this happens, the error is flagged to the upper layer.

Too small frames

Assume that a frame has passed its header checksum test but some of the data bytes have been dropped by the link. In such a case the receiver's routine which reads data and builds frames is expecting bytes which do not arrive. When this happens, the error is flagged to the upper layer.

3.4 Transport Layer

For the SCP, the Transport Layer provides transfer correctness, data recovery, and flow control. To allow reliable communication, it establishes, manages and terminates connections with clear phases of link establishment, information transfer, and link termination.

3.4.1 Functional Specification

This layer implements a reliable message service through a number of mechanisms:

- a sequence numbers insertion into segments,
- a positive acknowledgment of segment receipt,
- a timeout and a retry strategy.

3.4.1.1 Terminology

At this layer, a Protocol Data Unit (PDU) is called a Segment.

3.4.1.2 Commands and formats

Each segment is made of a “channel identifier”, a “sequence number”, a “control code” and optional data which are all enclosed in a data link frame.

Channel Identifier (ID)

The “channel identifier” field is one nibble and helps in matching responses with requests. The “channel identifier” is provided by the host to identify an active connection, and must be the same in both ingoing and outgoing segments.

This “channel identifier” is fixed for the whole connection.

Sequence Numbers (SEQ)

The “sequence numbers” field is one nibble and works with positive acknowledgement to inform the sender that his last segment was received: the received “sequence number” shall be sent back by the receiver to the sender. The “sequence number” shall be incremented by one modulo 16 by the sender for each new data segment (a segment represents each data exchange on one way; i.e. a ACK is not a new segment).

The “sequence number” initial value is 0, used for the first data exchange after session opening (i.e. after the HELLO-REPLY for ECDSA sessions).

```
seq=(seq + 1) mod 16
```

Control Code (CTL)

The “Control Code” field is one byte and identifies the type of segment. Codes are assigned as follows:

There are three categories of segments:

- Protocol segments
- Data transfer segments
- Maintenance and test segments

Category	*Name*	*Value*	*Description*
Protocol	CON_REQ	0x01	Connection request
	CON_REP	0x02	Connection reply
	DISC_REQ	0x03	Disconnection request
	DISC_REP	0x04	Disconnection reply
Data Transfer	DATA_TRANSFER	0x05	Data Exchange
	ACK	0x06	Acknowledge
Maintenance	ECHO_REQ	0x0B	Echo request
	ECHO_REP	0x0C	Echo reply

Table 1 Data Link segment types

3.4.1.3 Retransmission Timeout

Segments may be lost in transmission for two reasons: either they are lost or damaged due to the effects of the noisy transmission media or they are discarded by the receiver. The positive acknowledgment policy requires the receiver to acknowledge a segment only when the segment has been correctly received and accepted.

To detect missing segments, the sender must use a retransmission timer for each segment transmitted. The timer is set to a value approximating the transmission time of the segment in the network. When an acknowledgment is received for a segment, the timer is canceled for that segment. If the timer expires before an acknowledgment is received for a segment, that segment is retransmitted and the timer is restarted.

The re-transmission timeout is used to trigger the re-transmission of unacknowledged frames. **It is set to 10s**.

3.4.2 Protocol Operation

3.4.2.1 Establishing a connection

A "three-way handshake" is the procedure used to establish a connection. It is always initiated by the host.

The three-way handshake is illustrated in the Figure 6.

HOST	**On the link**	**CHIP**
	Connection is closed	
Connection request		
	CON_REQ ->	
		Connection accepted
	CON_REP <-	
Connection opened		
	ACK ->	
		Connection opened

Figure 6 Session Connection Sequence

3.4.2.2 Example

```
<host>.0000001.CON_REQ
beefed01000090f3
<chip >.0000002.CON_REP
beefed0200009001
<host>.0000003.ACK
beefed06000090c7
```

in the first line, BEEFED is the SYNCHRO, 01 is the CONTROL (for CON-REQ), 0000 is the length of data payload (no payload), 9 is the CHANNEL-ID, 0 is the SEQ-ID and F3 is the CRC.
In the second line, BEEFED is the SYNCHRO, 02 is the CONTROL (for CON-REP), 0000 is the length of data payload (no payload), 9 is the CHANNEL-ID, 0 is the SEQ-ID and 01 is the CRC. The SEQ-ID has not been incremented, as it is incremented only for new data segments.

3.4.3 Closing a Connection

The host may initiate the disconnection sequence (see Figure 7). This is accomplished by sending a segment with the DISC_REQ control code. No data may appear in a DISC_REQ segment. The other end of the connection responds by shutting down its end of the connection and sending an DISC_REP in response.

HOST	On the link	CHIP
	Connection is opened	
disconnection request		
	DISC_REQ ->	
		disconnection accepted
	DISC_REP <-	
Connection closed		
	ACK ->	
		Connection closed

Figure 7 Session Disconnection Sequence

In particular, if the ACK is not received within the *re-transmission timeout* then a DISC_REQ segment should be sent and the disconnection assumed to have succeeded.

3.4.3.1 Example

```
<host>.0002084.DISC_REQ
beefed030000a0a5
<usip>.0002085.DISC_REP
beefed040000a06d
```

3.4.4 Data Transfer

Once the connection is established, data are communicated by the exchange of segments with control code DATA_TRANSFER (see Figure 8). Because segments may be lost due to errors (e.g. detected by the checksum test failure), retransmission is used, after a timeout, to ensure delivery of every segment. Duplicate segments may arrive due to network or retransmission. The protocol performs some tests on the sequence numbers in the segments to verify their acceptability.

An acknowledgment should always be sent in response to a received data segment, even if the sequence number does not match the expected value. However, the sequence number of the acknowledgment should be set to the value from the last valid data segment received, not necessarily that of the data segment being responded to. If the acknowledgment is not received within the re-transmission timeout, or if an acknowledgment is received with a different sequence number from that sent in the data segment, then the data segment may be re-transmitted up to 8 times before the link is disconnected.
The sequence number must not be incremented when re-transmitting a data segment.

HOST	On the link	CHIP
	Connection is opened	
Data sent		
	CH-ID, SEQ_n, DATA	
		Data accepted
	CH-ID, SEQ_n, ACK	
Data sent		
	CH-ID, SEQ_{n+1}, DATA	
		Data accepted
	CH-ID, SEQ_{n+1}, ACK	

Figure 8 Session Data Transfer Sequence

3.4.4.1 Testing connection

For testing purpose, the host can initiate a loop-back sequence:

- The host sends an ECHO_REQ segments with as many data it wants.
- The chip shall respond with an ECHO_REP followed by the data previously sent by the host.

The sequence is illustrated in the Figure 9.

HOST		CHIP
	Connection is opened	
ECHO requested		
	SEQ_n, ECHO-REQ	
		ECHO accepted
	SEQ_n, ECHO-REP	

Figure 9 Testing Connection Sequence

3.4.5 Data Error Handling

3.4.5.1 Framing Errors

When this happens, the sender will retransmit the segment after the retransmission timeout interval.

3.5 Session Layer

For the SCP, the Session Layer manages security issues by providing data encryption and data signature.

3.5.1 Functional specification

3.5.1.1 Terminology

At this layer, a Protocol Data Unit (PDU) is called Data.

3.5.1.2 Commands and formats

Data have a variable length, made of two parts:
1. Header
2. Payload

Header Format
The header is 4-byte long:

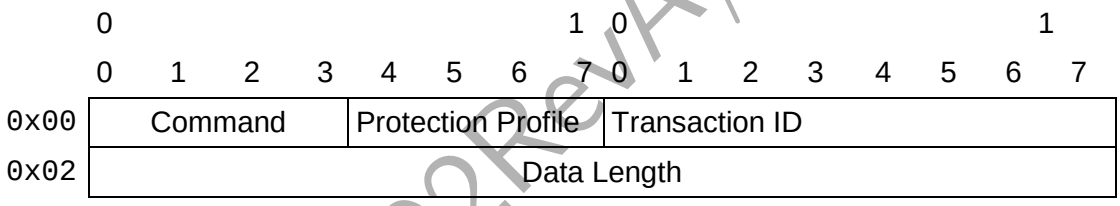


Figure 10 Session layer header format

Protection Profile
The “protection profile” field is one nibble. It determines the security options to be applied on data transfer.

The Figure 11 describes the protection profiles features.

Value	Description
0xA	The payload is transmitted in a plain form with ECDSA signature

Figure 11 Protection profiles list

Command
The “command” field is one nibble and identifies the type of packets. Command codes are assigned as described in Figure 12.

Name	Value	Description
HELLO	0x1	Hello-Request
HELLO_REP	0x2	Hello-Reply
DATA	0x5	Data Transfer

Figure 12 Session layer command codes

Transaction ID

The "Transaction ID" field is one byte. The initial value is fixed to zero. The Transaction ID is incremented after each data transfer ended successfully (modulo 256).

```
tr-id=(tr-id +1) mod 256
```

Length

The "Length" field is two bytes and codes the data payload length:

- byte2: length of the payload (LSB)
- byte3: length of the payload (MSB)

Note: the larger this "length" value is, the lesser the protocol-related fields have an impact on the performances. For example, in ECDSA signed protection profile, there is around 78 bytes related to the protocol. If the data "length" is around 4Kbytes, the ratio becomes 51, indicating a good efficiency.

3.5.1.3 Data Payload

The data payload has a possible length coded on two bytes. It is followed by a ECDSA digital signature (on 64 bytes).

Digital signature (ECDSA)

The ECDSA digital signature is a 64-byte cryptographic value, using the ECDSA private part and the SHA256 algorithm.

3.5.2 SCP session opening

Sessions shall be opened in order to allow the chip to receive and execute commands. After a connection opening, the session is opened by a HELLO command. This command mainly sets up the security protection of the commands run that will be run within the session.

3.5.2.1 Session Keys

An opened session is linked to one single set of keys: the session keys. This set is chosen accordingly to the protection profile. The set is one single key (CRK in phase #4 or MRK in phase #3)

3.5.3 SCP ECDSA protection profile

This protection profile provides authentication of packets sent by the host to the chip. Neither encryption nor authentication from the chip is possible.

The handshake protocol messages are presented in the order they must be sent; sending handshake messages in an unexpected order results in a fatal error.

The Figure 13 summarizes the message flow for a full handshake:

HOST	On the link	CHIP
	Connection is opened	
HELLO request		
	CH-ID, SEQ_0, HELLO-REQ, $TR\text{-}ID_0$ ->	
		HELLO reply
	<- CH-ID, SEQ_1, HELLO-REP, $TR\text{-}ID_0$	
Acknowledge		
	CH-ID, SEQ_1, ACK ->	

	Session is opened	
Application Data		
	CH-ID, SEQ_2, DATA->	

Figure 13 ECDSA session opening

3.5.3.1 HELLO Command

The HELLO command is sent by the host to signal the start of a session. The chip responds with HELLO_REPLY.

	0							1	0							1
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0x00	HELLO				0				0							
0x02	0x00								0x0A							
0x04	'H'								'E'							
0x06	'L'								'L'							
0x08	'O'								0x20							
0x0A	'B'								'L'							
0x0B	0x02								0x02							

Figure 14 ECDSA HELLO command format

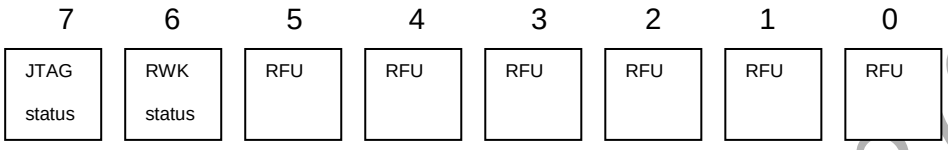
3.5.3.2 HELLO_REPLY Command

A HELLO_REPLY is sent by the chip in response to the HELLO command at the start of a session. This reply is used to inform the host about the chip configuration (including USN, JTAG status).

	0							1	0							1
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0x00	HELLO_REPLY				0				0							
0x02	0x00								0x32							
0x04	'H'								'E'							
0x06	'L'								'L							
0x08	'O'								0x20							
0x0A	'H'								'O'							
0x0C	'S'								'T'							
0x0E	secureROM version								secureROM version							
0x10	secureROM version								secureROM version							
0x12	life-cycle information								0x00							
0x14	0x00								Configuration							
0x16	USN padded with 0x00..00 bytes															
0x18																
0x1A																
0x1C																
0x1E																
0x20																
0x22																
0x24																
0x26	0x00															
0x28																
0x2A																
0x2C																
0x2E																
0x30																

Figure 15 ECDSA HELLO_REPLY command format

Configuration:
The configuration byte is made of:



- JTAG bit: '**1**' if JTAG is **deactivated**, '**0**' if **activated**.
- RWK bit: '**1**' if RWK mode is **available**, '**0**' if **not available**.
- RFU bits are set to **0**.

3.5.4 Data transfer

The Figure 16 summarizes the message flow for a data transfer :

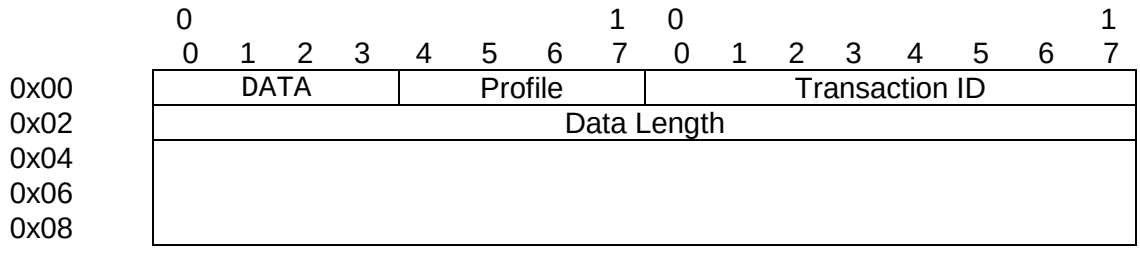
HOST	On the link	CHIP
	Session is opened	
Data transfer		
	CH-ID, SEQ_n, command, $TR\text{-}ID_m$ ->	
		Acknowledge
	<- CH-ID, SEQ_n, ACK	
		Command response
	<- CH-ID, SEQ_{n+1}, command response, $TR\text{-}ID_m$	
Acknowledge		
	CH-ID, SEQ_{n+1}, ACK ->	
Next data transfer		
	CH-ID, SEQ_{n+2}, command, $TR\text{-}ID_{m+1}$->	
		Acknowledge
	<- CH-ID, SEQ_{n+2}, ACK	
		Command response
	<- CH-ID, SEQ_{n+3}, command response, $TR\text{-}ID_m$	
Acknowledge		
	CH-ID, SEQ_{n+3}, ACK ->	

Figure 16 Transaction ID management

The Transaction ID is incremented after each successful data transfer from the host to the chip. The transfer is able to occur with the data payload signed by a ECDSA digital signature with CRK or MRK

3.5.4.1 DATA packet

The DATA packet is used to transfer at the upper layer the Application Data. The Transaction ID is incremented after each successful DATA transmission. This command is described in Figure 17.



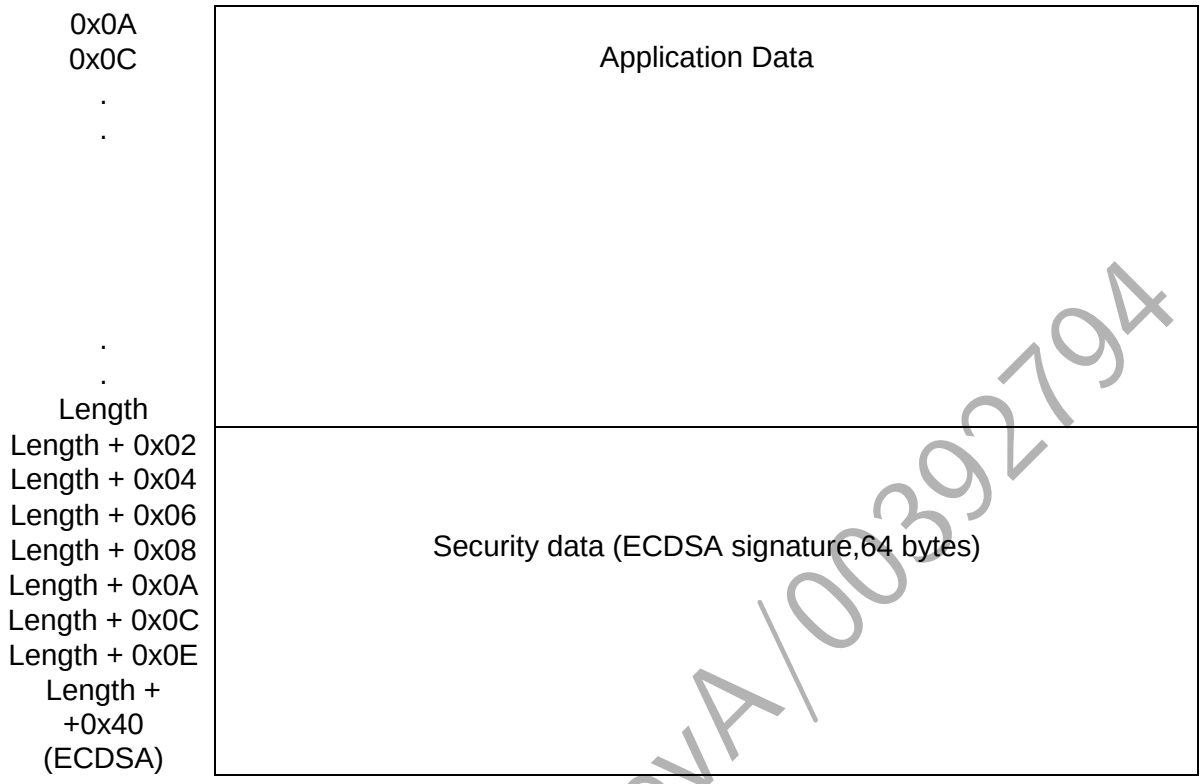


Figure 17 DATA command format

3.5.5 Security aspects

3.5.5.1 Signature message

The signature is computed on the message content (Application Data).

3.5.5.2 Delete or reorder messages attacks

The transaction ID ensures that attempts to delete or reorder messages will be detected.

3.6 Application layer

The application layer deals with the commands used to really perform personalization of the chip. The next chapter describes these commands in details.

4 COMMANDS REFERENCE

4.1 Introduction

Every command is available whatever the chosen protection profile. It is up to the host manager to choose the correct one depending on its own security constraints. Commands between the host and the chip are provided for the chip life cycle management, code and data downloading. In phase #3, only the `write-CRK` command can be run.

4.2 Commands formats

The commands correspond to the application layer of the OSI model and are enclosed in session protocol messages. A session protocol message must contain only one command that entirely fits within the message. Commands always return an error code which is also enclosed in a session protocol message.

Commands consist in a one-byte opcode optionally followed by additional data bytes (corresponding to the command arguments).

4.2.1 Command opcode

Several types of command are available:

- Administrative commands.
- Flash memory commands.
- Remote service commands.

The Table 2 lists the available commands.

Category	Commands	Value
Administrative/Keys	WRITE CRK	0x470A
	REWRITE CRK	0x461A
	WRITE OTP	0x4714
	WRITE TIMEOUT	0x4426
	KILL CHIP	0x4538
Memories	WRITE DATA	0x2402
	COMPARE DATA	0x2403
	ERASE DATA	0x4401
	EXECUTE CODE	0x2101

Table 2 Command List

Note : the commands list is inherited from MAX32590 SCP.

4.2.2 General Error Conditions

Each command returns an error code to determine if the operation has been done successfully or not.

Code	Value	Meaning
ERR_NO	0x00000000	No error
ERR_NO_CONFIG	0x00000001	No configuration found
ERR_NO_INIT	0x00000002	
ERR_BAD_VALUES	0x00000003	Unexpected values (e.g. length)
ERR_ALREADY	0x00000004	Command can not be done again
ERR_RUNNING	0x00000006	
ERR_NO_APPLI	0x00000007	
ERR_BAD_STATE	0x00000008	
ERR_CRITICAL	0x00000009	
ERR_PROHIBIT	0x0000000A	Command not allowed with this key (CRK instead of MRK)

Table 3 Commands error codes

4.3 Administrative/Keys Commands

The administrative commands are related to the life cycle management (used keys, life cycle value and download timeout).

4.3.1 WRITE-CRK Command

4.3.1.1 Definition and scope

This command writes the customer CRK on the chip user OTP.

Note1: this command is one of the two allowed ones with the MRK.
Note2: this command is not allowed with the CRK.
Note3: this command is only available in phase #3.
Note4: this command is allowed only once

4.3.1.2 Command Format

The WRITE-CRK (`0x4701`) command is coded according the Figure 18.

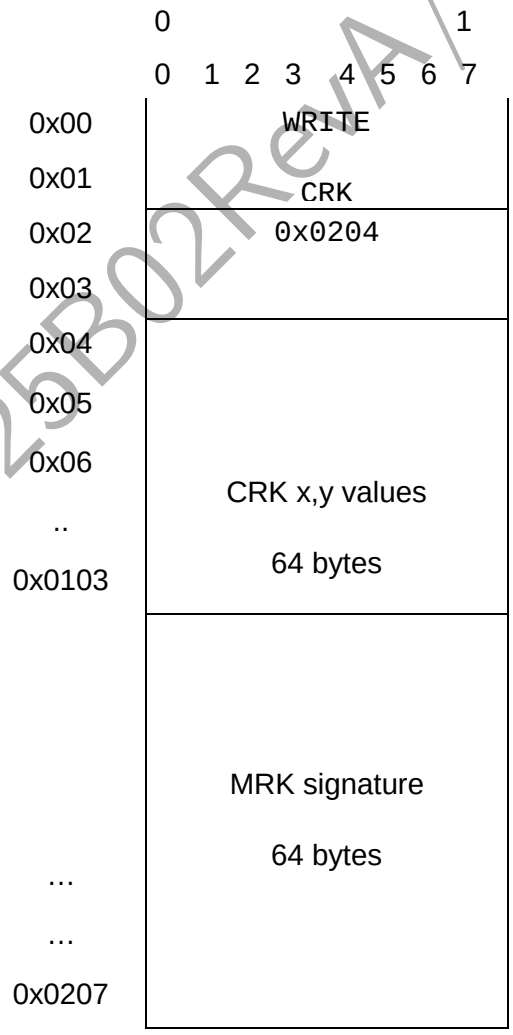


Figure 18 WRITE CRK command format

Key len: The second and third bytes are the key len. Fixed to `0x204` (64 bytes + 64 bytes)

CRK x,yvalue: The key value field is the 64-byte value of the ECDSA public key.

MRK signature value: The MRK signature of the CRK x,y value, using ECDSA-SHA256.
Note1: This MRK signature shall not be confused with the ECDSA signature of the packet (which will be also computed with the MRK).
Note2: the CRK can not be used for such command.
Note3: the REWRITE-CRK has the equivalent behavior of this command but in phase #4.

4.3.1.3 Response

A 4-byte error code is returned:

- ERR_NO
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.3.2 REWRITE-CRK Command

4.3.2.1 Definition and scope

This command rewrites a customer CRK on the chip user OTP.

Note1: this command is allowed only once
Note2: this command is not allowed with the CRK but only with the MRK.
Note3: this command is only available in phase #4.

4.3.2.2 Command Format

The REWRITE-CRK (0x461A) command is coded according the Figure 18.

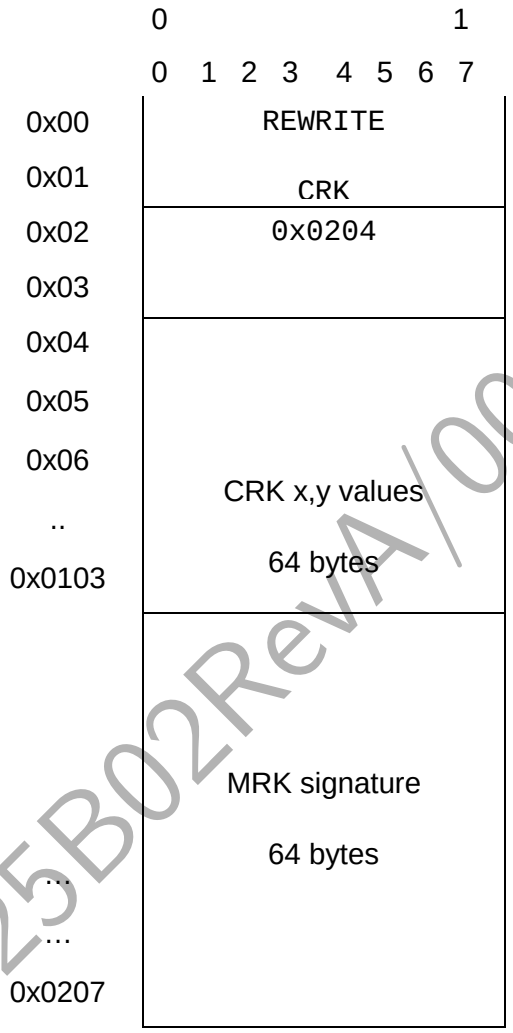


Figure 19 REWRITE CRK command format

Key len: The second and third bytes are the key len. Fixed to 0x204 (64 bytes + 64 bytes)

CRK x,yvalue: The key value field is the 64-byte value of the ECDSA public key.

MRK signature value: The MRK signature of the CRK x,y value, using ECDSA-SHA256.
Note1: This MRK signature shall not be confused with the ECDSA signature of the packet (which will be also computed with the MRK).
Note2: the CRK can not be used for such command.
Note3: the WRITE-CRK has an equivalent behavior to this command, but only in phase #3

4.3.2.3 Response

A 4-byte error code is returned:

- ERR_NO
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.3.3 WRITE-OTP Command

4.3.3.1 Definition and scope

This command writes the provided data in the chip user OTP.

Note1: if the data to be written are secret, this command shall be used either in encrypted mode or in secure environment.
Note2: this command is used for external memories configuration, customer data, …
Note3: this command permits to add USB support for SCP session; when USB and UART links are active, USB link is the preferred one (i.e. the first polled),
Note4: this command addresses user OTP only (the Maxim OTP is not accessible).

4.3.3.2 Command Format

The WRITE-OTP (0x4714) command is coded according the Figure 18.

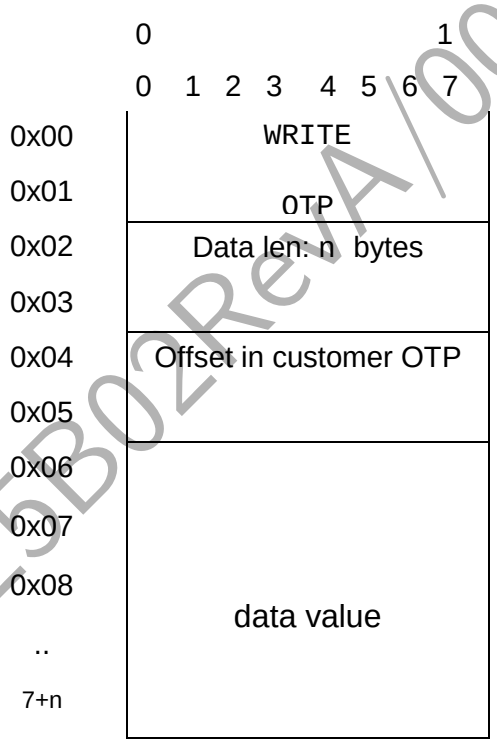


Figure 20 WRITE OTP command format

Data len: The second and third bytes are the data value length, expressed in bytes.
Offset: the fourth and fifth bytes are the offset in the customer OTP. This offset value is starting after the CRK data in the real OTP, i.e. the value 0 points to the first byte at the position 0x2C8.

Data value field: The data value field is the data value stored in OTP. This data value contains the lock bit. The CV value is not transmitted and is computed by the ROM code itself.

4.3.3.3 Response

A 4-byte error code is returned.

- ERR_NO
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.3.4 WRITE TIMEOUT Command

4.3.4.1 Definition and scope

This command sets up the value of the time frame within a session hello is waited for by the chip in phase #4. It is not possible to modify a value already written. It is also forbidden to set up the value to `0x0000`.

For example, a configuration may be the time frame of one second, that is one thousand milliseconds (data bytes = `0x000003E8`).

It is not possible to use this command twice, i.e. it is not possible to write two different values, or even clear the value.

This command applies for USB link, VBUS detect and UART links, which have distinct values.

For the VBUS detect path, the timeout is used to wait for a connection request if and only if the VBUS is detected. If the VBUS is not detected, there is no timeout window, meaning there is no time penalty on boot sequence.

Note1: when the timeout value is not set up in the OTP, a default value is used; it is 5sec.
Note2: this timeout ends counting when the HELLO-REPLY is effectively sent to the Host.
Note3: this timeout shall not be confused with the retransmission timeout as defined in section 3.3.2.3

4.3.4.2 Command Format

The WRITE TIMEOUT (`0x4426`) command is coded according to the Figure 21.

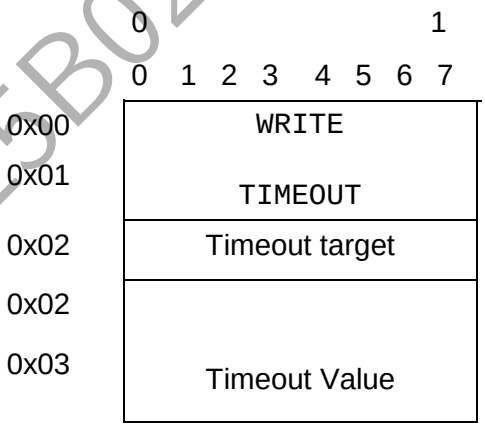


Figure 21 WRITE TIMEOUT command format

Timeout target field

The field is coded on one byte. It represents the targeted link that is set up by the timeout value field:

- the UART link corresponds to the value `0x00`,
- the USB link corresponds to the value `0x55` i.e., the char 'U'.
- the VBUS detect path corresponds to the value `0x56` i.e., the char 'V'.

Timeout value field

The field is coded on two bytes, representing the value of the timeout in milliseconds. All values between `0x0001` and `0xFFFF` are supported.

The value `0xFFFF` has a special meaning: there is no session opening and the loader does not wait for any connection. Therefore, this value is not recommended for real device as the SCP will not be available any more.

Note1: ***when USB and UART links are active, USB link is the preferred one (i.e. the first polled).***

4.3.4.3 Response

A 4-byte error code is returned.

- ERR_NO
- ERR_ALREADY
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.3.5 KILL CHIP Command

4.3.5.1 Definition and scope

This command updates the chip to phase #5. After the command is executed successfully, the chip cleans the sensitive registers and goes into shutdown mode immediately after sending the response (not waiting for the session closure).

4.3.5.2 Command Format

The KILL CHIP (0x4538) command is coded according to the Figure 21.

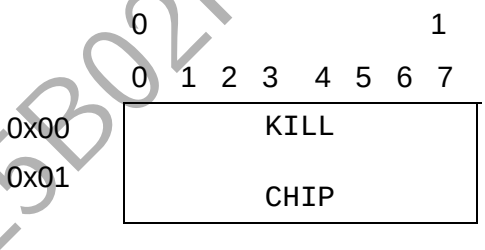


Figure 22 KILL CHIP command format

4.3.5.3 Response

A 4-byte error code is returned.

- ERR_NO
- ERR_PROHIBIT

4.4 Memories Commands

These commands are dedicated to the memories management (storage, check, erasure, execution). These commands have a generic, default behavior, able to handle simple or internal memories (RAM, embedded flash). The ROM code manages an indirection table containing three pointers to three addresses, corresponding to each of the three memory commands, WRITE DATA, ERASE DATA and COMPARE DATA. At each of these addresses, code for handling the command is available. By default, these addresses are in ROM (see Figure 23). By using the EXECUTE CODE command, the indirection table is modified and so, these addresses can be changed to addresses in internal RAM, where code previously loaded by a WRITE DATA command is substituted to the default code (see Figure 24). It makes the memory commands fully flexible. The benefit is to any external memory can be used, after this code loading and redirection, while staying in the secure context of the SCP framework.

Examples of plug-ins are available and can be modified for addressing specific memories or any other purposes (e.g. test). The applet mechanism is only limited to the three memory commands.

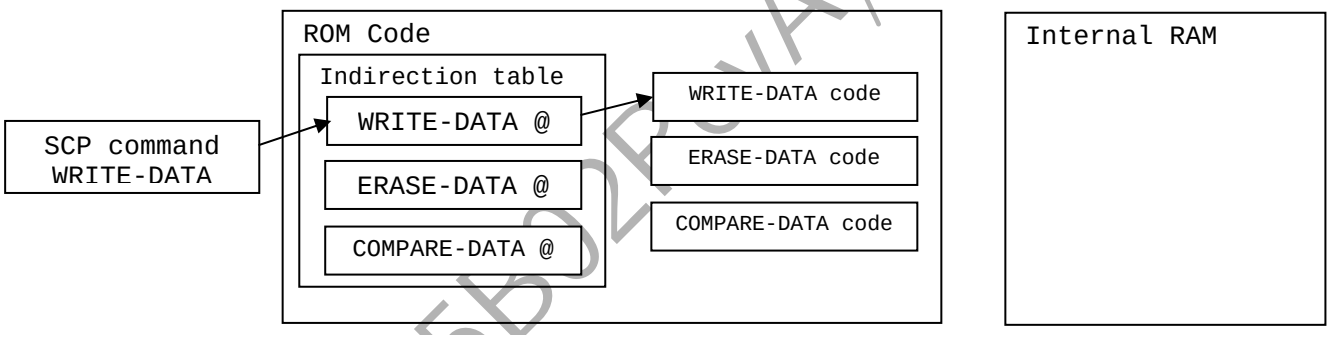


Figure 23 memory command default behavior

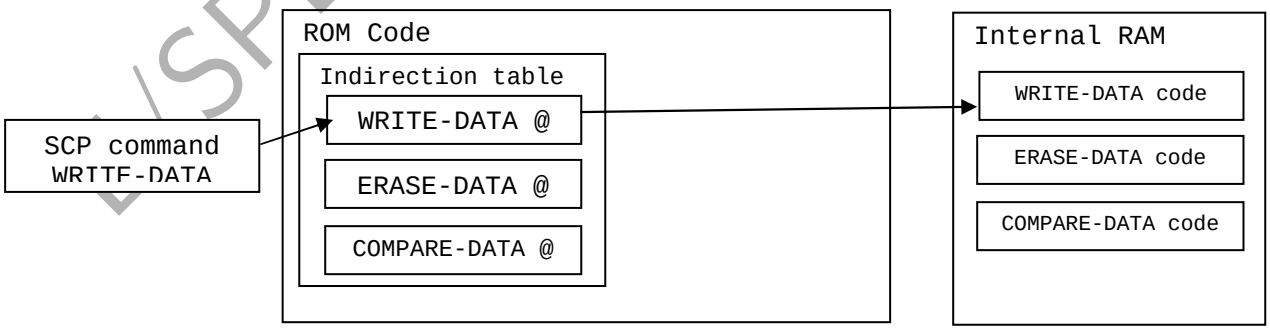


Figure 24 memory commands with loaded applet

4.4.1 WRITE DATA Command

4.4.1.1 Definition and scope

The WRITE DATA (0x2402) command is used to write data into memories.
The considered memories are the internal flashes and the internal memories.

There is no cross-checking to ensure that the area being written to does not correspond to an existing image.

Note 2: a verification operation is made implicitly after the write operation. There is so no need to perform a COMPARE DATA after a WRITE DATA

4.4.1.2 Command Format

The command is coded according the Figure 25:

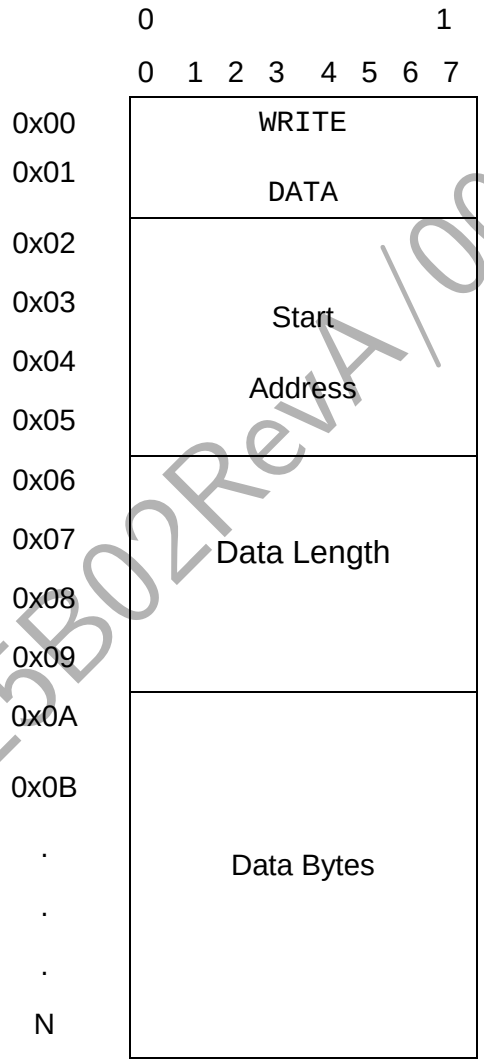


Figure 25 WRITE DATA command format

Start Address field

This field is coded on four bytes, and represents the address to begin storing data in the targeted memory.

Data Length field

This field is coded on four bytes, and represents the length of data to be stored.

Data Bytes field

This field represents the stream of data bytes to be stored.

4.4.1.3 Response

A 4-byte error code is returned.

- ERR_NO
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.4.2 COMPARE DATA Command

4.4.2.1 Definition and scope

The COMPARE DATA command (`0x2403`) is used to verify already written data into the memories. There is no cross-checking to ensure that the area being written to does not correspond to an existing image.

Note: a verification operation is made implicitly after any write operation.

4.4.2.2 Command Format

The command is coded according the Figure 25:

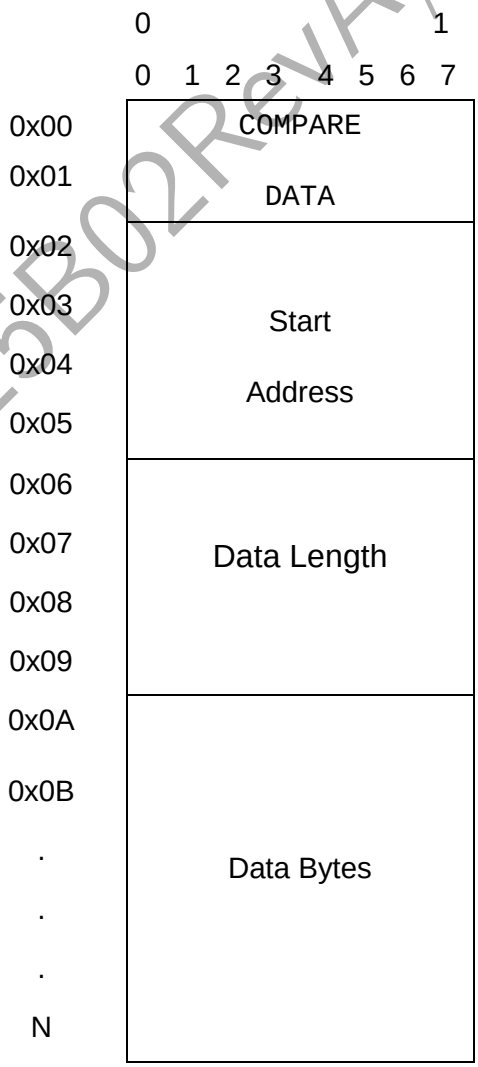


Figure 26 COMPARE DATA command format

Start Address field

This field is coded on four bytes, and represents the address to begin verifying data in the flash memory.

Data Length field

This field is coded on four bytes, and represents the length of data to be verified. Data Length Field value is at minimum 8 bytes (2 32-bit words).

Data Bytes field

This field represents the stream of data bytes to be verified and compared with.

4.4.2.3 Response

A 4-byte error code is returned.

- ERR_NO
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.4.3 ERASE DATA Command

4.4.3.1 Definition and scope

The ERASE DATA command (`0x4401`) is used to request a memory erasure (usually a flash).

4.4.3.2 Command Format

The command is coded according the Figure 27:

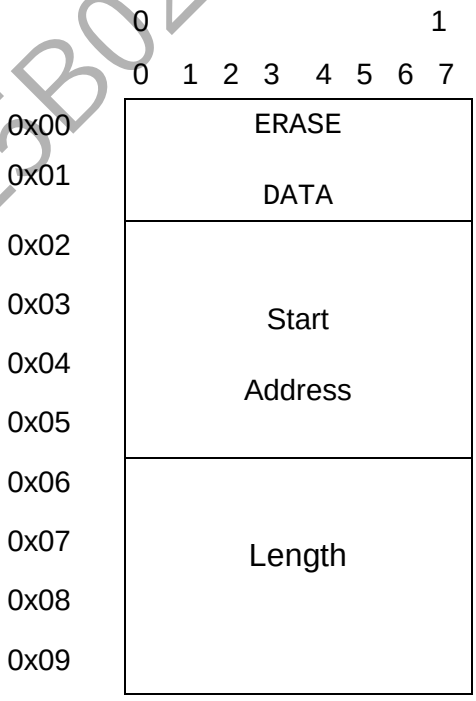


Figure 27 ERASE DATA command format

Start Address field

This field is coded on four bytes, and represents the address to begin erasing data in the targeted memory.

Data Length field

This field is coded on four bytes, and represents the length of data to be erased.

4.4.3.3 Response

A 4-byte error code is returned.

- ERR_NO
- ERR_BAD_VALUES
- ERR_PROHIBIT

4.4.4 EXECUTE CODE Command

4.4.4.1 Definition and scope

The EXECUTE CODE command (`0x2101`) is used to handle a previously loaded application. There are two main behaviors, depending on the loaded application. The way the ROM code determines which kind of application has been loaded is determined by magic word available at “Start Address”.

- If the application is identified as a applet application, to be used with memory commands (WRITE DATA, ERASE DATA, COMPARE DATA), the EXECUTE CODE behavior is the following:
  - The ROM code checks the address is in internal RAM,
  - The ROM code parses the applet header and retrieves configuration; this configuration consists in addresses for the three memory commands above. The indirection table is then updated with these addresses,
  - No applet code is executed at this time but only configuration modification; then the EXECUTE CODE returns
  - Then each time a memory command is used, the applet code is run, instead of the default code,
- If the application is identified as a final application (in that case, the application is not in the internal RAM), the launching sequence, as performed at the end of the ROM code process, is performed. Only an ACK is sent back to the Host. No disconnection information is sent, and it is not possible to come back in the session.

4.4.4.2 Command Format

The command is coded according the Figure 28:

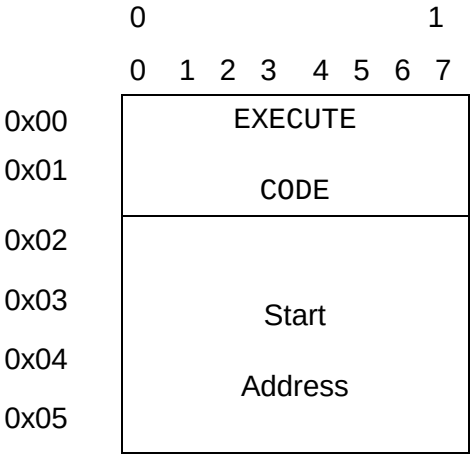


Figure 28 EXECUTE CODEcommand format

Start Address field

This field is coded on four bytes, and represents the address to begin parsing for application header in the targeted memory.

4.4.4.3 Response

- For applet code, a 4-byte error code is returned:
  - ERR_NO
  - ERR_BAD_VALUES
  - ERR_PROHIBIT
- For final application, no response is provided as the EXECUTE_CODE command never goes back from the run application.

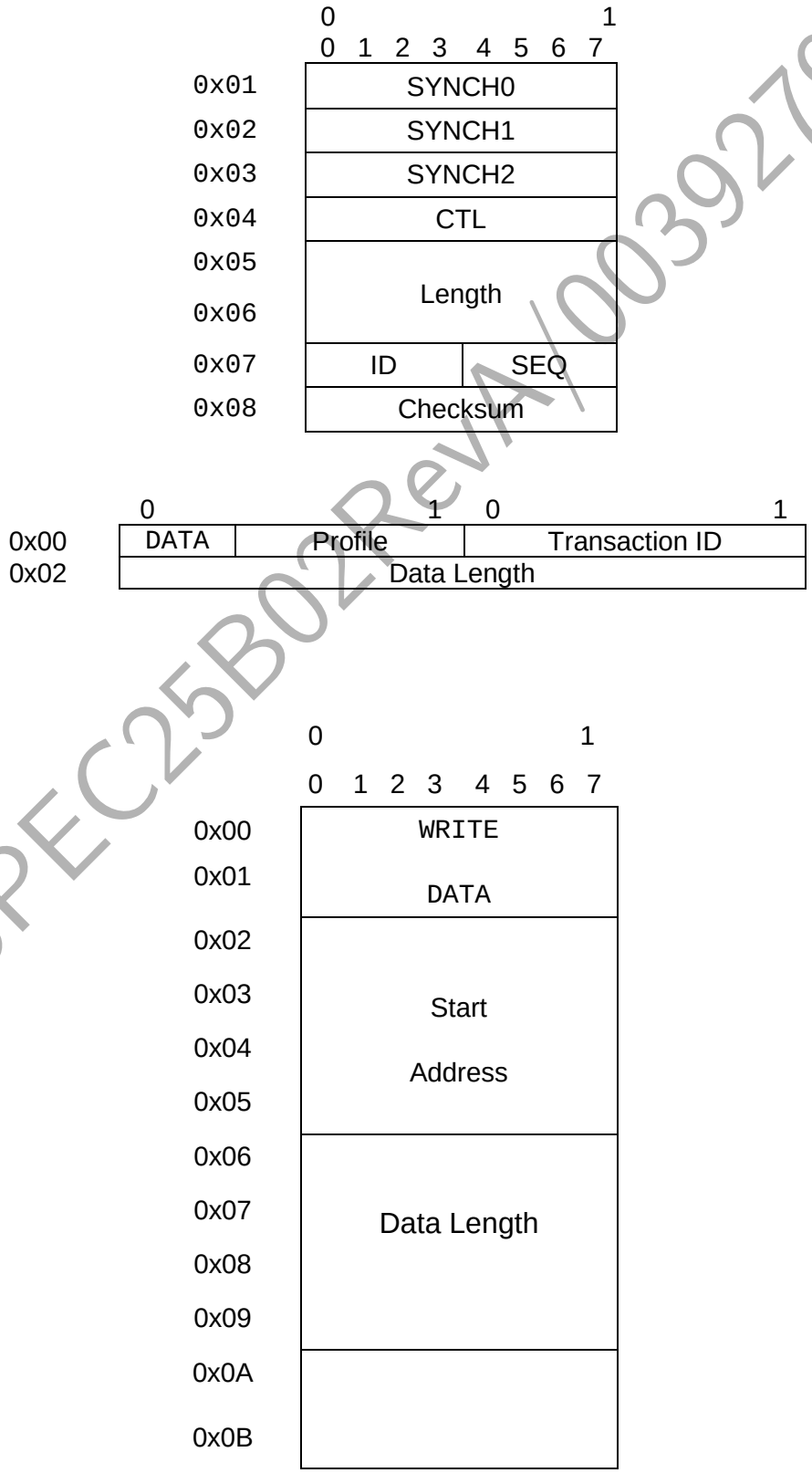
5 Annex A: typical scenarios

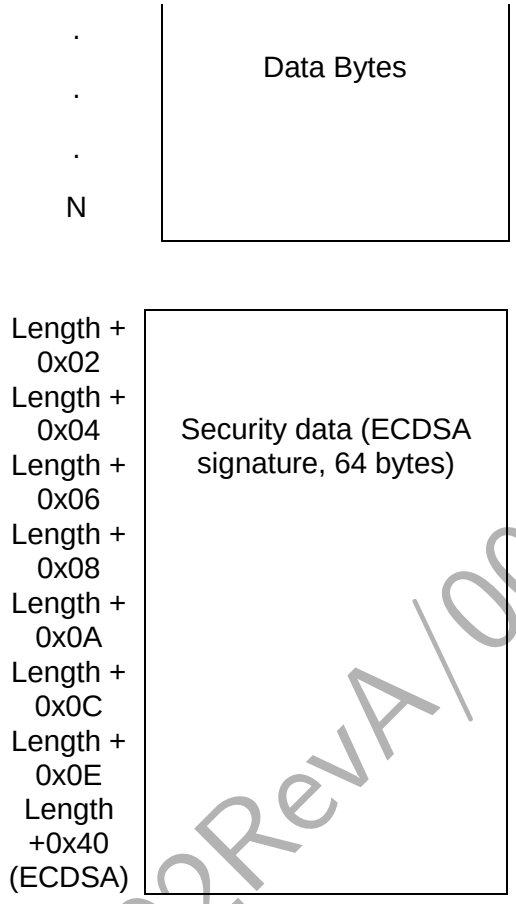
The typical scenario of use of the SCP is the following:

- Chips are sent with an empty internal flash to the customer, in phase #3,
- The customer personalizes the chip by using SCP script, using the UART:
  - write-crk; chip goes to phase #4
  - write-data application.bin, internal flash
  - write-timeout 2seconds
- the customer puts the device in the field
  - start the chip
  - wait for 2seconds a SCP session request
  - start final application in internal flash

6 Packets examples

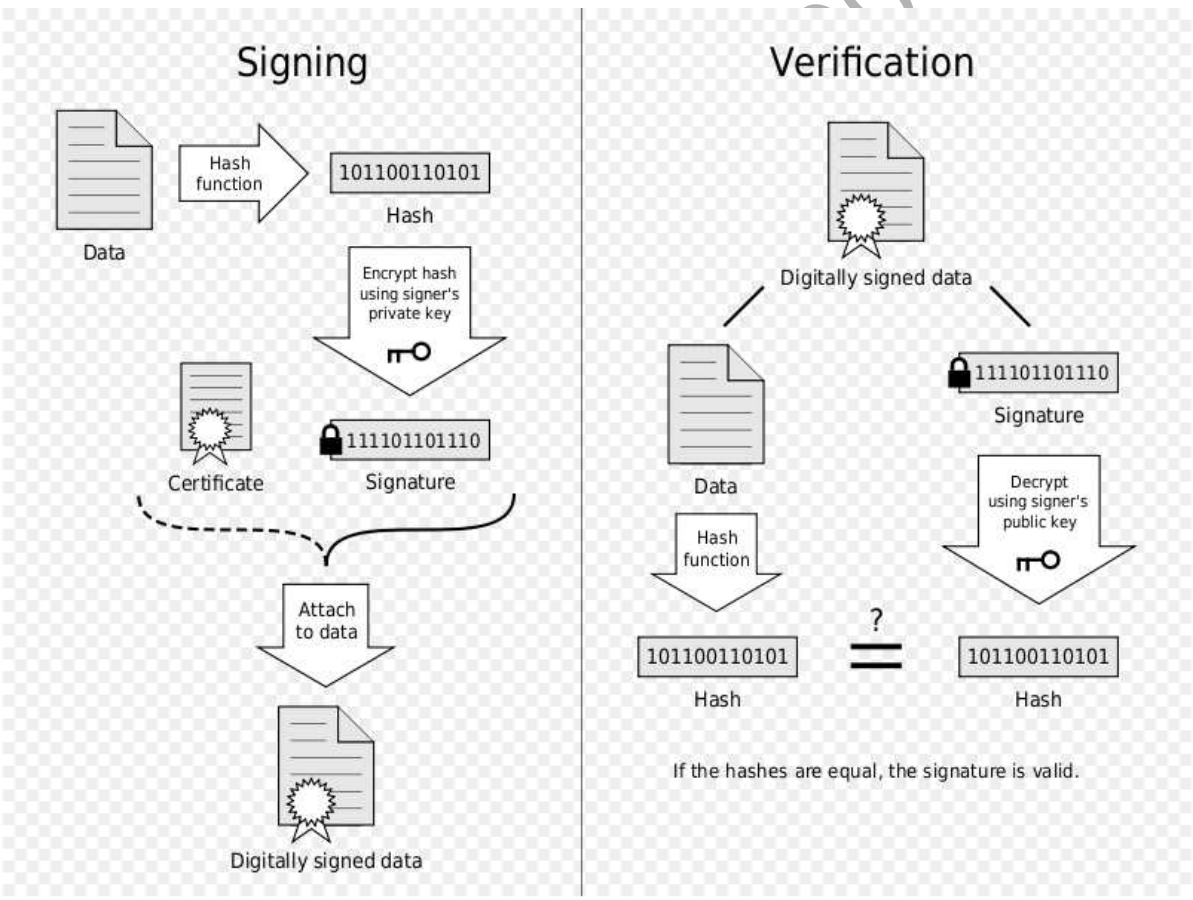
6.1 Full frame for WRITE-DATA command





7 Annex B: ECDSA signature

ECDSA Keys are 256-bit long (as required by [2]) The principle of one key for one use is enforced.
The ECDSA keys are used for digital signatures.
A digital signature scheme typically consists of three algorithms (extracted from [9]):
A key generation algorithm that selects a private key randomly from a set of possible private keys. The algorithm outputs the private key and a corresponding public key.
A signing algorithm which, given a message and a private key, produces a signature.
A signature verifying algorithm which given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.
The diagram below (Illustration 3 extracted from [9]) describes a digital signature scheme, applicable to the ECDSA signatures.



MAXIM INTEGRATED Reference

http://www.maximintegrated.com
support: goto www.maximintegrated.com/support

