C-API V2

MULTOS Standard C-API

MAO-DOC-TEC-016 v2.4

Copyright

Trademarks

Published by

MULTOS Limited,
350 Longwater Avenue,
Reading,
Berkshire,
RG2 6GF,
UK

General Enquiries

Email: dev.support@multos.com
Web: https://www.multos.com

Document References

All references to other available documentation is followed by the document acronym in square [ ] brackets. Details of the content of these documents can be found in the MULTOS Guide to Documentation, and the latest versions are always available from the MULTOS web site (https://www.multos.com).

Data References

All references to MULTOS data can be cross-referenced to the MULTOS Data Dictionary.

Contents

1 Introduction

The MULTOS Standard C API is intended to standardise the syntax used by C developers writing MULTOS applications. It is included as part of the SmartDeck SDK and is accessed by using the include file **multos.h**.

The Standard C API is meant to cover the needs of most C developers however, developers may still use assembler or development tool supplied C functions not included in this document.

IMPORTANT: This document describes version 2 of the C-API, introduced in March 2017. The new API replaces the vast majority of the compiler macros used by the original version of the C-API and replaces them with standard 'C' style function prototypes. The standard 'C' library included with SmartDeck v3.1.0 and later implements these functions using bytecode substitution (thus avoiding an actual function call).

To compile existing applications using the old API either:

- Use the old version of multos.h (1.4.1 or earlier) OR
- Add **–DCAPIV1** to the compiler command line.

A mapping of the old to new API can be found in Appendix B.

Only functions that provide an interface to a MULTOS instruction or primitive are included.

The Appendices to this document also include industry specific C APIs developed by Application Developers and contributed to this specification. The MULTOS Consortium welcomes the submission of new C APIs to this specification from the Application Developer community.

2 Exclusions

For the sake of clarity and simplicity only mainstream MULTOS functions are included in the Standard C API. Some less frequently used functions are excluded. APIs for the following MULTOS instructions and primitives are not included.

Instruction or Primitive	Reason for exclusion
ADDB	handled by multosAdd
ADDW	handled by multosAdd
BRANCH	C programming handles this implicitly
CALL	C programming handles this implicitly
CMPB	C programming handles this implicitly
CMPN	C programming handles this implicitly
CMPBW	C programming handles this implicitly
INDEX	C programming handles this implicitly
JUMP	C programming handles this implicitly
LOAD	C programming handles this implicitly
LOADA	C programming handles this implicitly
LOADI	C programming handles this implicitly
PRIMRET	API provides primitive interface
SETB	C programming handles this implicitly
SETW	C programming handles this implicitly
STACK	C programming handles this implicitly
STORE	C programming handles this implicitly
STOREI	C programming handles this implicitly
SUBB	handled by multosSubtract
SUBW	handled by multosSubtract
Bit Manipulate Byte	handled by included binary and unary instructions
Bit Manipulate Word	handled by included binary and unary instructions
Get Purse Type	not required by most programmers
Load CCR	not required by most programmers
Store CCR	not required by most programmers

3 Notes

3.1 Data Types

The following Data Types are used:

Data Type	Definition
BOOL	boolean (byte)
BYTE	unsigned byte (byte)
SBYTE	signed byte (byte)
WORD	unsigned word (2 bytes)
SWORD	signed word (2 bytes)
DWORD	unsigned double word (4 bytes)
SDWORD	signed double word (4 bytes)
QWORD	unsigned long long (8 bytes)

3.2 Conventions

The conventions used in this document are:

- All function names start with "multos".
- The keyword "const" is used to indicate whether the parameter must be a compile-time constant (i.e. not a value held in a variable).

3.3 System Variables

Type	Name	Description
BYTE	multosProtocolFlags	Protocol flags in Public, encoded as follows: MULTOS_MASK_P3_VALID, MULTOS_MASK_LC_VALID, MULTOS_MASK_LE_VALID, MULTOS_MASK_CMD_DATA_RECEIVED
BYTE	multosProtocolType	Protocol type in Public, encoded as follows: MULTOS_PROTOCOL_T0, MULTOS_PROTOCOL_T1
BYTE	multosGetResponseCLA	Get Response CLA in Public
BYTE	multosGetResponseSW1	Get Response SW1 in Public
BYTE	multosCLA	CLA in Public
BYTE	multosINS	INS in Public
BYTE	multosP1	P1 in Public
BYTE	multosP2	P2 in Public
BYTE	multosP3	P3 in Public
WORD	multosP1P2	P1P2 in Public
WORD	multosLc	Lc in Public

WORD	multosLe	Le in Public
WORD	multosLa	La in Public
BYTE	multosSW1	SW1 in Public
BYTE	multosSW2	SW2 in Public
WORD	multosSW12	SW1 and SW2 in Public

4 Function Prototypes

Function prototypes in blue are implemented in libc.hza using bytecode substitution. Function prototypes in green are implemented as macros.

4.1 multosAcceleratedReadNoBAC

BOOL **multosAcceleratedReadNoBAC** (BYTE *dataAddr, BYTE channel);

The parameters are:

- BYTE channel: always set to 0 (input)
- BYTE *dataAddr: address of the 8 byte parameter block for the channel (input)

Returns TRUE if the command is successful.

This calls the primitive Configure Read Binary to activate accelerated read mode without the use of secure messaging.

4.2 multosAcceleratedReadBAC

BOOL **multosAcceleratedReadBAC** (const BYTE options, BYTE *dataAddr, BYTE channel,
BYTE *keyEncAddr, BYTE *keyMacAddr, BYTE *sscAddr);

The parameters are:

- BYTE* sscAddr: address of an 8 byte counter used in the MAC computation
- BYTE* keyMacAddr: address of the 16 byte MAC key
- BYTE* keyEncAddr: address of the 16 byte ENC key
- BYTE channel: always set to 0
- BYTE* dataAddr: address of the 8 byte parameter block for the channel
- const BYTE options: as defined in [MDRM]

Returns TRUE if the command is successful.

This calls the primitive Configure Read Binary to activate accelerated read mode with secure messaging.

4.3 multosAcceleratedReadBACLite

BOOL **multosAcceleratedReadBACLite** (const BYTE options, BYTE *dataAddr, BYTE channel,
BYTE *keyEncAddr);

The parameters are:

- BYTE* keyEncAddr: address of the 16 byte ENC key
- BYTE channel: always set to 0
- BYTE* dataAddr: address of the 8 byte parameter block for the channel

- const BYTE options: as defined in [MDRM]

Returns TRUE if the command is successful.

This calls the primitive Configure Read Binary to activate accelerated read mode with encryption only (no MAC).

4.4 multosAdd

void **multosAdd** (const BYTE blockLength, BYTE *block1, BYTE *block2, BYTE *result)

The parameters are:

- const BYTE *blockLength*: size of the blocks to add.
- BYTE **block1*: address of the first block
- BYTE **block2*: address of the second block
- BYTE **result*: address of the block that will hold the result of the operation

This function adds the value found in *block1* to that found in *block2* and places the sum in the block indicated in the *result* parameter. Note that *block1*, *block2* and *result* are all considered to be of size *blockLength*.

This is an interface to the instruction ADDN.

4.5 multosAnd

void **multosAnd** (const BYTE blockLength, BYTE *block1, BYTE *block 2, BYTE *result)

The parameters are:

- const BYTE *blockLength*: size of the blocks to add. Both blocks must be the same size.
- BYTE **block1*: address of the first block
- BYTE **block2*: address of the second block
- const BYTE **result*: address of the block that will hold the result of the operation

This function performs a logical AND using the values found in *block1* and *block2*. The result is written the location given in the *result* parameter. Note that *block1*, *block2* and *result* are all considered to be of size *blockLength*.

This is an interface to the instruction ANDN.

4.6 multosBCDtoBIN

BOOL **multosBCDtoBIN** (BYTE *sourceAddr, BYTE *destAddr, BYTE sourceLen, BYTE destLen);

The parameters are:

- BYTE sourceLen: length of data to convert (input)
- BYTE destLen: length of output buffer pointed to by destAddr (input)
- BYTE *sourceAddr: pointer to buffer containing the data to convert (input)
- BYTE *destAddr: pointer to output buffer to hold result of conversion (output)

This function converts between binary coded decimal and binary. It returns TRUE if the conversion is successful.

This is an interface to the primitive Convert BCD.

4.7 multosBINtoBCD

BOOL **multosBINtoBCD** (BYTE *sourceAddr, BYTE *destAddr, BYTE sourceLen, BYTE destLen);

The parameters are:

- BYTE sourceLen: length of data to convert (input)
- BYTE destLen: length of output buffer pointed to by destAddr (input)
- BYTE *sourceAddr: pointer to buffer containing the data to convert (input)
- BYTE *destAddr: pointer to output buffer to hold result of conversion (output)

This function converts between binary and binary coded decimal. It returns TRUE if the conversion is successful.

This is an interface to the primitive Convert BCD.

4.8 multosCallCodelet

void **multosCallCodelet** (BYTE *codeAddress, WORD codeletID);

The parameters are:

- WORD *codeletID*: the unique identifier of the codelet being called
- BYTE * *codeAddress*: codelet entry point

This function invokes a codelet and executes the code starting at the entry address.

This is an interface to the primitive Call Codelet.

4.9 multosCallExtensionPrimitive

void **multosCallExtensionPrimitive** (const BYTE extensionNum, const BYTE primTypeLo, const BYTE primTypeHi, const BYTE paramByte)

The parameters are:

- const BYTE *extensionNum*: number indicating the MULTOS implementor
- const BYTE *primTypeLo*: least significant byte of the proprietary primitive identifier as allocated by the implementor
- const BYTE *primTypeHi*: most significant byte of the proprietary primitive identifier as allocated by the implementor
- const BYTE *paramByte*: byte that may be used to pass a parameter to the proprietary primitive

This function calls a proprietary extension primitive.

This is an interface to the primitive Call Extension.

4.10 multosCardBlock

BOOL **multosCardBlock** (**const** BYTE MSB_StartAddress_MAC, **const** BYTE LSB_StartAddress_MAC);

The parameters are:

- const BYTE *MSB_StartAddress_MAC*: most significant byte of the address of the CBMAC in public memory
- const BYTE *LSB_StartAddress_MAC*: least significant byte of the address of the CBMAC in public memory

This function evaluates a card block MAC and blocks the MULTOS card if the MAC is verified (returning TRUE) otherwise it returns FALSE.

This is an interface to the primitive Card Block.

4.11 multosCardUnBlock

BOOL **multosCardUnBlock** (void);

Calls the primitive Card Unblock and returns TRUE if the card is successfully unblocked.

4.12 multosCheckCase

BOOL **multosCheckCase** (BYTE isoCase)

The parameter is a single byte indicating the expected ISO case of the incoming command.

This function (a) instructs the operating system as to how to interpret the APDU received and (b) checks the ISO case of the received command. It returns TRUE if the case is recognised.

This is an interface to the primitive Check Case.

4.13 multosCheckBCD

BYTE **multosCheckBCD** (BYTE *address, BYTE length);

The parameters are:

- BYTE *length*: the length (in bytes) of the block containing the value to be checked
- BYTE **address*: the address of the block containing the value to be checked

The function checks if the provided block contains a valid Binary Coded Decimal value. Retuns 1 if the value is BCD, 0 if not.

This is an interface to the primitive Check BCD.

4.14 multosChecksum

DWORD **multosChecksum** (BYTE *blockAddr, WORD length);

The parameters are:

- WORD *length*: the length of the block to use as input to the checksum algorithm
- BYTE **blockAddr*: the address of the first byte of the input block

The function generates a four-byte checksum of the data block.

This is an interface to the primitive Checksum.

4.15 multosClear

void **multosClear** (const BYTE blockLength, const BYTE *block)

The parameters are:

- const BYTE *blockLength*: size of the block to clear
- const BYTE **block*: address of the block to be cleared

This function sets the value of each byte of the block of size *blockLength* to zero.

This is an interface to the instruction CLEARN.

4.16 multosCompare

BYTE **multosCompare** (BYTE *addr2, BYTE *addr1, WORD length);

The parameters are:

- WORD *length*: size of the blocks to be compared. Both blocks must be the same size.
- BYTE **addr1*: address of the first block
- BYTE **addr2*: address of the second block

This function compares block1 and block2, returns one of the following results:

- MULTOS_BLOCK1_GT_BLOCK1
- MULTOS_BLOCK2_GT_BLOCK1
- MULTOS_BLOCK1_EQ_BLOCK2

Note that the blocks at *addr1* and *addr2* are considered to be of size *blockLength*.

This is an interface to the primitive Memory Compare.

Note: The function **multosCompareFixedLength** (const BYTE length, BYTE* addr2, BYTE* addr1) also exists but uses the primitive *Memory Compare Fixed Length* instead.

4.17 multosCompareEnhanced

WORD **multosCompareEnhanced** (const BYTE mode, BYTE *addr2, BYTE *addr1, WORD length);

The parameters are:

- const BYTE mode: 0 = equality only test, 1 = equality and greater/less than test
- WORD *length*: size of the blocks to be compared. Both blocks must be the same size.
- BYTE **addr1*: address of the first block
- BYTE **addr2*: address of the second block

Equality Only Test

The two memory areas are tested for equality and the function can return one of the following values:-

- 0x5555 = blocks not equal

- 0xAAAA = blocks equal

Equality and Greater/Less Than Test
The comparison is based on subtraction and the function can return one of the following values:-

- 0x5A5A = byte block at *addr1* > byte block at *addr2*
- 0xA5A5 = byte block at *addr1* < byte block at *addr2*
- 0xAAAA = blocks equal

This is an interface to the primitive Memory Compare Enhanced.

4.18 multosCopy

void **multosCopy** (BYTE *sourceAddr, BYTE *destAddr, WORD length);

The parameters are:

- WORD *length*: the size of the block to copy
- BYTE **sourceAddr*: address of the source block
- BYTE **destAddr*: address of the destination block

This function copies data of size *length* from *sourceAddr* to *destAddr.*

This is an interface to the primitive Memory Copy.

Note 1: multosCopyNonAtomic performs a non-atomic copy. See [MDRM] for details and a definition of non-atomic memory operations.

4.19 multosCopyFixedLength

void **multosCopyFixedLength** (const BYTE length, BYTE *sourceAddr, BYTE *destAddr);

The parameters are:

- BYTE *length*: the size of the block to copy
- BYTE **sourceAddr*: address of the source block
- BYTE **destAddr*: address of the destination block

This function copies data of size *length* from *sourceAddr* to *destAddr.*

This is an interface to the primitive Memory Copy Fixed Length.

Note 1: multosCopyFixedLengthNonAtomic performs a non-atomic copy. See [MDRM] for details and a definition of non-atomic memory operations.

4.20 multosCopyFromAdditionalStatic

void **multosCopyFromAdditionalStatic** (DWORD staticOffset, BYTE *segAddr, WORD length);

The parameters are:

- BYTE *segAddr: address in “normal” memory to copy to (input)
- DWORD staticOffset: the offset in “additional” static* to copy from (input)
- WORD length: the number of bytes to copy (input)

This copies a block of bytes from Additional Static memory to a location in one of the normal memory segments (static, session, public or local). A version of this function, **multosCopyFromAdditionalStaticAtomic**, with the same parameters, performs an atomic copy.

* Note: Refer to the MULTOS Developers’ Guide [MDG] for more information about Additional Static memory and how to use it.

This is an interface to the primitive Memory Copy Additional Static.

4.21 multosCopySessionFromReplacedApp

void **multosCopySessionFromReplacedApp** (WORD sessionOffset, BYTE *segAddr, WORD length);

The parameters are:

- WORD *sessionOffset*: An offset into the session memory of the application being replaced.
- BYTE **segAddr:* The destination to copy the memory to.
- WORD *length*: The length of data to copy.

This function allows for the currently executing application to copy the Session data belonging to the application that it is replacing. It can only be called when an application is being called as part of an application replacement event. This can be determined by calling *multosGetReplacedAppState()*.

The copy function is non-atomic. For an atomic copy, use *multosCopySessionFromReplacedAppAtomic()* which has identical parameters.

This is an interface to the primitive Memory Copy From Replaced Application.

4.22 multosCopyStaticFromReplacedApp

void **multosCopyStaticFromReplacedApp** (DWORD staticOffset, BYTE *segAddr, WORD length);

The parameters are:

- DWORD *staticOffset*: An offset into the static memory of the application being replaced.
- BYTE **segAddr:* The destination to copy the memory to.
- WORD *length*: The length of data to copy.

This function allows for the currently executing application to copy the Static data belonging to the application that it is replacing. It can only be called when an application is being called as part of an application replacement event. This can be determined by calling *multosGetReplacedAppState()*.

The copy function is non-atomic. For an atomic copy, use *multosCopyStaticFromReplacedAppAtomic()* which has identical parameters.

This is an interface to the primitive Memory Copy From Replaced Application.

4.23 multosCopyToAdditionalStatic

void **multosCopyToAdditionalStatic** (BYTE *segAddr, DWORD staticOffset, WORD length);

The parameters are:

- BYTE *segAddr: address in “normal” memory to copy from (input)
- DWORD staticOffset: the offset in “additional” static* to copy to (input)
- WORD length: the number of bytes to copy (input)

This copies a block of bytes from one of the normal memory segments (static, session, public or local) to Additional Static memory. A version of this function, **multosCopyToAdditionalStaticAtomic**, with the same parameters, performs an atomic copy.

* Note: Refer to the MULTOS Developers’ Guide [MDG] for more information about Additional Static memory and how to use it.

This is an interface to the primitive Memory Copy Additional Static.

4.24 multosCopyWithinAdditionalStatic

void **multosCopyWithinAdditionalStatic** (DWORD srcOffset, DWORD destOffset, DWORD length);

The parameters are:

- DWORD destOffset: the offset in “additional” static* to copy to (input)
- DWORD srcOffset: the offset in “additional” static* to copy (input)
- DWORD length: the number of bytes to copy (input)

This function copies a block of bytes from one location in Additional Static memory to another. A version of this function, **multosCopyWithinAdditionalStaticAtomic**, with the same parameters, performs an atomic copy.

* Note: Refer to the MULTOS Developers’ Guide [MDG] for more information about Additional Static memory and how to use it.

This is an interface to the primitive Memory Copy Additional Static.

4.25 multosDeactivateAcceleratedRead

BOOL **multosDeactivateAcceleratedRead** (void);

This calls the primitive Configure Read Binary to deactivate accelerated read mode for the Read Binary command and returns TRUE if the operation was successful.

4.26 multosDecipherCBC

void **multosDecipherCBC** (**const** BYTE algo, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen, BYTE *ivAddr, BYTE ivLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- WORD inputLength: Length of cipher text (input)
- BYTE *inputAddr: pointer to cipher text to decipher (input)
- BYTE *outputAddr: pointer to memory in which to write the plain text output. (output)
- BYTE ivLen: length of the Initial Chaining Vector pointed to by ivAddr (input)
- BYTE *ivAddr: Pointer to ICV value (input)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)

This function deciphers the cipher text using the cipher block chaining method and supports a number of algorithms.

This is an interface to the primitive Block Decipher.

4.27 multosDecipherCFB

void **multosDecipherCFB** (**const** BYTE algo, BYTE counterWidth, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen, BYTE *ivAddr, BYTE ivLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- BYTE feedbackSize: the number of bits to use as feedback (max = blocksize of algo) (input)
- WORD inputLength: Length of cipher text (input)
- BYTE *inputAddr: pointer to cipher text to decipher (input)
- BYTE *outputAddr: pointer to memory in which to write the plain text output. (output)
- BYTE ivLen: length of the Initial Chaining Vector pointed to by ivAddr (input)
- BYTE *ivAddr: Pointer to ICV value (input)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)

This function deciphers the cipher text using CFB mode and supports a number of algorithms.

This is an interface to the primitive Block Decipher.

4.28 multosDecipherCTR

void **multosDecipherCTR** (**const** BYTE algo, BYTE counterWidth, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen, BYTE *ivAddr, BYTE ivLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- BYTE counterWidth: width of counter in bytes up to *ivLen* bytes (input)
- WORD inputLength: Length of cipher text (input)
- BYTE *inputAddr: pointer to cipher text to decipher (input)
- BYTE *outputAddr: pointer to memory in which to write the plain text output. (output)
- BYTE ivLen: length of the Initial Chaining Vector pointed to by ivAddr (input)
- BYTE *ivAddr: Pointer to ICV value (input)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)

This function deciphers the cipher text using CTR mode as specified in ISO/IEC-10116 and supports a number of algorithms.

This is an interface to the primitive Block Decipher.

4.29 multosDecipherECB

void **multosDecipherECB** (**const** BYTE algo, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- WORD inputLen: Length of cipher text (input)
- BYTE *inputAddr: pointer to cipher text to decipher (input)
- BYTE *outputAddr: pointer to memory in which to write the plain text output. (output)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)

This function deciphers the cipher text using the electronic code book method and supports a number of algorithms.

This is an interface to the primitive Block Decipher.

4.30 multosDecrement

void **multosBlockDecrement** (const BYTE blockLength, const BYTE *block)

The parameters are:

- const BYTE *blockLength*: size of the block on which to perform the operation
- const BYTE **block*: address of the block on which to perform the operation

This function decrements the value held in *block* by one. This is an interface to the instruction DECN.

4.31 multosDelegate

void **multosDelegate** (BYTE *aidAddr);

The parameter is a pointer to a buffer containing the AID of the application that will be delegated to. The first byte of the buffer must contain the length of the AID.

This function delegates execution to an application with the given AID.

This is an interface to the primitive Delegate.

4.32 multosDisableAutoResetWWT

void **multosDisableAutoResetWWT**(void);

This is an interface to the primitive Control Auto Reset WWT.

4.33 multosDivide

void **multosDivide** (const BYTE blockLength, BYTE *numerator, BYTE *denominator, BYTE *quotient, BYTE *remainder)

The parameters are:

- const BYTE *blockLength*: the size of all operand and result blocks (input)
- BYTE **numerator*: address of the block where the numerator is held (input).
- BYTE **denominator*: address of the block where the denominator is held (input).
- BYTE **quotient*: address of the block where the quotient is to be written (output).
- BYTE **remainder*: address of the block where the remainder is to be written (output).

This function divides the *numerator* by the *denominator*. The results of the operation are written to the blocks *quotient* and *remainder*.

This is an interface to the primitive DivideN.

4.34 multosEccGenerateKeyPair

BOOL **multosEccGenerateKeyPair** (BYTE *keyAddr, BYTE *domainAddr);

The parameters are:

- BYTE *keyAddr: address of buffer (size = 3 x prime length) to hold generated key pair (output)
- BYTE *domainAddr: address of domain parameters structure (input)

This generates and ECC key pair for the given domain parameters. The domain parameter structure is described in [MDRM]. The function returns TRUE if a key pair was successfully generated. A version of this function, multosEccGenerateKeyPairProtected, internally encrypts the private key.

This is an interface to the primitive ECC Generate Key Pair.

4.35 multosECDH

BOOL **multosECDH** (BYTE *sharedAddr, BYTE *publicKeyAddr, BYTE *privateKeyAddr, BYTE *domainAddr);

The parameters are:

- BYTE *domainAddr: address of domain parameters structure (input)
- BYTE *privateKeyAddr: private part of key A (input)
- BYTE *publicKeyAddr: public part of key B (input)
- BYTE *sharedAddr: address of buffer to store computed shared secret key (output)

Computes a shared key using a private key and a public key belonging to another key-pair. If the private key is protected (i.e. created using multosEccGenerateKeyPairProtected) then the function multosECDHProtected must be used instead. Both functions return TRUE if the shared key was successfully calculated.

This is an interface to the primitive ECC Elliptic Curve Diffie Hellman.

4.36 multosECDSA

BOOL **multosECDSA** (BYTE *sigAddr, BYTE *hashAddr, BYTE *privateKeyAddr, BYTE *domainAddr);

The parameters are:

- BYTE *domainAddr: address of domain parameters structure (input)
- BYTE *privateKeyAddr: address of the private key to use for signing (input)
- BYTE *hashAddr: address of the hash to sign (input)
- BYTE *sigAddr: address of buffer to hold the signature (output)

This performs an Elliptic Curve Digital Signature Algorithm signature of the hash data provided. The buffer pointed to by sigAddr must be twice the key's prime length. If the private key is protected (i.e. created

using multosEccGenerateKeyPairProtected) then the function multosECDSAProtected must be used instead. Both functions return TRUE if the signature was successfully calculated.

This is an interface to the primitive ECC Generate Signature.

4.37 multosECDSAVerify

BOOL **multosECDSAVerify** (BYTE *hashAddr, BYTE *sigAddr, BYTE *publicKeyAddr, BYTE *domainAddr);

The parameters are:

- BYTE *domainAddr: address of domain parameters structure (input)
- BYTE *hashAddr: address of the hash to compare (input)
- BYTE *sigAddr: address of the signature to decrypt and compare to the given hash (input)
- BYTE *publicKeyAddr: address of the public key to use to decrypt the signature (input)

The function returns TRUE if the signature was validated.

This is an interface to the primitive ECC Verify Signature.

4.38 multosECIESEncipher

BOOL **multosECIESEncipher** (const BYTE options, BYTE *outputAddr, BYTE *messageAddr, BYTE *publicKeyAddr, WORD length, BYTE *domainAddr);

The parameters are:

- BYTE *domainAddr: address of domain parameters structure (input)
- WORD length: the length of the message to encipher (input)
- BYTE *publicKeyAddr: address of the public key to use (input)
- BYTE *messageAddr: address of the message to encipher (input)
- BYTE *outputAddr: address of the buffer to hold the ciphertext (output)
- const BYTE options: refer to [MDRM] (input)

Returns TRUE if the process is successful.

This is an interface to the primitive ECC ECIES Encipher.

4.39 multosECIESDecipher

BOOL **multosECIESDecipher** (**const** BYTE options, BYTE *messageAddr, BYTE *inputAddr, BYTE *privateKeyAddr, WORD length, BYTE *domainAddr);

The parameters are:

- BYTE *domainAddr: address of domain parameters structure (input)
- WORD length: the length of the message to decipher (input)
- BYTE *privateKeyAddr: address of the key to use (input)

- BYTE *inputAddr: address of ciphertext (input)
- BYTE *messageAddr: address of buffer to hold cleartext (output)
- const BYTE options: refer to [MDRM] (input)

Returns TRUE if the process is successful.

This is an interface to the primitive ECC ECIES Decipher.

4.40 multosEnableAutoResetWWT

void **multosEnableAutoResetWWT**(void);

This is an interface to the primitive Control Auto Reset WWT.

4.41 multosEncipherCBC

void **multosEncipherCBC** (**const** BYTE algo, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen, BYTE *ivAddr, BYTE ivLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- WORD inputLength: Length of plain text (input)
- BYTE *inputAddr: pointer to plain text to encipher (input)
- BYTE *outputAddr: pointer to memory in which to write the cipher text output. (output)
- BYTE ivLen: length of the Initial Chaining Vector pointed to by ivAddr (input)
- BYTE *ivAddr: Pointer to ICV value (input)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)

This function enciphers the plain text using the cipher block chaining method and supports a number of algorithms.

This is an interface to the primitive Block Encipher.

4.42 multosEncipherCFB

void **multosEncipherCFB** (**const** BYTE algo, BYTE feedbackSize, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen, BYTE *ivAddr, BYTE ivLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- BYTE feedbackSize: number of bits to be used for feedback (max is block size of the algo) (input)
- BYTE *inputAddr: pointer to plain text to encipher (input)
- BYTE *outputAddr: pointer to memory in which to write the cipher text output. (output)
- BYTE keyLen: length of key, depends on algorithm being used (input)

- BYTE *keyAddr: pointer to the key to use (input)
- WORD inputLength: Length of plain text (input)
- BYTE *ivAddr: Pointer to ICV value (input)
- BYTE ivLen: length of the Initial Chaining Vector pointed to by ivAddr (input)

This function enciphers the plain text using the CFB mode and supports a number of algorithms.

This is an interface to the primitive Block Encipher.

4.43 multosEncipherCTR

void **multosEncipherCTR** (**const** BYTE algo, BYTE counterWidth, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen, BYTE *ivAddr, BYTE ivLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- BYTE counterWidth: the width of the counter used. Can be up to *ivLen*. (input)
- BYTE *inputAddr: pointer to plain text to encipher (input)
- BYTE *outputAddr: pointer to memory in which to write the cipher text output. (output)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)
- WORD inputLength: Length of plain text (input)
- BYTE *ivAddr: Pointer to ICV value (input)
- BYTE ivLen: length of the Initial Chaining Vector pointed to by ivAddr (input)

This function enciphers the plain text using the CTR mode as specified in ISO/IEC-10116 and supports a number of algorithms.

This is an interface to the primitive Block Encipher.

4.44 multosEncipherECB

void **multosEncipherECB** (**const** BYTE algo, BYTE *inputAddr, BYTE *outputAddr, BYTE keyLen, BYTE *keyAddr, WORD inputLen);

The parameters are:

- const BYTE algo: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES (input)
- WORD inputLen: Length of plainText (input)
- BYTE *inputAddr: pointer to plain text to encipher (input)
- BYTE *outputAddr: pointer to memory in which to write the cipher text output. (output)
- BYTE keyLen: length of key, depends on algorithm being used (input)
- BYTE *keyAddr: pointer to the key to use (input)

This function enciphers the plain text using the electronic code book method and supports a number of algorithms.

This is an interface to the primitive Block Encipher.

4.45 multosExchangeData

void **multosExchangeData** (BYTE *dataAddr, BYTE channel);

The parameters are:

- BYTE *channel*: identifier of the channel to which the data should be sent
- BYTE **dataAddr*: address of the data to send

This function exchanges data through the channel specified.

This is an interface to the primitive Exchange Data.

4.46 multosExit

void **multosExit** (void);

This function exits the application.

This is an interface to the instruction SYSTEM.

4.47 multosExitLa

void **multosExitLa** (const BYTE la)

The parameter is a single byte value indicating the actual length of response data.

This function exits the application setting La to the value given as *la*.

This is an interface to the instruction SYSTEM.

4.48 multosExitSW

void **multosExitSW** (const WORD sw)

The parameter is a word value indicating the value of the status word.

This function exits application with status word of *sw*.

This is an interface to the instruction SYSTEM.

4.49 multosExitSWLa

void **multosExitSWLa** (const WORD sw, const BYTE la)

The paratmeters are:

- const WORD *sw*: a word value indicating the value of the status word
- const BYTE *la*: a single byte value indicating the actual length of response data.

This function exits application with an SW of *sw* and an La of *la*.

This is an interface to the instruction SYSTEM.

4.50 multosExitToMultosAndRestart

void **multosExitToMultosAndRestart** (void);

There are no parameters for this function. This is an interface to the primitive Exit To MULTOS And Restart.

4.51 multosFill

void **multosFill** (BYTE *address, WORD length, BYTE value);

The parameters are:

- BYTE *address: starting address of memory to fill (input)
- BYTE value: the value to fill with (input)
- WORD length: the number of bytes to fill (input)

This fills a block of memory with the given value.

This is an interface to the primitive Memory Fill.

4.52 multosFillAdditionalStatic

void **multosFillAdditionalStatic** (DWORD staticOffset, DWORD length, BYTE value);

The parameters are:

- DWORD staticOffset: starting offset in “additional” static to fill (input)

- BYTE value: the value to fill “additional” static with (input)
- DWORD length: the number of bytes to fill (input)

This fills a block of memory in Additional Static with the given value. This can also be performed as an atomic operation using the function **multosFillAdditionalStaticAtomic**.

This is an interface to the primitive Memory Fill Additional Static.

4.53 multosFlushPublic

BOOL **multosFlushPublic** (WORD blockSize);

The parameters are:

- WORD *blockSize*: the number of bytes in the block of data to send to flush from public. (input)

Returns TRUE if La is set to a value > 0.

This is an interface to the primitive Flush Public.

4.54 multosGenerateAESCBCMAC

void **multosGenerateAESCBCMAC** (BYTE keyLength, BYTE *inputAddr, BYTE *MACAddr, BYTE *keyAddr, BYTE *IVAddr, WORD length, BYTE paddingByte);

The parameters are:

- BYTE keyLength: 16, 24 or 32.
- BYTE *inputAddr: address of the data to be MAC’d
- BYTE *MACAddr: address of the buffer to hold the resulting 16 byte MAC
- BYTE *keyAddr: address of the key to use to generate the MAC
- BYTE *IVAddr: address of the initialisation vector for the MAC algorithm
- WORD length: length of the input data in bytes
- BYTE paddingByte: the value of the padding byte to use

This performs an AES CBC MAC according to ISO9797-1 algorithm 1. If the input data is a multiple of the AES block length (16-bytes) then no padding is applied. This can be used to provide an IV to multosGenerateAESCMAC().

This is an interface to the primitive Generate MAC.

4.55 multosGenerateAESCMAC

void **multosGenerateAESCMAC** (BYTE keyLength, BYTE *inputAddr, BYTE *MACAddr, BYTE *keyAddr, BYTE *IVAddr, WORD length, BYTE paddingByte);

The parameters are:

- BYTE keyLength: 16, 24 or 32.
- BYTE *inputAddr: address of the data to be MAC'd
- BYTE *MACAddr: address of the buffer to hold the resulting 16 byte MAC
- BYTE *keyAddr: address of the key to use to generate the MAC
- BYTE *IVAddr: address of the initialisation vector for the MAC algorithm
- WORD length: length of the input data in bytes
- BYTE paddingByte: the value of the padding byte to use

This performs an AES CMAC according to ISO9797-1 MAC Algorithm 5 and in NIST SP 800-38B.

This is an interface to the primitive Generate MAC.

4.56 multosGenerateDESCBCSignature

void **multosGenerateDESCBCSignature** (BYTE *inputAddr, BYTE *sigAddr, BYTE *keyAddr, BYTE *ivAddr, WORD length);

The parameters are:

- WORD *length*: the length of the input data (input)
- BYTE **inputAddr*: the address of the input data (input)
- BYTE **ivAddr*: initial value to use in DES CBC algorithm [8 bytes] (input)
- BYTE **keyAddr*: DES key to use [8 bytes] (input)
- BYTE **signAddr*: array to hold 8-byte signature (output)

This function generates an eight byte DES CBC Signature over the input data.

This is an interface to the primitive Generate DES CBC Signature.

4.57 multosGenerateDESCBCMAC

void **multosGenerateDESCBCMAC** (BYTE keyLength, BYTE *inputAddr, BYTE *MACAddr, BYTE *keyAddr, BYTE *IVAddr, WORD length, BYTE paddingByte);

The parameters are:

- BYTE keyLength: 8, 16 or 24.
- BYTE *inputAddr: address of the data to be MAC'd
- BYTE *MACAddr: address of the buffer to hold the resulting 8 byte MAC
- BYTE *keyAddr: address of the key to use to generate the MAC
- BYTE *IVAddr: address of the initialisation vector for the MAC algorithm
- WORD length: length of the input data in bytes
- BYTE paddingByte: the value of the padding byte to use

This performs a DES CBC MAC according to ISO9797-1 algorithm 1.

This is an interface to the primitive Generate MAC.

4.58 multosGenerateDESMAC

void **multosGenerateDESMAC** (BYTE *inputAddr, BYTE *MACAddr, BYTE *keyAddr, BYTE *IVAddr, WORD length, BYTE paddingByte);

The parameters are:

- BYTE *inputAddr: address of the data to be MAC'd
- BYTE *MACAddr: address of the buffer to hold the resulting 8 byte MAC
- BYTE *keyAddr: address of the key (length 16 bytes) to use to generate the MAC
- BYTE *IVAddr: address of the initialisation vector for the MAC algorithm
- WORD length: length of the input data in bytes
- BYTE paddingByte: the value of the padding byte to use

This performs a DES MAC according to EMV 2000, Version 4.0 : December 2000. Integrated Circuit Card Specification for Payment Systems Book 2 – Security and Key Management, appendix A1.2 (see also ISO9797-1 algorithm 3).

This is an interface to the primitive Generate MAC.

4.59 multosGenerateHMAC

void **multosGenerateHMAC** (BYTE keyLength, BYTE *inputAddr, BYTE *MACAddr, BYTE *keyAddr, WORD length, BYTE hashAlgo);

The parameters are:

- BYTE keyLength: length can be up to the hash block length, padded with zeros.
- BYTE *inputAddr: address of the data to be MAC'd
- BYTE *MACAddr: address of the buffer to hold the resulting MAC of length *hash length*.
- BYTE *keyAddr: address of the key to use to generate the MAC
- WORD length: length of the input data in bytes
- BYTE hashAlgo: 0 = SHA-1 (block length = 64, hash length = 20),
  1 = SHA-256 (block length = 64, hash length = 32),
  2 = SHA-512 (block length = 128, hash length = 64)

This performs a HASH MAC according to ISO9797-2 MAC algorithm 2.

This is an interface to the primitive Generate MAC.

4.60 multosGenerateMAC

void **multosGenerateMAC** (const BYTE algorithm, BYTE *inputAddr, BYTE *MACAddr, BYTE *keyAddr, BYTE *IVAddr, WORD length, BYTE paddingByte);

The parameters are:

- const BYTE algorithm: Specifies the MAC algorithm to use. For supported algorithms see [MDRM].
- BYTE *inputAddr: address of the data to be MAC'd
- BYTE *MACAddr: address of the buffer to hold the resulting MAC
- BYTE *keyAddr: address of the key to use to generate the MAC
- BYTE *IVAddr: address of the initialisation vector for the MAC algorithm
- WORD length: length of the input data in bytes
- BYTE paddingByte: the value of the padding byte to use

This is an interface to the primitive Generate MAC.
Note: because of the varying inputs and outputs of different MAC algorithms, this function is now deprecated in favour of the individual functions for each MAC type.

4.61 multosGenerateRsaKeyPair

BOOL **multosGenerateRsaKeyPair** (const BYTE method, const BYTE mode, WORD mLen, BYTE *mAddr, BYTE *dpdqpquAddr, BYTE *eAddr, WORD eLen, WORD keyLen);

The parameters are:

- const BYTE method: Generation method, 0x00 = default, 0x01 = X9.31 (input)
- const BYTE mode: Default gen mode, 0x00 = performance, 0x01 = balanced, 0x02 = confidence (input)
- WORD keyLen: required length, in bytes, of the key pair public modulus (input)
- WORD eLen: length, in bytes, of the given exponent (input)
- BYTE* eAddr: address of the exponent value to use (input)
- BYTE* dpdqpquAddr: address of buffer to hold generated CRT private key (output)
- BYTE* mAddr: address of buffer to hold the generated public modulus (output)
- WORD mLen: equal to keyLen or 0 if the modulus is not to be returned (input)

Returns TRUE if the command is successful.

This is an interface to the primitive Generate RSA Key Pair.

4.62 multosGenerateTripleDESCBCSignature

void **multosGenerateTripleDESCBCSignature** (BYTE *inputAddr, BYTE *sigAddr, BYTE *keyAddr, BYTE *ivAddr, WORD length);

The parameters are:

- WORD *length*: the length of the input data (input)
- BYTE **inputAddr*: the address of the input data (input)
- BYTE **ivAddr*: initial 8 byte value to use in Triple DES CBC algorithm (input)
- BYTE **keyAddr*: 16 byte DES key to use (input)
- BYTE **sigAddr*: array to hold 8-byte signature (output)

This function generates an eight byte Triple DES CBC Signature.

This is an interface to the primitive Generate Triple DES CBC Signature.

4.63 multosGetAID

BYTE **multosGetAID** (BYTE appNumber, BYTE *dest);

The parameters are:

- BYTE *appNumber*: The number of the application to get the AID of.
- BYTE **dest*: The destination address for the 17-byte AID (one byte length followed by a 16-byte body)

This function provides the AID of the application with the given number (a 1 based index of the loaded applications in the order they are loaded). An application number of zero refers to the executing application.

The function return 0 indicates when an application with the specified application number does not exist and 1 indicates that the application does exist. The AID is only saved in the destination if the application exists.

This is an interface to the Primitive Get AID.

4.64 multosGetConfigData

WORD **multosGetConfigData (WORD token, BYTE* outAddr);**

The parameters are:

- WORD token: A two byte value containing the P1 P2 values defined for the Get Configuration Data APDU command (see [MDRM]).
- BYTE* outAddr: Points to a buffer to contain the requested configuration data.

Example:

```
dataLen = multosGetConfigData(0x0200,outAddr);
```

Returns the number of bytes written to *outAddr* or zero if an error condition occurred.

This is an interface to the primitive Get Configuration Data.

4.65 multosGetData

BYTE **multosGetData** (const BYTE readLength, BYTE *outAddr);

The parameters are:

- const BYTE readLength: the maximum number of bytes to read – should correspond with the size of the output buffer (input)
- BYTE *outAddr: pointer to a buffer to hold the output of the function. (output)

Returns the actual number of bytes written to the output buffer.

This is an interface to the primitive Get Data.

4.66 multosGetDelegatorAID

BOOL **multosGetDelegatorAID** (const BYTE readBytes, BYTE *aidAddr);

The parameters are:

- const BYTE readBytes: length of the AID to fetch (input)
- BYTE *aidAddr: address where delegator AID is written (output)

This function stores the AID of the delegating application into *aidAddr*. It returns TRUE if the application has not been delegated to.

This is an interface to the primitive Get Delegator AID.

4.67 multosGetDIRFile

BYTE **multosGetDIRFile** (const BYTE readLength, BYTE recNo, BYTE *addr);

The parameters are:

- const BYTE *readLength*: the number of bytes of the DIR record to copy (input)
- BYTE *recNo*: DIR record number (input)
- BYTE **addr*: address where result is written (output)

This function retrieves the record *recNo* from the DIR File and copies it to *addr*. It returns the actual number of bytes copied.

Note: *recNo* is a 1 based index. A value of 0 indicates the currently selected application.

This is an interface to the primitive Get DIR File Record.

4.68 multosGetFCI

BYTE **multosGetFCI** (const BYTE readLength, BYTE recNo, BYTE *addr);

The parameters are:

- const BYTE *readLength*: the number of bytes of the FCI record to copy (input)
- BYTE *recNo*: FCI record number (input)
- BYTE **addr*: address where result is written (output)

This function retrieves the File Control Information for the application with index *recNo* and copies it to *addr*. It returns the actual number of bytes copied.

Note: *recNo* is a 1 based index. A value of 0 indicates the currently selected application.

This is an interface to the primitive Get File Control Information.

4.69 multosGetFCIState

BYTE **multosGetFCIState();**

This function returns 1 if the executing application has a dual FCI or 0 if it has a normal FCI.

This is an interface to the primitive Get FCI State.

4.70 multosGetManufacturerData

BYTE **multosGetManufacturerData** (**const** BYTE readLength, BYTE *addr);

The parameters are:

- const BYTE *readLength*: the number of bytes to be copied (input)
- BYTE **addr*: address where result is written (output)

This function retrieves the Manufacturer Data from MULTOS chip and writes the result to *addr*. It also returns the actual number of bytes copied.

This is an interface to the primitive Get Manufacturer Data.

4.71 multosGetMemoryReliability

BYTE **multosGetMemoryReliability** (void);

This function returns the status of the current reliability of the non-volatile memory as follows:

- MULTOS_MEMORY_RELIABLE
- MULTOS_MEMORY_MARGINAL
- MULTOS_MEMORY_UNRELIABLE

This is an interface to the primitive Get Memory Reliability.

4.72 multosGetMultosData

BYTE **multosGetMultosData** (const BYTE readLength, BYTE *addr);

The parameters are:

- const BYTE *readLength*: the number of bytes to be copied (input)
- BYTE **addr*: address where result is written (output)

This function retrieves the MULTOS Data from MULTOS chip and writes it to *output*. It also returns the actual number of bytes copied.

This is an interface to the primitive Get MULTOS Data.

4.73 multosGetPINStatus

BYTE **multosGetPINStatus** (void);

Returns the PIN status value as defined in the [MDRM].

This is an interface to the primitive Get PIN Data.

4.74 multosGetPINTryCounter

BYTE **multosGetPINTryCounter** (void);

The parameter returns the current PIN Try Counter value.

This is an interface to the primitive Get PIN Data.

4.75 multosGetPINTryLimit

BYTE **multosGetPINTryLimit** (void);

The function returns the current PIN Try Limit value.

4.76 multosGetPINVerificationStatus

BYTE **multosGetPINVerificationStatus** (void);

The function returns 0xA5 if the PIN is verified and 0x5A if unverified.

This is an interface to the primitive Get PIN Data.

4.77 multosGetProcessEvent

BYTE **multosGetProcessEvent** (void);

The parameter returns the ID of the event that caused the application to be run. Current events defined in multos.h are:-

- EVENT_APP_APDU 0
- EVENT_SELECT_APDU 1
- EVENT_AUTO_SELECT 2
- EVENT_RESELECT_APDU 3
- EVENT_DESELECT 4
- EVENT_CREATE 5
- EVENT_DELETE 6

This is an interface to the primitive Get Process Event.

4.78 multosGetRandomNumber

QWORD **multosGetRandomNumber** (void);

This function generates an 8-byte random number.

This is an interface to the primitive Get Random Number.

4.79 multosGetReplacedAppState

BYTE **multosGetReplacedAppState** (void);

This function returns

- 0 = No replaced application exists
- 1 = Replaced application exists but is not readable (bit 13 of its access_list is not set)

- 2 = Replaced application exists and is readable (bit 13 of its access_list is set)

This is an interface to the primitive Get Replaced Application State.

4.80 multosGetSessionSize

WORD **multosGetSessionSize** (void);

Returns the current size of the executing application's Session data. It is an interface to the primitive Get Session Size.

4.81 multosGetStaticSize

BOOL **multosGetStaticSize** (DWORD* value);

The parameters are:

- DWORD *value: amount of static memory allocated (output)

Returns TRUE if the command is successful.

This calls the primitive Get Static Size to return the amount of static memory allocated by the OPEN MEL APPLICATION command which includes "normal" static memory included in the ALU and "additional" static memory that is accessed using the Additional Static primitives.

4.82 multosGetStaticSizeHuge

BOOL **multosGetStaticSizeHuge** (QWORD *value);

The parameters are:

- QWORD *value: address of 8 byte value to hold the amount of static memory allocated (output)

Returns TRUE if the command is successful.

This calls the primitive Get Static Size to return the amount of static memory allocated by the OPEN MEL APPLICATION command which includes "normal" static memory included in the ALU and "additional" static memory that is accessed using the Additional Static primitives.

4.83 multosGetSupportedInterfaces

WORD **multosGetSupportedInterfaces** (const BYTE type);

The parameters are:

- BYTE type: 0 = standard interfaces, 1 = customer interfaces

Calls the primitive Get Available Interface Types and returns a 16 bit field as described in [MDRM].

4.84 multosGetTransactionState

BYTE **multosGetTransactionState** (void);

The function returns 1 if transaction protection is on and 0 if it is off.

This is an interface to the primitive Get Transaction State.

4.85 multosGsmAuthenticate

void **multosGsmAuthenticate** (BYTE *sreskcAddr, BYTE *keyAddr, BYTE *randAddr);

The parameters are:

- BYTE *randAddr: address of the 16 byte random challenge to use (input)
- BYTE *keyAddr: address of the 16 byte key to use (input)
- BYTE *sreskcAddr: address of buffer to store the 4 byte SRES and 8 byte Kc values (output)

This is an interface to the primitive GSM Authenticate.

4.86 multosIncrement

void **multosIncrement** (const BYTE blockLength, const BYTE *block)

The parameters are:

- const BYTE *blockLength*: size of the block on which to perform the operation
- const BYTE **block*: address of the block on which to perform the operation

This function increments the value held in *block* by one.

This is an interface to the instruction INCN

4.87 multosIndex

void **multosIndex** (BYTE blockLength, BYTE index, BYTE *baseAddr, BYTE *resultData)

The parameters are:

- BYTE blockLength: the length of records in the file (input)
- BYTE index: the index of the record to be retrieved (input)

- BYTE *baseAddr: the address of the first byte of the file (input)
- BYTE *resultData: a pointer to the memory to hold blockLength number of bytes output.

This is an interface to the instruction Index.

4.88 multosInitialisePIN

void **multosInitialisePIN** (BYTE *initDataAddr);

The parameter is a pointer to a data block containing the PIN initialisation data formatted as follows:

- PIN Reference Data (8 bytes)
- PIN Length (1 byte)
- PIN Try Counter (1 byte)
- PIN Try Limit (1 byte)
- Checksum (4 bytes)

This is an interface to the primitive Initialise PIN.

4.89 multosInitialisePINExtended

void **multosInitialisePINExtended** (BYTE *initDataAddr);

The parameter is a pointer to a data block containing the PIN initialisation data formatted as follows:

- PIN Length (1 byte)
- PIN Reference Data (PIN Length bytes)
- PIN Try Counter (1 byte)
- PIN Try Limit (1 byte)
- Checksum (4 bytes)

This is an interface to the primitive Initialise PIN Extended.

4.90 multosInvert

```
void multosInvert (const BYTE blockLength, const BYTE *block)
```

The parameters are:

- const BYTE *blockLength*: size of the block on which to perform the operation
- const BYTE **block*: address of the block on which to perform the operation

This function logically inverts the value held in *block*.

This is an interface to the instruction NOTN

4.91 multosLoadMaskedKey

WORD **multosLoadMaskedKey** (BYTE *key, BYTE keyType);

The parameters are:

- BYTE **key*: a pointer to the masked key to load
- BYTE *keyType*: the crypto algorithm the key will be used with. See valid values below.

This function tells the operating system to load and internally unmask the *key* ready for use as a key of *keyType*. It returns the checksum of the clear key allowing for a comparison with the checksum returned when the clear key was originally masked by the function **multosMaskData()**.

Valid key types are:

- MASK_DES_KEY
- MASK_3DES_2KEY
- MASK_3DES_3KEY
- MASK_SEED_KEY
- MASK_AES_16KEY
- MASK_AES_24KEY
- MASK_AES_32KEY

This function is an interface to an extension primitive.

4.92 multosLookup

BOOL **multosLookup** (BYTE *arrayAddr, BYTE value, BYTE *resultOffset);

The parameters are:

- BYTE *value*: the value to locate
- BYTE **arrayAddr*: address of the array to be searched.
- BYTE **resultOffset*: address of the byte to which the result will be written

This function locates the first occurrence of *value* within the search array. Note that this function treats the first byte of the array as indicating the total number of bytes in the array. The function returns TRUE if *value* is found and *resultOffset* is the offset within the array where *value* is first found.

This is an interface to the primitive Lookup.

4.93 multosLookupWord

BYTE **multosLookupWord** (WORD *arrayAddr, WORD value, WORD *resultOffset);
The parameters are:

- WORD *value*: the value to locate
- WORD **block*: address of the word array to be searched.

- WORD **result*: address to which the result will be written
- BOOL **wordFound*: true if the value is found

This function locates the first occurrence of *value* within the array. Note that this function treats the first word of the array as indicating the total number of words in the array. The function returns the following values:

9: full match,
8: MSB of value matches,
1: LSB of value matches.

resultOffset is the offset within the array where *value* is first found.

This is an interface to the primitive Lookup Word.

4.94 multosMaskData

WORD **multosMaskData** (BYTE *data, BYTE *dest, BYTE length);

The parameters are:

- BYTE **data*: pointer to the data to mask.
- BYTE **dest*: location to store the masked data (can be the same as *data*).
- BYTE *length*: the length of *data* (maximum is 32 bytes)

This function is used to mask the value of a key by the operating system so that its clear value cannot be accidentally or maliciously leaked. The mask is random and different for each application loaded to the card. The function returns a checksum which can be used to verify the integrity of the masked key (see **multosVerifyMaskedKey**()).

If the key value forms part of the personalised application data it is recommended that it is masked during the processing of the Application Create event.

In order to use a masked key to encrypt or decrypt, the **multosLoadMaskedKey**() function must first be used.

This is an interface to an extension primitive.

4.95 multosModularExponentiation

void **multosModularExponentiation** (BYTE *outAddr, BYTE *inAddr, BYTE *mAddr, BYTE *eAddr, WORD mLen, WORD eLen);

The parameters are:

- WORD *eLen*: the length of the exponent used (input)
- WORD *mLen*: the length of the modulus (input)
- BYTE **eAddr*: address of the exponent (input)

- BYTE **mAddr*: address of the modulus (input)
- BYTE **inAddr*: address of the input value (input)
- BYTE **outAddr*: address of where to write the result of the operation (output)

This function performs a modular exponentiation. Note that the values held at *mAddr*, *inAddr* and *outAddr* are all considered to be of size *mLen*.

This is an interface to the primitive Modular Exponentiation.

4.96 multosModularExponentiationCRT

void **multosModularExponentiationCRT** (BYTE *outAddr, BYTE *inAddr, BYTE *pquAddr, BYTE *dpdqAddr, WORD dpdqLen);

The parameters are:

- WORD *dpdqLen*: length of dpdq (input)
- BYTE **dpdqAddr*: address of dpdq (input)
- BYTE **pquAddr*: address of pqu (input)
- BYTE **inAddr*: address of input (input)
- BYTE **outAddr*: address of output (output)

This function performs a modular exponentiation using Chinese Remainder Theorem.

This is an interface to the primitive Modular Exponentiation CRT.

Note: A version of this function, **multosModularExponentiationCRTProtected**, exists with the same prototype that interfaces to the primitive Modular Exponentiation CRT Protected instead.

4.97 multosMultiply

void **multosMultiply** (const BYTE blockLength, BYTE *block1, BYTE *block2, BYTE *result)

The parameters are:

- const BYTE *blockLength*: the size of the operands
- BYTE **block1*: address of the first byte of block1
- BYTE **block2*: address of the first byte of block2
- BYTE **result*: address of the first byte of result

This function multiplies the value held in *block1* by that held in *block2* and writes the result to the block result of size *blockLength + blockLength.*

This is an interface to the primitive MultiplyN.

4.98 multosOr

void **multosOr** (const BYTE blockLength, BYTE *block1, BYTE *block2, BYTE *result)

The parameters are:

- const BYTE *blockLength*: the size of all operand and result blocks
- BYTE **block1*: address of block1
- BYTE **block2*: address of block2
- BYTE **result*: address of the first byte of block where the result is to be written.

This function performs a logical OR operation using the values in *block1* and *block2* as operands. The output of the operation is written to *result*.

This is an interface to the instruction ORN.

4.99 multosPad

WORD **multosPad** (const BYTE scheme, BYTE *addrMsg, WORD lenMsg, BYTE blockLen);

The parameters are:

- WORD lenMsg: the length of data pointed to by msg (input)
- BYTE *addrMsg: address of the data to be padded (input/output)
- BYTE blockLen: size in bytes of the block length to pad to (input)
- BYTE scheme: (const input)
  - 0x01 = 0x80 followed by zero or more 0x00
  - 0x02 = 0x80 followed by one or more 0x00

This uses the primitive Pad to pad a message to a given block length using one of the two methods specified. Returns the length of the padded message.

4.100 multosPlatformOptimisedChecksum

DWORD **multosPlatformOptimisedChecksum** (BYTE *blockAddr, WORD length);

The parameters are:

- WORD length: length of the data to calculate the checksum over
- BYTE *blockAddr: pointer to the data to calculate the checksum over

This function calls the primitive Platform Optimised Checksum to return an implementation specific 4 byte checksum.

4.101 multosRestoreStack

void **multosRestoreStack** (void);

This function restores the application's stack to the copy previously saved with *multosSaveStack()*. The intention of the function is to allow an application to resume execution at the point just after it called *multosExitToMultosAndRestart()*. It may only be used is specific circumstances. See [MDRM] for details.

This is an interface to the primitive Manage Stack.

4.102 multosQueryChannel

BOOL **multosQueryChannel** (BYTE channelID);

This function verifies the existence of a specific channel with ID *channelID* and returns TRUE if the channel is supported.

This is an interface to the primitive Query Channel.

4.103 multosQueryCodelet

BOOL **multosQueryCodelet** (WORD codeletID);

This function verifies the existence of a specific codelet with ID *codeletID* and returns TRUE if it is supported.

This is an interface to the primitive Query Codelet.

4.104 multosQueryCryptographicAlgorithm

BOOL **multosQueryCryptographicAlgorithm** (BYTE algorithmID);

Returns TRUE if the algorithm is supported by the device.

This is an interface to the primitive Query Cryptographic Algorithm. See [MDRM] for the valid values of *algorithmId*.

4.105 multosQueryInterfaceType

BOOL **multosQueryInterfaceType** (void);

This function calls primitive functions to determine if the interface is contactless.

4.106 multosQueryPrimitive

void **multosQueryPrimitive** (const BYTE setNum, const BYTE primitiveNum, BOOL *primitiveSupported)

The parameters are:

- const BYTE *setNum*: the set to which the primitive belongs
- const BYTE *primitiveNum*: the number of the primitive within its set
- BOOL **primitiveSupported*: Boolean flag

This function verifies the existence of a specific primitive with number *primitiveNum* within set *setNum*. The flag *primitiveSupported* is set to TRUE if the primitive is supported, otherwise it is set to FALSE.

This is an interface to the primitive Query Primitive.

4.107 multosReadPIN

BYTE **multosReadPIN** (BYTE *outAddr);

The parameters points to a buffer to hold the returned PIN. The function returns the length of the PIN.

This is an interface to the primitive Read PIN.

4.108 multosRejectProcessEvent

void **multosRejectProcessEvent** (void);

This is an interface to the primitive Reject Process Event. See [MDRM] for details of Process Events.

4.109 multosResetSessionData

void **multosResetSessionData** (void);

This function allows a shell application to reset the session data of all other applications on the MULTOS card.

This is an interface to the primitive Reset Session Data.

4.110 multosResetWWT

void **multosResetWWT** (void);

This function sends a WWT extension request.

This is an interface to the primitive Reset WWT.

4.111 multosReturnFromCodelet

void **multosReturnFromCodelet** (const BYTE numBytesIn, const BYTE numBytesOut);
The parameters are:

- const BYTE *numBytesIn*: the number of stack bytes that were passed to the codelet
- const BYTE *numBytesOut*: the number of stack bytes returned by the codelet.

This function returns from the currently executing codelet and ensures that *numBytesIn* are removed from the stack and *numBytesOut* replace them.

This is an interface to the primitive Return From Codelet.

4.112 multosRotateLeft

void **multosRotateLeft** (BYTE *dataAddr, WORD dataLen, WORD numBits);

The parameters are:

- WORD *numBits:* the number of bits to rotate by
- WORD *dataLen*: the length of the data (in bytes) pointed to by *data_addr*
- BYTE* *dataAddr*: the address of the data to be rotated

This function performs bit-wise rotation of the data pointed to *dataAddr* rotating the bits from most significant (leftmost) to least significant (rightmost).

This is an interface to the primitive Shift Rotate.

4.113 multosRotateRight

void **multosRotateRight** (BYTE *dataAddr, WORD dataLen, WORD numBits);

The parameters are:

- WORD *numBits:* the number of bits to rotate by
- WORD *dataLen*: the length of the data (in bytes) pointed to by *data_addr*
- BYTE* *dataAddr*: the address of the data to be rotated

This function performs bit-wise rotation of the data pointed to *dataAddr* rotating the bits from least significant (rightmost) to most significant (leftmost).

This is an interface to the primitive Shift Rotate.

4.114 multosSaveStack

void **multosSaveStack** (void);

This function saves the application's stack to be restored with multosRestoreStack(). The intention of the function is to allow an application to save its execution state at the point just before calling *multosExitToMultosAndRestart()*. It may only be used is specific circumstances. See [MDRM] for details.

This is an interface to the primitive Manage Stack.

4.115 multosRsaVerify

void **multosRsaVerify** (BYTE *outAddr, BYTE *inAddr, BYTE *mAddr, BYTE *eAddr, WORD mLen, WORD eLen);

The parameters are:

- WORD *eLen*: length of the exponent value pointed to by *eAddr*. (input)
- WORD *mLen*: length of the public modulus pointed to by *mAddr.* (input)
- BYTE **eAddr*: buffer holding the exponent value (input)
- BYTE **mAddr*: buffer holding the public modulus (input)
- BYTE **inAddr*: buffer holding the input to the modular exponentiation operation (input)
- BYTE **outAddr*: buffer for the result of the modulus exponentiation operation (output)

This primitive performs modular exponentiation operation and the result is written at the specified address outAddr.

This is an interface to the primitive RSA Verify.

4.116 multosSecureHash

void **multosSecureHash** (BYTE *addrMsg, BYTE *addrHash, WORD lenHash, WORD lenMsg);

The parameters are:

- WORD lenMsg: length of the data pointed to by msgIn (input)
- WORD lenHash: length of the required hash value (input)
- BYTE *addrHash: address of buffer to hold hash result (output)
- BYTE *addrMsg: address of the message to hash (input)

This is an interface to the primitive Secure Hash and supports SHA-1 and SHA-2 digests of various lengths as documented in the [MDRM].

4.117 multosSecureHashIV

void **multosSecureHashIV** (WORD msgLen, WORD hashLen, BYTE *hashOut, BYTE *msgIn, BYTE *intermediateHash, DWORD *numPrevHashedBytes, WORD *numMsgRemainder, WORD *msgRemainder);

The parameters are:

- WORD msgLen: The size, in bytes, of msgIn (input)
- WORD hashLen: The length of the required hash, in bytes (input)
- BYTE *hashOut: Address of buffer to hold the resulting hash (output)
- BYTE *msgIn: Address of the message to hash (input)
- BYTE *intermediateHash: Address of initialisation vector to input to the hash (input)
- DWORD *numPrevHashedBytes: Address of count of number of bytes previously input to the hashing algorithm (input/output)
- WORD *numMsgRemainder: Number of non block aligned bytes (input/output)
- WORD *msgRemainder: Non block-aligned bytes (input/output)

This is an interface to the primitive Secure Hash IV. The following is a code fragment that shows how this primitive can be used to hash long stream of data passed via multiple APDU calls.

```
#pragma melsession
BYTE bRemain[64];
WORD wLenMsgRem;

void
main(void)
{
  // …
  case CMD_HASHINIT:
     pRemainder = bRemain;
     wLenMsgRem = 0;
     // etc.

  case CMD_HASHIV:
    // On entry, pRemainder points to the buffer storing the remainder from the previous call
    multosSecureHashIV(Lc, 32, bHash2, pub, bIMHash, &dwPrevHashedBytes, &wLenMsgRem,
&pRemainder);

    // On exit, pRemainder points to the data in public that was not hashed.
    // That data needs to be saved for the next calculation
    memcpy(bRemain,pRemainder,wLenMsgRem);
    //etc
}
```

4.118 multosSetATRFileRecord

BYTE **multosSetATRFileRecord** (BYTE *atrAddr);

This function writes a record into the ATR File and returns number of bytes written. The first byte of the data pointed to by *atrAddr* contains the length of the ATR that follows.

This is an interface to the primitive Set ATR File Record.

4.119 multosSetATRHistoricalCharacters

BYTE **multosSetATRHistoricalCharacters** (BYTE *histAddr);

This function writes data to the historical characters of the card's ATR and returns the number of bytes written. The first byte of *histAddr* is the length of the data that follows.

This is an interface to the primitive Set ATR Historical Characters.

4.120 multosSetATSHistoricalCharacters

BYTE **multosSetATSHistoricalCharacters** (BYTE *histAddr);

This function writes data to the historical characters of the card's ATS and returns the number of bytes written. The first byte of *histAddr* is the length of the data that follows.

This is an interface to the primitive Set ATS Historical Characters.

4.121 multosSetFCIFileRecord

BYTE **multosSetFCIFileRecord** (BYTE *fciAddr);

The first byte of the data pointed to by *fciAddr* must indicate the length of the remaining data. The function returns the number of bytes written.

This is an interface to the primitive Set FCI File Record.

4.122 multosSetPINTryCounter

void **multosSetPINTryCounter** (BYTE value);

Sets the current PIN Try Counter to the value given.

This is an interface to the primitive Set PIN Data.

4.123 multosSetPINTryLimit

void **multosSetPINTryLimit** (BYTE value);

Sets the current PIN Try Limit to the value given.

This is an interface to the primitive Set PIN Data.

4.124 multosSetPINVerificationStatus

void **multosSetPINVerificationStatus** (BYTE value);

Sets the current PIN Verification Status to the value given. The value must be 0x5A (unverified) or 0xA5 (verified).

This is an interface to the primitive Set PIN Data.

4.125 multosSetProtectedMemoryAccess

void **multosSetProtectedMemoryAccess** (const BYTE options);

options is either 0x00 for off, or 0x01 for on. This is an interface to the Set Protected Memory Access primitive.

4.126 multosSetSelectCLSW

void **multosSetSelectCLSW** (const BYTE sw1, const BYTE sw2);

The parameters are:

- const BYTE *sw1*: the value to be written to the most significant byte of the status word
- const BYTE *sw2*: the value to be written to the least significant byte of the status word

This function sets the 2-byte status word that will be returned by MULTOS when the application is next selected over the contactless interface.

This is an interface to the primitive SetContactlessSelectSW.

4.127 multosSetSelectSW

void **multosSetSelectSW** (const BYTE sw1, const BYTE sw2);

The parameters are:

- const BYTE *sw1*: the value to be written to the most significant byte of the status word
- const BYTE *sw2*: the value to be written to the least significant byte of the status word

This function sets the 2-byte status word that will be returned by MULTOS when the application is next selected.

This is an interface to the primitive Set SelectSW.

4.128 multosSetSilentMode

void **multosSetSilentMode** (const BYTE mode);

This is an interface to the primitive Set Silent Mode. See [MDRM] for mode values.

4.129 multosSetTransactionProtection

void **multosSetTransactionProtection** (const BYTE options);

The parameter is a byte value specifying what option to use with transaction protection.
The only valid options are the following:

- MULTOS_TP_OFF_AND_DISCARD
- MULTOS_TP_OFF_AND_COMMIT
- MULTOS_TP_ON_AND_DISCARD
- MULTOS_TP_ON_AND_COMMIT

This is an interface to the primitive Set Transaction Protection.

4.130 multosSHA1

void **multosSHA1** (BYTE *addrMsg, BYTE *addrHash, WORD lenMsg);

The parameters are:

- WORD *lenMsg*: length of the message to submit to the SHA-1 algorithm (input)
- BYTE **addrMessage*: address of the message (input)

- BYTE *addrH*ash*: address where to write the 20-byte digest (output)

This function uses the value found in *addrMessage* of size *lenMsg* as input to the SHA-1 hashing algorithm. The resulting 20-byte digest is written to addrH*ash*.

This is an interface to the primitive SHA-1.

4.131 multosShiftLeft

void **multosShiftLeft** (BYTE *dataAddr, WORD dataLen, WORD numBits);

The parameters are:

- WORD *numBits:* the number of bits to shift by
- WORD *dataDen*: the length of the data (in bytes) pointed to by *data_addr*
- BYTE* *dataAddr*: the address of the data to be shifted

The function performs left shifts on the value found in *dataAddr*.

This is an interface to the primitive Shift Rotate.

4.132 multosShiftRight

void **multosShiftRight** (BYTE *dataAddr, WORD dataLen, WORD numBits);

The parameters are:

- WORD *numBits:* the number of bits to shift by
- WORD *dataLen*: the length of the data (in bytes) pointed to by *data_addr*
- BYTE* *dataAddr*: the address of the data to be shifted

The function performs right shifts on the value found in *dataAddr.*

This is an interface to the primitive Shift Rotate.

4.133 multosSubtract

void **multosSubtract** (const BYTE blockLength, BYTE *block1, BYTE *block2, const BYTE *result)

The parameters are:

- const BYTE *blockLength*: size of the blocks to subtract. Both blocks must be the same size.
- BYTE **block1*: address of the first block
- BYTE **block2*: address of the second block

- const BYTE *_result_: address of the block that will hold the result of the operation

This function subtracts the value found in *block1* to that found in *block2* and places the difference in the block indicated in *result*.

This is an interface to the instruction SUBN.

4.134 multosSubtractBCD

void multosSubtractBCD (const BYTE length, BYTE *operand1, BYTE *operand2, BYTE *result)

The parameters are:

- const BYTE length: the length of each BCD operand (input)
- BYTE *operand1: address of the first operand (input)
- BYTE *operand2: address of the second operand (input)
- BYTE *result: address to hold the result of the subtraction (output)

This function subtracts operand2 from operand1.

This is an interface to the primitive Subtract BCDN.

4.135 multosTestZero

void **multosTestZero** (const BYTE blockLength, const BYTE *block, BOOL *isZero)

The parameters are:

const BYTE *blockLength*: size of the block to test
const BYTE **block*: the address of the block to test
BOOL **isZero*: flag indicating if all bytes are zero

This function tests each byte in block has a value of zero. The flag *isZero* is set to TRUE if all bytes are zero, otherwise it is set to FALSE.

This is an interface to the instruction TESTN.

4.136 multosUnPad

WORD **multosUnPad** (**const** BYTE scheme, BYTE *addrMsg, WORD lenMsg);

The parameters are:

- WORD lenMsg: the length of data pointed to by addrMsg (input)

- BYTE *addrMsg: address of the data to have the padding removed (input/output)
- BYTE scheme: (const input)
    - 0x01 = 0x80 followed by zero or more 0x00
    - 0x02 = 0x80 followed by one or more 0x00

This uses the primitive Unpad to remove the padding from a message and returns the length of the unpadded message.

4.137 multosUpdateProcessEvents

void **multosUpdateProcessEvents** (WORD mask);

Mask is a 16 bit bitmap, bit15 being the leftmost, most significant bit.

- bit0 = APDU event mask.
- bit1 = SELECT event mask.
- bit2 = Automatic SELECT event mask.
- bit3 = RESELECT event mask.
- bit4 = DESELECT event mask.
- bit5 = CREATE event.
- bit6 = DELETE event.
- bit7 – bit15 = RFU (set to zero)

This primitive enables or disables individual events for an application according to the mask provided.

This is an interface to the primitive Update Process Events.

4.138 multosUpdateSessionSize

BYTE **multosUpdateSessionSize** (WORD sessionSize);

This primitive updates the total size of the application's session memory allowing you to free up space no longer required or allocate more space if needed (up to the maximum allowed in the ALC and available remaining RAM). Returns 1 if the operation succeeds or 0 if it fails.

This is an interface to the primitive Update Session Size.

4.139 multosUpdateStaticSize

BYTE **multosUpdateStaticSize** (DWORD staticSize);

This primitive updates the total size of the application's Static memory allowing you to free up space no longer required or allocate more space if needed (up to the maximum allowed in the ALC and available remaining space). Returns 1 if the operation succeeds or 0 if it fails.

This is an interface to the primitive Update Static Size.

4.140 multosVerifyMaskedKey

WORD multosVerifyMaskedKey (BYTE unloadKey);

The parameters are:

- BYTE unloadKey: MASK_VERIFY_KEY_AND_TERMINATE or MASK_VERIFY_KEY_ONLY

This primitive can be used to verify the integrity of the masked key currently loaded by the operating system and optionally unload it (such that it needs to be loaded again before the next use).

This function returns the checksum of the currently unmasked key for comparison with that calculated when the key was originally masked using **multosMaskData().**

This is an interface to an extension primitive.

4.141 multosVerifyPIN

WORD **multosVerifyPIN** (BYTE *pinAddr, BYTE pinLen);

The parameters are:

- BYTE pinLen: Length of the PIN held in the given buffer (input)
- BYTE *pinAddr: Pointer to the buffer holding the PIN (input)

This function returns the result of the verification process. 0x5AA5 for verified, 0xA55A for NOT verified.

This is an interface to the primitive Verify PIN.

4.142 multosXor

void **multosXor** (const BYTE blockLength, BYTE *block1, BYTE *block2, BYTE *reuslt)

The parameters are:

- const BYTE *blockLength*: the size of all operand and result blocks
- BYTE **block1*: address of block1
- BYTE **block2*: address of block2
- BYTE **result*: address of the first byte of block where the result is to be written.

This function performs a logical XOR operation using the values in *block1* and *block2* as operands. The output pf the operation is written to *result*.

This is an interface to the instruction XORN.

A. Appendix A : Biometric C API

Introduction

This appendix documents the MULTOS Biometric C API. This API has been chosen to be compatible with the Java Card biometric API and to simplify the porting of existing biometric Java Card applets to MULTOS.

Constants

	Description
BIO_VERSION_LENGTH	The length of the Biometric API version string.
BIO_MAX_PUBLIC_TEMPLATE_LENGTH	The maximum length of the public template.
BIO_MINIMUM_SUCCESSFUL_MATCH_SCORE	The minimum successful template match score.
BIO_MATCH_NEEDS_MORE_DATA	The match score that indicates that more data is required to complete the match process.

Data Types

```
enum BIO_TYPE
{ /* Facial feature recognition (visage). */
FACIAL_FEATURE,
/* Pattern is a voice sample (specific or unspecific speech).*/
VOICE_PRINT,
/* Fingerprint identification (any finger). */
FINGERPRINT,
/* Pattern is a scan of the eye's iris. */
IRIS_SCAN,
/* Pattern is an infrared scan of blood vessels of the retina of
the eye. */
RETINA_SCAN,
/* Hand geometry ID is based on overall geometry/shape of the
hand. */
HAND_GEOMETRY,
/* Written signature dynamics ID (behavioral). */
SIGNATURE,
/* Keystrokes dynamics (behavioral). */
KEYSTROKES,
/* Lip movement (behavioral). */
LIP_MOVEMENT,
/* Thermal face image. */
THERMAL_FACE,
/* Thermal hand image. */
THERMAL_HAND,
/* Gait (behavioral). */
GAIT_STYLE,
/* Body odor. */
BODY_ODOR,
/* Pattern is a DNA sample for matching. */
```

```
DNA_SCAN,
/* Ear geometry ID is based on overall geometry/shape a ear. */
EAR_GEOMETRY,
/* Finger geometry ID is based on overall geometry/shape of a
finger. */
FINGER_GEOMETRY,
/* Palm gemoetry ID is based on overall geometry/shape of palm.
*/
PALM_GEOMETRY,
/* Pattern is an infrared scan of the vein pattern in a face,
wrist or hand. */
VEIN_PATTERN,
/* General password (a PIN is a special case of the password).
*/
PASSWORD
};

typedef BYTE BIO_VERSION[BIO_VERSION_LENGTH];
An array that holds the Biometic API version string.

typedef BYTE
BIO_PUBLIC_TEMPLATE[BIO_MAX_PUBLIC_TEMPLATE_LENGTH];
An array that holds the public template.

struct BIO_TEMPLATE
{
  // Contents implementation-specific
};
A structure that contains the biometric template.
```

Data

The application must define one Static data structure of type `BIO_TEMPLATE` for each biometric template that it wishes to use.

Function Prototypes

A.1.1 bioInit

```
void bioInit (struct BIO_TEMPLATE *refTemplate, BYTE
*dataBuffer, WORD dataLength)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE *dataBuffer: pointer to buffer holding data to be enrolled
- WORD dataLength: length of data in data buffer

This function initialises the enrolment of a reference template.

A.1.2 bioUpdate

```
void bioUpdate (struct BIO_TEMPLATE *refTemplate, BYTE
*dataBuffer,     WORD dataLength)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE *dataBuffer: pointer to buffer holding data to be enrolled
- WORD dataLength: length of data in data buffer

This function continues the enrolment of a reference template. This function should only be used if all the data required for the initialisation is not available in one data buffer.

A.1.3 bioDoFinal

```
void bioDoFinal (struct BIO_TEMPLATE *refTemplate,
     BYTE newTryLimit)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE newTryLimit: the number of tries allowed before the reference is blocked

This function finalises the enrolment of a reference template. Final action of enrolment is to designate a reference template as being complete and ready for use (marks the reference as initialised, set the try limit and resets the try counter). This function may also include some error checking prior to the validation of reference template as ready for use.

A.1.4 bioResetUnblockAndSetTryLimit

```
void bioResetUnblockAndSetTryLimit (struct BIO_TEMPLATE
*refTemplate,       BYTE newTryLimit)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE newTryLimit: the number of tries allowed before the reference is blocked

This function resets the global validated flag, updates the try limit value and resets the try counter to the try limit value.

A.1.5 bioGetBioType

```
BYTE bioGetBioType (void)
```

This function returns the biometric type. Valid types are described in `BIO_TYPE`.

A.1.6 bioIsInitialized

```
BOOL bioIsInitialized (struct BIO_TEMPLATE *refTemplate)
```

The parameter is:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory

This function returns the initialisation status of the reference template. This is independent of whether or not the match process has been initialised (see `bioInitMatch`).

A.1.7 bioIsValidated

```
BOOL bioIsValidated (struct BIO_TEMPLATE *refTemplate)
```

The parameter is:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory

This function returns TRUE if the template has been successfully checked since the last card reset or last call to `bioReset()`.

A.1.8 bioGetVersion

```
BYTE bioGetVersion (BIO_VERSION bioVersion)
```

The parameter is:

- BIO_VERSION bioVersion: pointer to the array in which the version ID will be stored

This function gets the matching algorithm version or ID and returns the number of bytes written in the version data buffer.

A.1.9 bioGetPublicTemplateData

```
WORD bioGetPublicTemplateData (struct BIO_TEMPLATE *refTemplate,
BYTE *dataBuffer,  WORD dataLength)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE *dataBuffer: pointer to the destination area
- WORD dataLength: the number of bytes to copy

This function gets the public part of the reference template. It copies all or a piece of the reference public data to the destination area.

A.1.10 bioGetTriesRemaining

```
BYTE bioGetTriesRemaining (struct BIO_TEMPLATE *refTemplate)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory

This function returns the number of times remaining that an incorrect candidate template can be presented before the reference template is blocked.

A.1.11 bioReset

```
void bioReset (struct BIO_TEMPLATE *refTemplate)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory

This function resets the resets the reference validated flag.

A.1.12 bioInitMatch

```
SWORD bioInitMatch (struct BIO_TEMPLATE *refTemplate, BYTE
*dataBuffer, WORD dataLength)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE *dataBuffer: pointer to (a part of) the candidate template
- WORD dataLength: length of the candidate data to be used

This function initialises or re-initialises a biometric matching session. The exact return score value is implementation-dependant and can be used, for example, to code a confidence rate. The returns score cam fall into one of the following bands:

0…(BIO_MINIMUM_SUCCESSFUL_MATCH_SCORE-1): the match has failed.
BIO_MINIMUM_SUCCESSFUL_MATCH_SCORE or more: the match has succeeded.
BIO_MATCH_NEEDS_MORE_DATA: the match process requires more data.

If a matching session is in progress, a call to `bioInitMatch()` makes the current session to end in the failed state and starts a new matching session.

A.1.13 bioMatch

```
SWORD bioMatch (struct BIO_TEMPLATE *refTemplate, BYTE
*dataBuffer, WORD dataLength)
```

The parameters are:

- struct BIO_TEMPLATE *refTemplate: pointer to the reference template data memory
- BYTE *dataBuffer: pointer to (a part of) the candidate template
- WORD dataLength: length of the candidate data to be used

This function continues the biometric matching session. Refer to `bioInitMatch()` for further details.

B. Mapping of C-API v1 to C-API v2

This appendix lists the API definitions from v1 of the C-API and indicates what has happened to them in v2. It is intended to help porting applications from v1 of the API to v2.

B.1 Macros that have been replaced

The follow macros have been replaced with a function prototype and implemented (using bytecode substitution) in libc.

V1 name	V2 name(s) – if different
multosAcceleratedReadBAC	
multosAcceleratedReadBACLite	
multosAcceleratedReadNoBAC	
multosBlockCompare	multosCompare
multosBlockCompareFixedLength	multosCompareFixedLength
multosBlockCopy	multosCopy
multosBlockCopyFixedLength	multosCopyFixedLength
multosBlockCopyNonAtomic	multosCopyNonAtomic
multosBlockCopyNonAtomicFixedLength	multosCopyFixedLengthNonAtomic
multosBlockDecipherCBC	multosDecipherCBC
multosBlockDecipherECB	multosDecipherECB
multosBlockEncipherCBC	multosEncipherCBC
multosBlockEncipherECB	multosEncipherECB
multosBlockLookup	multosLookup
multosBlockLookupWord	multosLookupWord
multosBlockRotateLeft	multosRotateLeft
multosBlockRotateRight	multosRotateRight
multosBlockShiftLeftVar	multosShiftLeft
multosBlockShiftRightVar	multosShiftRight
multosCallCodelet	
multosCardBlock	
multosCardUnBlock	
multosCheckCase	
multosChecksum	
multosControlAutoResetWWT	multosEnableAutoResetWWT & multosDisableAutoResetWWT
multosConvertBCD	multosBCDtoBIN & multosBINtoBCD
multosCopyAdditionalToStatic	multosCopyFromAdditionalStatic & multosCopyFromAdditionalStaticAtomic
multosCopyToAdditionalStatic	multosCopyToAdditionalStatic & multosCopyToAdditionalStaticAtomic
multosCopyWithinAdditionalStatic	multosCopyWithinAdditionalStatic & multosCopyWithinAdditionalStaticAtomic
multosDeactivateAcceleratedRead	
multosDelegate	
multosEccDiffieHelman	multosECDH & multosECDHProtected
multosEccECIESDecipher	multosECIESDecipher

multosEccGenerateKeyPair	multosEccGenerateKeyPair & multosEccGenerateKeyPairProtected
multosEccGenerateSignature	multosECDSA & multosECDSAProtected
multosEccVerifySignature	multosECDSAVerify
multosExchangeData	
multosExit	
multosExitToMultosAndRestart	
multosFillAdditionalStatic	multosFillAdditionalStatic & multosFillAdditionalStaticAtomic
multosFlushPublic	
multosGenerateDESCBCSignature	
multosGenerateRsaKeyPair	
multosGenerateTripleDESCBCSignature	
multosGetData	
multosGetDelegatorAID	
multosGetDIRFileRecord	multosGetDIRFile
multosGetFileControlInformation	multosGetFCI
multosGetManufacturerData	
multosGetMemoryReliability	
multosGetMultosData	
multosGetPINStatus	
multosGetPINTryCounter	
multosGetPINTryLimit	
multosGetProcessEvent	
multosGetRandomNumber	
multosGetStaticSize	
multosGsmAuthenticate	
multosInitialisePIN	
multosModularExponentiation	
multosModularExponentiationCRT	
multosModularExponentiationCRTProtected	
multosPad	
multosPlatformOptimisedChecksum	
multosQueryChannel	
multosQueryCodelet	
multosQueryCryptographicAlgorithm	
multosQueryInterfaceType	
multosReadPIN	
multosRejectProcessEvent	
multosResetSessionData	
multosResetWWT	
multosReturnFromCodelet	
multosRsaVerify	
multosSecureHash	
multosSecureHashIV	

multosSetATRFileRecord	
multosSetATRHistoricalCharacters	
multosSetATSHistoricalCharacters	
multosSetFCIFileRecord	
multosSetPINTryCounter	
multosSetPINTryLimit	
multosSetProtectedMemoryAccess	
multosSetSelectCLSW	
multosSetSelectSW	
multosSetSilentMode	
multosSetTransactionProtection	
multosSHA1	
multosUnPad	
multosUpdateStaticSize	
multosVerifyPIN	

B.2 Macros that have been changed

The following macros are functionally the same but have either had their name changed or have been compressed onto a single line.

V1 name	V2 name(s) – if different
multosBlockAdd	multosAdd
multosBlockAnd	multosAnd
multosCallExtensionPrimitive	
multosCheckCase	
multosBlockClear	multosClear
multosBlockDecrement	multosDecrement
multosExitLa	
multosExitSW	
multosExitSWLa	
multosBlockIncrement	multosIncrement
multosBlockInvert	multosInvert
multosBlockOr	multosOr
multosBlockXor	multosXor

B.3 Macros that have been deprecated

The following v1 macros are no longer supported in v2. This is because they support older primitives that have been replaced with better alternatives. If using old cards

V1 name	V2 function to use instead	Remark
multosAESECBDecipher	multosDecipherECB	V2 function uses Block Encipher primitive.
multosAESECBEncipher	multosEncipherECB	V2 function uses

		Block Encipher primitive.
multosBlockShiftLeft	multosShiftLeft	V2 function uses Shift Rotate primitive.
multosBlockShiftRight	multosShiftRight	V2 function uses Shift Rotate primitive.
multosDESECBDecipher	multosDecipherECB	V2 function uses Block Encipher primitive.
multosDESECBEncipher	multosEncipherECB	V2 function uses Block Encipher primitive.
multosGenerateAsymmetricHash	n/a	
multosGenerateAsymmetricHashIV	n/a	
multosModularMultiplication	n/a	
multosModularReduction	n/a	

----- End of Document -----