C-API

MULTOS Standard C-API

MAO-DOC-TEC-016 v1.4.1

Copyright

Trademarks

Published by

MAOSCO Limited
1st Floor,
GPS House,
215 Great Portland Street,
W1W 5PN,
London,
United Kingdom.

General Enquiries

Email: dev.support@multos.com
Web: http://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 (http://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. The Standard C API will be supported by MULTOS Development Tools either directly or by using C header libraries. A header file suitable for use with SmartDeck, **multos.h**, can be downloaded from the **multos.com** website.

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.

Only functions that provide an interface to a MULTOS instruction or primitive are included. For this reason only the API of the function is described, not the function itself. If further clarification of a particular function is required, including examples, consult the MULTOS Developers Reference Manual, [MDRM].

The Appendices to this document contain 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 multosBlockAdd
ADDW	handled by multosBlockAdd
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 multosBlockSubtract
SUBW	handled by multosBlockSubtract
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)

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

4.1 multosAcceleratedReadNoBAC

```
void multosAcceleratedReadNoBAC (BYTE channel, BYTE* dataAddr,
BOOL *success)
```

The parameters are:

- BYTE channel: always set to 0 (input)
- BYTE *dataAddr: address of the 8 byte parameter block for the channel (input)
- BOOL *success: set to true if the command is successful (output)

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

4.2 multosAcceleratedReadBAC

```
void multosAcceleratedReadBAC (BYTE* SSCAddr, BYTE* KeyMacAddr,
BYTE* KeyEncAddr, BYTE channel, BYTE* dataAddr, BOOL optional,
BOOL *success)
```

The parameters are:

- BYTE* SSCAddr: address of an 8 byte counter used in the MAC computation (input)
- BYTE* KeyMacAddr: address of the 16 byte MAC key (input)
- BYTE* KeyEncAddr: address of the 16 byte ENC key (input)
- BYTE channel: always set to 0 (input)
- BYTE* dataAddr: address of the 8 byte parameter block for the channel (input)
- BOOL optional: set to **true** if the use of secure messaging is to be optional (input)
- BOOL *success: set to true if the command is successful (output)

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

4.3 multosAcceleratedReadBACLite

```
void multosAcceleratedReadBACLite (BYTE* KeyEncAddr, BYTE
channel, BYTE* dataAddr, BOOL optional, BOOL *success)
```

The parameters are:

- BYTE* KeyEncAddr: address of the 16 byte ENC key (input)
- BYTE channel: always set to 0 (input)
- BYTE* dataAddr: address of the 8 byte parameter block for the channel (input)
- BOOL optional: set to true if the use of encryption is to be optional (input)
- BOOL *success: set to true if the command is successful (output)

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

4.4 multosAESECBDecipher

```
void multosAESECBDecipher (BYTE *cipherText, BYTE *plainText,
BYTE keyLength, BYTE *key)
```

The parameters are:

- BYTE *cipherText: address of 16 bytes of ciphertext (input)
- BYTE *plaintext: address of buffer to hold plaintext (output)
- BYTE keyLength: length of key in bytes (input)
- BYTE *key: address of key to use (input)

This is an interface to the primitive AES ECB Decipher.

4.5 multosAESECBEncipher

```
void multosAESECBEncipher (BYTE *plainText, BYTE *cipherText,
BYTE keyLength, BYTE *key)
```

The parameters are:

- BYTE *plaintext: address of 16 bytes of plaintext (input)
- BYTE *cipherText: address of buffer to hold ciphertext (output)
- BYTE keyLength: length of key in bytes (input)
- BYTE *key: address of key to use (input)

This is an interface to the primitive AES ECB Encipher.

4.6 multosBlockAdd

```
void multosBlockAdd (const BYTE blockLength, BYTE *block1, BYTE
*block2, const 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
- const 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.

4.7 multosBlockAnd

```
void multosBlockAnd (const BYTE blockLength, BYTE *block1, BYTE
*block 2, const 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.

4.8 multosBlockCompare

```
BYTE multosBlockCompare (WORD blockLength, BYTE *block1, BYTE
*block2)
```

The parameters are:

- WORD *blockLength*: size of the blocks to be compared. Both blocks must be the same size.
- BYTE **block1*: address of the first block
- BYTE **block2*: 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 *block1*and *block2* are considered to be of size *blockLength*.

This is an interface to the primitive Memory Compare.

Note: The function multosBlockCompareFixedLength, with identical prototype, also exists but uses the primitive *Memory Compare Fixed Length* instead.

4.9 multosBlockClear

```
void multosBlockClear (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.

4.10 multosBlockCopy

```
void multosBlockCopy (WORD blockLength, BYTE *blockSource, BYTE
*blockDest)
```

The parameters are:

- WORD *blockLength*: the size of the block to copy
- BYTE **blockSource*: address of the source block
- BYTE **blockDest*: address of the destination block

This function copies data of size *blockLength* from *blockSource* to *blockDest*.

This is an interface to the primitive Memory Copy.

Note 1: An alternative multosBlockCopyFixedLength function is implemented that interfaces to the primitive *Memory Copy Fixed Length*. blockLength is a BYTE in this case.
Note 2: Non-atomic versions of these functions are available that call the non-atomic memory copy primitives. See [MDRM] for details and a definition of non-atomic memory operations.

4.11 multosBlockDecipherCBC

```
void multosBlockDecipherCBC (const BYTE algorithm, WORD inputLength,
                             BYTE *cipherText, BYTE *plainText,
                             BYTE initialValueLength, BYTE *initialValue,
                             BYTE keyLength, BYTE *key);
```

The parameters are:

- const BYTE algorithm: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES 128bit (input)
- WORD inputLength: Length of cipherText (input)
- BYTE *cipherText: pointer to data to decipher (input)
- BYTE *plainText: pointer to memory in which to write the plain text output. (output)
- BYTE initialValueLength: length of the Initial Chaining Vector pointed to by initialValue (input)
- BYTE *initialValue: Pointer to ICV value (input)
- BYTE keyLength: length of key, depends on algorithm being used (input)
- BYTE *key: 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.

4.12 multosBlockDecipherECB

```
void multosBlockDecipherECB (const BYTE algorithm, WORD inputLength,
```

```
BYTE *cipherText, BYTE *plainText, BYTE
keyLength, BYTE *key);
```

The parameters are:

- const BYTE algorithm: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES 128bit (input)
- WORD inputLength: Length of cipherText (input)
- BYTE *cipherText: pointer to data to decipher (input)
- BYTE *plainText: pointer to memory in which to write the plain text output. (output)
- BYTE keyLength: length of key, depends on algorithm being used (input)
- BYTE *key: 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.

4.13 multosBlockDecrement

```
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.

4.14 multosBlockDivide

```
void multosBlockDivide (const BYTE blockLength, BYTE *numerator,
BYTE *denominator, BYTE *quotient, BYTE *remainder, BYTE
*status)
```

The parameters are:

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

This function divides the *numerator* by the *denominator*. The results of the operation are written to the blocks *quotient* and *remainder*. In order to flag any special circumstances, the flag *status* is set to

one of the following values: MULTOS_DIVIDE_BY_ZERO, MULTOS_DIVIDE_QUOTIENT_ZERO, MULTOS_DIVIDE_OK.

This is an interface to the primitive DivideN.

4.15 multosBlockEncipherCBC

```
void multosBlockEncipherCBC (const BYTE algorithm, WORD inputLength,
                             BYTE *plainText, BYTE *cipherText,
                             BYTE initialValueLength, BYTE *initialValue,
                             BYTE keyLength, BYTE *key);
```

The parameters are:

- const BYTE algorithm: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES 128bit (input)
- WORD inputLength: Length of plainText (input)
- BYTE *plainText: pointer to plain text to encipher (input)
- BYTE *cipherText: pointer to memory in which to write the cipher text output. (output)
- BYTE initialValueLength: length of the Initial Chaining Vector pointed to by initialValue (input)
- BYTE *initialValue: Pointer to ICV value (input)
- BYTE keyLength: length of key, depends on algorithm being used (input)
- BYTE *key: pointer to the key to use (input)

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

This is an interface to the primitive Block Encipher.

4.16 multosBlockEncipherECB

```
void multosBlockEncipherECB (const BYTE algorithm, WORD inputLength,
                             BYTE *plainText, BYTE *cipherText, BYTE
                             keyLength, BYTE *key);
```

The parameters are:

- const BYTE algorithm: 0x03 = DES, 0x04 = 3DES, 0x05 = SEED, 0x06 = AES 128bit (input)
- WORD inputLength: Length of plainText (input)
- BYTE *plainText: pointer to plain text to encipher (input)
- BYTE *cipherText: pointer to memory in which to write the cipher text output. (output)
- BYTE keyLength: length of key, depends on algorithm being used (input)
- BYTE *key: pointer to the key to use (input)

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

This is an interface to the primitive Block Encipher.

4.17 multosBlockIncrement

```
void multosBlockIncrement (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

4.18 multosBlockInvert

```
void multosBlockInvert (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

4.19 multosBlockLookup

```
BOOL multosBlockLookup (BYTE value, BYTE *block, BYTE *result,
BOOL *byteFound)
```

The parameters are:

- BYTE *value*: the value to locate
- BYTE **block*: address of the array to be searched.
- BYTE **result*: address of the byte to which the result will be written
- BOOL *byteFound: true if the value is found

This function locates the first occurrence of *value* within *block*. Note that this function treats the first byte of the array as indicating the total number of bytes in the array. The function sets byteFound to TRUE if *value* is found and *result* is the offset within the array where *value* is first found.

This is an interface to the primitive Lookup.

4.20 multosBlockLookupWord

```
BOOL multosBlockLookupWord (WORD value, WORD *block, WORD
*result, BOOL *wordFound)
```

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 *block*. Note that this function treats the first word of the array as indicating the total number of words in the array. The function sets wordFound to TRUE if *value* is found and *result* is the offset within the array where *value* is first found.

This is an interface to the primitive Lookup Word.

4.21 multosBlockMultiply

```
void multosBlockMultiply (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.

4.22 multosBlockOr

```
void multosBlockOr (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 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.

4.23 multosBlockRotateLeft

```
void multosBlockRotateLeft (WORD num_bits, WORD data_len, BYTE
*data_addr)
```

The parameters are:

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

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

This is an interface to the primitive Shift Rotate.

4.24 multosBlockRotateRight

```
void multosBlockRotateRight (WORD num_bits, WORD data_len, BYTE
*data_addr)
```

The parameters are:

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

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

This is an interface to the primitive Shift Rotate.

4.25 multosBlockShiftLeft

```
void multosBlockShiftLeft (const BYTE blockLength, const BYTE
numShiftBits, BYTE *blockSource, BYTE *blockDest)
```

The parameters are:

- const BYTE *blockLength*: size of the block to shift
- const BYTE *numShiftBits*: number of bits to shift
- BYTE **blockSource*: address of the source block
- BYTE **blockDest*: address of the destination block

This function does left shifts on the value found in *blockSource, numShiftBits* times and writes the result to *blockDest*.

This is an interface to the primitive Shift Left.

4.26 multosBlockShiftLeftVar

```
void multosBlockShiftLeftVar (WORD num_bits, WORD data_len, BYTE
*data_addr)
```

The parameters are:

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

The function performs left shifts on the value found in *data_addr*. Unlike *multosBlockShiftLeft()* the number of bits to shift by can be variable.

This is an interface to the primitive Shift Rotate.

4.27 multosBlockShiftRight

```
void multosBlockShiftRight (const BYTE blockLength, const BYTE
numShiftBits, BYTE *blockSource, BYTE *blockDest)
```

The parameters are:

const BYTE *blockLength*: size of the block to shift
const BYTE *numShiftBits*: number of bits to shift
BYTE **blockSource*: source block
BYTE **blockDest*: destination block

This function does right shifts the value found in *blockSource, numShiftBits* times and writes the result to *blockDest*.

This is an interface to the primitive Shift Right.

4.28 multosBlockShiftRightVar

```
void multosBlockShiftRightVar (WORD num_bits, WORD data_len,
BYTE *data_addr)
```

The parameters are:

- WORD *num_bits:* the number of bits to shift by

- WORD *data_len*: the length of the data (in bytes) pointed to by *data_addr*
- BYTE* *data_addr*: the address of the data to be shifted

The function performs right shifts on the value found in *data_addr*. Unlike *multosBlockShiftRight()* the number of bits to shift by can be variable.

This is an interface to the primitive Shift Rotate.

4.29 multosBlockSubtract

```
void multosBlockSubtract (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.

4.30 multosBlockTestZero

```
void multosBlockTestZero (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.

4.31 multosBlockXor

```
void multosBlockXor (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.

4.32 multosCallCodelet

```
void multosCallCodelet (WORD codeletID, WORD entryAddress)
```

The parameters are:

- WORD *codeletID*: the unique identifier of the codelet being called
- WORD *entryAddress*: 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.

4.33 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.

4.34 multosCardBlock

```
void multosCardBlock (const BYTE offsetMAChi, const BYTE
offsetMAClo, BOOL *cardBlockedOk)
```

The parameters are:

- const BYTE *offsetMAChi*: most significant byte of the address of the CBMAC in public memory
- const BYTE *offsetMAClo*: least significant byte of the address of the CBMAC in public memory
- BOOL **cardBlockedOk*: set to TRUE if the block is successful

This function evaluates a card block MAC and blocks the MULTOS card if the MAC is verified.

This is an interface to the primitive Card Block.

4.35 multosCardUnBlock

```
void multosCardUnBlock (BOOL *cardUnBlockedOk)
```

Calls the primitive Card Unblock and sets cardUnBlockedOk to true if the card is successfully unblocked.

4.36 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.

4.37 multosChecksum

```
void multosChecksum (WORD blockLength, BYTE *block, DWORD
*result)
```

The parameters are:

- WORD *blockLength*: the length of the block to use as input to the checksum algorithm
- BYTE **block*: the address of the first byte of the input block
- DWORD *result: address of the resulting 4 byte checksum.

The function generates a four-byte checksum of block and returns the four-byte value.

This is an interface to the primitive Checksum.

4.38 multosControlAutoResetWWT

```
void multosControlAutoResetWWT (BOOL disable)
```

The parameter is Boolean value set to TRUE if the auto reset WWT request function is to be disabled.

This function enables or disables the automatic generation of WWT extension messages.

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

4.39 multosConvertBCD

```
void multosConvertBCD (BYTE mode, BYTE srcLength, BYTE destLength,
                       BYTE *src, BYTE *dest, BOOL *convertedOK);
```

The parameters are:

- BYTE mode: 0x00 = BCD to BIN, 0x01 = BIN to BCD (input)
- BYTE srcLength: length of data to convert (input)
- BYTE destLength: length of output buffer dest (input)
- BYTE *dest: pointer to output buffer to hold result of coversion (output)
- BOOL *convertedOK: zero if conversion fails (output)

This function converts between binary and binary coded decimal and vice-versa.

This is an interface to the primitive Convert BCD.

4.40 multosCopyAdditionalToStatic

```
void multosCopyAdditionalToStatic(BYTE* LONG destAddr, DWORD
srcAddr, WORD toCopy, BYTE atomic)
```

The parameters are:

- BYTE *destAddr: address in “normal” static to copy to (input)
- DWORD srcAddr: the offset in “additional” static to copy from (input)
- WORD toCopy: the number of bytes to copy (input)
- BYTE atomic: if greater than zero, the copy will be atomic (input)

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

4.41 multosCopyToAdditionalStatic

```
void multosCopyToAdditionalStatic (DWORD destAddr, BYTE* srcAddr,
WORD toCopy, BYTE atomic)
```

The parameters are:

- DWORD destAddr: the offset in “additional” static to copy to* (input)
- BYTE* srcAddr: pointer to the data in “normal” static to copy (input)
- WORD toCopy: the number of bytes to copy (input)
- BYTE atomic: if greater than zero, the copy will be atomic (input)

* Note: “additional” static memory is contiguous with “normal” static memory and shares the same base address. However, “additional” static may extend beyond 64K bytes so cannot be addressed with a normal 2 byte address.

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

4.42 multosCopyWithinAdditionalStatic

```
void multosCopyWithinAdditionalStatic (DWORD destAddr, DWORD
srcAddr, DWORD toCopy, BYTE atomic)
```

The parameters are:

- DWORD destAddr: the offset in "additional" static to copy to (input)
- DWORD srcAddr: the offset in "additional" static to copy (input)
- DWORD toCopy: the number of bytes to copy (input)
- BYTE atomic: if greater than zero, the copy will be atomic (input)

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

4.43 multosDeactivateAcceleratedRead

```
void multosDeactivateAcceleratedRead (BOOL *success)
```

This calls the primitive Configure Read Binary to deactivate accelerated read mode for the Read Binary command.

4.44 multosDelegate

```
void multosDelegate (BYTE *aid)
```

The parameter is a pointer to the AID of the application that will be delegated to.

This function delegates execution to an application with and AID of aid.

This is an interface to the primitive Delegate.

4.45 multosDESECBDecipher

```
void multosDESECBDecipher (BYTE *cipherText, BYTE *plainText,
BYTE key[8])
```

The parameters are:

- BYTE **cipherText*: address of the ciphertext input
- BYTE **plaintext*: address to which to write the plaintext
- BYTE *key[8]*: the 8-byte DES key

This function performs a DES ECB decipher on eight bytes of *cipherText* using key into eight bytes of *plainText*.

This is an interface to the primitive DES ECB Decipher.

4.46 multosDESECBEncipher

```
void multosDESECBEncipher (BYTE *plainText, BYTE *cipherText,
BYTE key[8])
```

The parameters are:

- BYTE **plaintext*: address of the plaintext input
- BYTE **cipherText*: address to which to write the ciphertext
- BYTE *key[8]*: the 8-byte DES key

This function performs a DES ECB encipher on eight bytes of *plainText* using key into eight bytes of *cipherText*.

This is an interface to the primitive DES ECB Encipher.

4.47 multosEccGenerateKeyPair

```
void multosEccGenerateKeyPair(BYTE *keyOut, BYTE *domain, const
BYTE option, BOOL *success)
```

The parameters are:

- BYTE *keyOut: address of buffer (size = 3 x prime length) to hold generated key pair (output)
- BYTE *domain: address of domain parameters structure (input)
- const BYTE option: 0x80 to protect private key, 0x00 otherwise (input)
- BOOL *success: set to true if key pair generated successfully (output)

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

4.48 multosEccGenerateSignature

```
void multosEccGenerateSignature(BYTE *domainIn, BYTE *keyIn, BYTE
*hashIn, BYTE *sigOut, const BYTE option, BOOL *success)
```

The parameters are:

- BYTE *domainIn: address of domain parameters structure (input)
- BYTE *keyIn: address of the private key to use for signing (input)
- BYTE *hashIn: address of the hash to sign (input)
- BYTE *sigOut: address of buffer to hold the signature (output)
- const BYTE option: 0x80 protected private key, 0x00 otherwise (input)
- BOOL *success: set to true if the signature generated successfully (output)

This is an interface to the primitive ECC Generate Signature.

4.49 multosEccVerifySignature

```
void multosEccVerifySignature(BYTE *domainIn, BYTE *hashIn, BYTE
*sigIn, BYTE *pubKeyIn, const BYTE option, BOOL *success)
```

The parameters are:

- BYTE *domainIn: address of domain parameters structure (input)
- BYTE *hashIn: address of the hash to compare (input)
- BYTE *sigIn: address of the signature to decrypt and compare to hashIn (input)
- BYTE *pubKeyIn: address of the public key to use to decrypt sigIn (input)
- const BYTE option: fixed, 0x00 (input)
- BOOL *success: set to true if the signature is verified (output)

This is an interface to the primitive ECC Verify Signature.

4.50 multosEccDiffieHelman

```
void multosEccDiffieHelman(BYTE *domainIn, BYTE *privateKeyIn,
BYTE *publicKeyIn, BYTE *bufferOut, BYTE option, BOOL *success)
```

The parameters are:

- BYTE *domainIn: address of domain parameters structure (input)
- BYTE *privateKeyIn: private part of key A (input)
- BYTE *publicKeyIn: public part of key B (input)
- BYTE *bufferOut: address of buffer to store computed shared secret key (output)
- BYTE option: 0x80 protected private key, 0x00 otherwise (input)
- BOOL *success: set to true if the process is successful (output)

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

4.51 multosEccECIESEncipher

```
void multosEccECIESEncipher(BYTE *domainIn, WORD msgLen, BYTE
*publicKeyIn, BYTE *msgIn, BYTE *bufferOut, BYTE option, BOOL
*success)
```

The parameters are:

- BYTE *domainIn: address of domain parameters structure (input)
- WORD msgLen: the length of the message to encipher (input)
- BYTE *publicKeyIn: address of the public key to use (input)
- BYTE *msgIn: address of the message to encipher (input)
- BYTE *bufferOut: address of the buffer to hold the ciphertext (output)
- BYTE option: refer to [MDRM] (input)
- BOOL *success: set to true if the process is successful (output)

This is an interface to the primitive ECC ECIES Encipher.

4.52 multosEccECIESDecipher

```
void multosEccECIESDecipher(BYTE *domainIn, WORD msgLen, BYTE
*privateKeyIn, BYTE *msgIn, BYTE *bufferOut, BYTE option, BOOL
*success)
```

The parameters are:

- BYTE *domainIn: address of domain parameters structure (input)

- WORD msgLen: the length of the message to decipher (input)
- BYTE *privateKeyIn: address of the key to use (input)
- BYTE *msgIn: address of ciphertext (input)
- BYTE *bufferOut: address of buffer to hold cleartext (output)
- BYTE option: refer to [MDRM] (input)
- BOOL *success: set to true if the process is successful (output)

This is an interface to the primitive ECC ECIES Decipher.

4.53 multosExchangeData

```
void multosExchangeData (BYTE channelID, BYTE *data)
```

The parameters are:

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

This function exchanges data through channel with ID *channelID*.

This is an interface to the primitive Exchange Data.

4.54 multosExit

```
void multosExit (void)
```

This function exits application.

This is an interface to the instruction SYSTEM.

4.55 multosExitLa

```
void multosExitLa (const BYTE la)
```

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

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

This is an interface to the instruction SYSTEM.

4.56 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.

4.57 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.

4.58 multosExitToMultosAndRestart

```
void multosExitToMultosAndRestart (void);
```

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

4.59 multosFillAdditionalStatic

```
void multosFillAdditionalStatic (DWORD destAddr, BYTE value,
DWORD toFill, BYTE atomic)
```

The parameters are:

- DWORD destAddr: starting address in "additional" static to fill (input)
- BYTE value: the value to fill "additional" static with (input)
- DWORD toFill: the number of bytes to fill (input)
- BYTE atomic: if greater than zero, the copy will be atomic (input)

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

4.60 multosFlushPublic

```
void multosFlushPublic (WORD blockSize, BOOL *LaFlag)
```

The parameters are:

- WORD *blockSize*: the number of bytes in the block of data to send to flush from public. (input)
- BOOL **LaFlag*: set to true if La is set to a value > 0 (output)

This is an interface to the primitive Flush Public.

4.61 multosGenerateAsymmetricHash

```
void multosGenerateAsymmetricHash (WORD plainTextLength, BYTE
*plainText, BYTE hash[16])
```

The parameters are:

- WORD *plainTextLength*: the length of the input data
- BYTE **plainText*: the address of the input data
- BYTE *hash[16]*: array that holds the resulting hash digest

This function generates a 16 byte hash from *plainText*.

This is an interface to the primitive Generate Asymmetric Hash, where the default IV is used.

4.62 multosGenerateAsymmetricHashIV

```
void multosGenerateAsymmetricHashIV (BYTE initialValue[16], WORD
plainTextlength, BYTE *plainText, BYTE hash[16])
```

The parameters are:

- BYTE *initialValue[16]*: the IV to use for the asymmetric hash algorithm
- WORD *plainTextLength*: the length of the input data
- BYTE **plainText*: the address of the input data
- BYTE *hash[16]*: array that holds the resulting hash digest

This function generates a 16 byte hash from the value held at *plainText* using IV *initialValue*.

This is an interface to the primitive Generate Asymmetric Hash.

4.63 multosGenerateDESCBCSignature

```
void multosGenerateDESCBCSignature (WORD plainTextLength, BYTE
*plainText, BYTE initialValue[8], BYTE key[8], BYTE
signature[8])
```

The parameters are:

- WORD *plainTextLength*: the length of the input data
- BYTE **plaintext*: the address of the input data
- BYTE *initialValue[8]*: initial value to use in DES CBC algorithm
- BYTE *key[8]*: DES key to use
- BYTE *signature[8]*: array to hold 8-byte signature

This function generates an eight byte DES CBC Signature over *plainText* using *initialValue* and *key* and writes the resulting value to *signature*.

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

4.64 multosGenerateRsaKeyPair

```
void multosGenerateRsaKeyPair ( const BYTE method, const BYTE
mode, WORD keyLen, WORD expLen, BYTE* expAddr, BYTE* dpdqpquAddr,
BYTE* mAddr, WORD modLen, BOOL *success)
```

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 expLen: length, in bytes, of the given exponent (input)
- BYTE* expAddr: 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 modLen: equal to keyLen or 0 if the modulus is not to be returned (input)
- BOOL *success: set to true if the command is successful (output)

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

4.65 multosGenerateTripleDESCBCSignature

```
void multosGenerateTripleDESCBCSignature (WORD plainTextLength,
BYTE *plainText, BYTE initialValue[8], BYTE keys[16], BYTE
signature[8])
```

The parameters are:

- WORD *plainTextLength*: the length of the input data
- BYTE **plaintext*: the address of the input data
- BYTE *initialValue[8]*: initial value to use in Triple DES CBC algorithm
- BYTE *key[16]*: DES keys to use
- BYTE *signature[8]*: array to hold 8-byte signature

This function generates an eight byte DES CBC Signature over *plainText* using *initialValue* and *key* and writes the resulting value to *signature*.

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

4.66 multosGetData

```
void multosGetData (const BYTE numDataBytes, BYTE *numCopied,
                    BYTE *output)
```

The parameters are:

- const BYTE numDataBytes: the maximum number of bytes to read – should correspond with the size of the output buffer (input)

- BYTE *numCopied: the number of bytes actually copied to the output buffer (output)
- BYTE *output: pointer to a buffer to hold the output of the function.

This is an interface to the primitive Get Data.

4.67 multosGetDelegatorAID

```
void multosGetDelegatorAID (const BYTE aidLength, BYTE *aid,
BOOL *notADelegate)
```

The parameters are:

- const BYTE aidLength: length of the AID to fetch
- BYTE *aid: address where delegator AID is written
- BOOL *notADelegate: Boolean value indicating if the application has been delegated to

This function stores the AID of the delegating application into *aid* and sets the flag *notADelegate* to TRUE if the application has not been delegated to.

This is an interface to the primitive Get Delegator AID.

4.68 multosGetDIRFileRecord

```
void multosGetDIRFileRecord (const BYTE dirRecordLength, BYTE
recordNum, BYTE *numCopied, BYTE *output, BOOL *noRecordFound)
```

The parameters are:

- const BYTE *dirRecordLength*: the number of bytes of the DIR record to copy
- BYTE *recordNum*: DIR record number
- BYTE **numCopied*: actual number of bytes copied
- BYTE **output*: address where result is written
- BOOL **noRecordFound*: Boolean value

This function retrieves the record *recordNum* from the DIR File and copies it to *output*. It also indicates the actual number of bytes copied in *numCopied* as well as setting the flag *noRecordFound* to TRUE if there is no record *recordNum* in the DIR File.

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

4.69 multosGetFileControlInformation

```
void multosGetFileControlInformation (const BYTE
fciRecordLength, BYTE recordNum, BYTE *numCopied, BYTE *output,
BOOL *noRecordFound)
```

The parameters are:

- const BYTE *fciRecordLength*: the number of bytes of the FCI record to copy
- BYTE *recordNum*: FCI record number
- BYTE **numCopied*: actual number of bytes copied
- BYTE **output*: address where result is written
- BOOL **noRecordFound*: Boolean value

This function retrieves the record *recordNum* from the FCI file and copies it to *output*. It also indicates the actual number of bytes copied in *numCopied* as well as setting the flag *noRecordFound* to TRUE if there is no record *recordNum* in the FCI file.

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

4.70 multosGetManufacturerData

```
void multosGetManufacturerData (const BYTE numDataBytes, BYTE
*numCopied, BYTE *output)
```

The parameters are:

- const BYTE *numDataBytes*: the number of bytes to be copied
- BYTE **numCopied*: actual number of bytes copied
- BYTE **output*: address where result is written

This function retrieves the Manufacturer Data from MULTOS chip and writes the result to *output*. It also indicates the actual number of bytes copied in *numCopied*.

This is an interface to the primitive Get Manufacturer Data.

4.71 multosGetMemoryReliability

```
void multosGetMemoryReliability (BYTE *result)
```

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

```
void multosGetMultosData (const BYTE numDataBytes, BYTE
*numCopied, BYTE *output)
```

The parameters are:

- const BYTE *numDataBytes*: the number of bytes to be copied

- BYTE **numCopied*: actual number of bytes copied
- BYTE **output*: address where result is written

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

This is an interface to the primitive Get MULTOS Data.

4.73 multosGetPINStatus

```
void multosGetPINStatus (BYTE *status)
```

The parameter holds the returned status value, as defined in the [MDRM].

This is an interface to the primitive Get PIN Data.

4.74 multosGetPINTryCounter

```
void multosGetPINTryCounter (BYTE *ptc)
```

The parameter returns the current PIN Try Counter value.

This is an interface to the primitive Get PIN Data.

4.75 multosGetPINTryLimit

```
void multosGetPINTryLimit (BYTE *ptl)
```

The parameter returns the current PIN Try Limit value.

This is an interface to the primitive Get PIN Data.

4.76 multosGetProcessEvent

```
void multosGetProcessEvent (BYTE *event)
```

The parameter returns the ID (one byte) 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.

4.77 multosGetRandomNumber

```
void multosGetRandomNumber (BYTE result[8])
```

The parameter is an eight byte array where the random number will be written.

This function generates an 8-byte random number and writes that value to *result[8]*.

This is an interface to the primitive Get Random Number.

4.78 multosGetStaticSize

```
void multosGetStaticSize ( DWORD* value, BOOL *success )
```

The parameters are:

- DWORD *value: amount of static memory allocated (output)
- BOOL *success: set to true if the command is successful (output)

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.

4.79 multosGetStaticSizeHuge

```
void multosGetStaticSizeHuge ( BYTE* value, BOOL *success )
```

The parameters are:

- BYTE *value: address of 8 byte buffer to hold the amount of static memory allocated (output)
- BOOL *success: set to true if the command is successful (output)

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.

4.80 multosGsmAuthenticate

```
void multosGsmAuthenticate(BYTE* random, BYTE* key, BYTE*
result)
```

The parameters are:

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

This is an interface to the primitive GSM Authenticate.

4.81 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.

4.82 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.

4.83 multosModularExponentiation

```
void multosModularExponentiation (WORD exponentLength, WORD
modulusLength, BYTE *exponent, BYTE *modulus, BYTE *input, BYTE
*output)
```

The parameters are:

- WORD *exponentLength*: the length of the exponent used
- WORD *modulusLength*: the length of the modulus
- BYTE **exponent*: address of the exponent
- BYTE **modulus*: address of the modulus
- BYTE **input*: address of the input value
- BYTE **output*: address of where to write the result of the operation

This function performs a modular exponentiation. Note that the values held at *modulus*, *input* and *output* are all considered to be of size *modulusLength*.

This is an interface to the primitive Modular Exponentiation.

4.84 multosModularExponentiationCRT

```
void multosModularExponentiationCRT (WORD dpdqLength, BYTE
*dpdq, BYTE *pqu, BYTE *input, BYTE *output)
```

The parameters are:

- WORD *dpdqLength*: length of dpdq
- BYTE **dpdq*: address of dpdq
- BYTE **pqu*: address of pqu
- BYTE **input*: address of input
- BYTE **output*: address of 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.

4.85 multosModularMultiplication

```
void multosModularMultiplication (WORD modulusLength, BYTE
*modulus, BYTE *block1, BYTE *block2)
```

The parameters are:

- WORD *modulusLength*: the length of the modulus used
- BYTE **modulus*: address of the modulus
- BYTE **block1*: address of the first operand
- BYTE **block2*: address of the second operand

This function performs a modular multiplication. The result of the operation is written to *block1*.

This is an interface to the primitive Modular Multiplication.

4.86 multosModularReduction

```
void multosModularReduction (WORD operandLength, WORD
modulusLength, BYTE *operand, BYTE *modulus)
```

The parameters are:

- WORD operandLength: the length of the operand
- WORD modulusLength: length of the modulus
- BYTE *operand: address of the operand
- BYTE *modulus: address of the modulus

This function performs a modular reduction. The result overwrites the operand.

This is an interface to the primitive Modular Reduction.

4.87 multosPad

```
void multosPad (WORD msgLen,BYTE *msg,BYTE blockLen,WORD
*paddedMsgLen, const BYTE padMethod)
```

The parameters are:

- WORD msgLen: the length of data pointed to by msg (input)
- BYTE *msg: address of the data to be padded (input/output)
- BYTE blockLen: size in bytes of the block length to pad to (input)
- WORD *paddedMsgLen: address to write the new length of msg to (output)
- BYTE padMethod: (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.

4.88 multosPlatformOptimisedChecksum

```
void multosPlatformOptimisedChecksum (WORD blockLength, BYTE *block,
                                      DWORD *result)
```

The parameters are:

- WORD blockLength: length of the data to calculate the checksum over (input)
- BYTE *block: pointer to the data to calculate the checksum over (input)
- DWORD *result: address of a DWORD to hold the 4 byte checksum (output)

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

4.89 multosQueryInterfaceType

void multosQueryInterfaceType (BOOL *isContactless)

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

4.90 multosQueryChannel

```
void multosQueryChannel (BYTE channelID, BOOL *channelSupported)
```

The parameter is a byte value indicating the channel to query.

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

This is an interface to the primitive Query Channel.

4.91 multosQueryCodelet

```
void multosQueryCodelet (WORD codeletID, BOOL *codeletSupported)
```

The parameter is a byte value indicating the codelet to query.

This function verifies the existence of a specific codelet with ID *codeletID*. Sets codeletSupported to TRUE if codelet is supported.

This is an interface to the primitive Query Codelet.

4.92 multosQueryCryptographicAlgorithm

```
void multosQueryCryptographicAlgorithm (BYTE algoId, BOOL
*isSupported)
```

The parameters are:

- BYTE algoId: The algorithm ID to check, as defined in the [MDRM] (input)
- BOOL *isSupported: address to hold result (output)

isSupported is set to true if the algorithm is supported by the device.

This is an interface to the primitive Query Cryptographic Algorithm.

4.93 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.

4.94 multosReadPIN

```
void multosReadPIN (BYTE *outAddr, BYTE *pinLength);
```

The parameters are:

- BYTE *outAddr: buffer to hold the returned PIN (output)
- BYTE *pinLength: length of the returned PIN (output)

This is an interface to the primitive Read PIN.

4.95 multosRejectProcessEvent

```
void multosRejectProcessEvent (void)
```

This is an interface to the primitive Reject Process Event.

4.96 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.

4.97 multosResetWWT

```
void multosResetWWT (void)
```

This function sends a WWT extension request.

This is an interface to the primitive Reset WWT.

4.98 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.

4.99 multosRsaVerify

```
void multosRsaVerify (WORD exponentLength, WORD modulusLength,
BYTE *exponent, BYTE *modulus, BYTE *input, BYTE *output)
```

The parameters are:

- WORD *exponentLength*: length of the exponent value pointed to by *exponent*. (input)
- WORD *modulusLength*: length of the public modulus pointed to by *modulus.* (input)
- BYTE **exponent*: buffer holding the exponent value (input)
- BYTE **modulus*: buffer holding the public modulus (input)
- BYTE **input*: buffer holding the input to the modular exponentiation operation (input)
- BYTE **output*: 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.

4.100 multosSecureHash

```
void multosSecureHash (WORD msgLen, WORD hashLen, BYTE *hashOut,
BYTE *msgIn)
```

The parameters are:

- WORD msgLen: length of the data pointed to by msgIn (input)
- WORD hashLen: length of the required hash value (input)
- BYTE *hashOut: address of buffer to hold hash result (output)
- BYTE *msgIn: 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].

4.101 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
}
```

4.102 multosSetATRFileRecord

```
void multosSetATRFileRecord (BYTE length, BYTE *record, BYTE
*numBytesWritten)
```

The parameters are:

- BYTE *length*: the length of the record
- BYTE **record*: the address of the value to write to the ATR file
- BYTE *numBytesWritten: the number of bytes written to the ATR file.

This function writes a record into the ATR File returns number of bytes written.

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

4.103 multosSetATRHistoricalCharacters

```
void multosSetATRHistoricalCharacters (BYTE *input, BYTE
*numBytesWritten)
```

The parameters are:

- BYTE **input*: the address of the value to write to the ATR historical characters.
- BYTE *numBytesWritten: the number of bytes written to the ATR

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

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

4.104 multosSetATSHistoricalCharacters

```
void multosSetATSHistoricalCharacters (BYTE *input, BYTE
*numBytesWritten)
```

The parameters are:

- BYTE **input*: the address of the value to write to the ATS historical characters.
- BYTE *numBytesWritten: the number of bytes written to the ATS

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

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

4.105 multosSetFCIFileRecord

```
void multosSetFCIFileRecord (BYTE *record, BYTE
*numBytesWritten)
```

The parameters are:

- BYTE **record*: the address of the FCI data to write (input)
- BYTE *numBytesWritten: the number of bytes written to the FCI (output)

The first byte of the data pointed to by *record* must indicate the length of the remaining day.

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

4.106 multosSetPINTryCounter

```
void multosSetPINTryCounter (BYTE ptc)
```

Sets the current PIN Try Counter to the value given.

This is an interface to the primitive Set PIN Data.

4.107 multosSetPINTryLimit

```
void multosSetPINTryLimit (BYTE ptl)
```

Sets the current PIN Try Limit to the value given.

This is an interface to the primitive Set PIN Data.

4.108 multosSetProtectedMemoryAccess

```
void multosSetProtectedMemoryAccess (const BYTE options)
```

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

4.109 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.

4.110 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.

4.111 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.

4.112 multosSetSilentMode

```
void multosSetSilentMode (const BYTE mode)
```

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

4.113 multosSHA1

```
void multosSHA1 (WORD messageLength, BYTE *message, BYTE *hash)
```

The parameters are:

- WORD *messageLength*: length of the message to submit to the SHA-1 algorithm
- BYTE **message*: address of the message
- BYTE **hash*: address where to write the 20-byte digest

This function uses the value found in *message* of size *messagelength* as input to the SHA-1 hashing algorithm. The resulting 20-byte digest is written to *hash*.

This is an interface to the primitive SHA-1.

4.114 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.

4.115 multosTripleDESECBDecipher

```
void multosTripleDESECBDecipher (BYTE *cipherText, BYTE
*plainText, BYTE keyLength, BYTE *key)
```

The parameters are:

- BYTE *cipherText: address of an 8 byte block of data to decipher (input)
- BYTE *plainText: address of buffer to hold the result (output)
- BYTE keyLength: 16 or 24 indicating 2 or 3 key TDES (input)
- BYTE *key: address of buffer holding the key (input)

This is an interface to the primitive Triple DES Decipher.

4.116 multosTripleDESECBEncipher

```
void multosTripleDESECBEncipher (BYTE *plainText, BYTE
*cipherText, BYTE keyLength, BYTE *key)
```

The parameters are:

- BYTE *plainText: address of an 8 byte block of data to encipher (input)
- BYTE *cipherText: address of buffer to hold the result (output)
- BYTE keyLength: 16 or 24 indicating 2 or 3 key TDES (input)
- BYTE *key: address of buffer holding the key (input)

This is an interface to the primitive Triple DES Encipher.

4.117 multosUnPad

```
void multosUnPad (WORD msgLen,BYTE *msg, WORD *unpaddedMsgLen,
const BYTE padMethod)
```

The parameters are:

- WORD msgLen: the length of data pointed to by msg (input)
- BYTE *msg: address of the data to have the padding removed (input/output)
- WORD *unpaddedMsgLen: address to write the new length of msg to (output)
- BYTE padMethod: (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.

4.118 multosUpdateStaticSize

void **multosUpdateStaticSize** (DWORD length, BYTE * result)

The parameters are:

- DWORD length: The new length to set for the application's static memory.
- BYTE *result: Address of the byte to write the operation's result to. A value of 1 = success, 0 = failed.

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).

This is an interface to the primitive Update Static Size.

4.119 multosVerifyPIN

```
void multosVerifyPIN (BYTE pinLen, BYTE *pinAddr, WORD *status)
```

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)
- WORD *status: Returns the result of the verification process. 0x5AA5 for verified, 0xA55A for NOT verified (output)

This is an interface to the primitive Verify PIN.

A. Appendix A : Biometric C API

A.1 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.

A.2 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.

A.3 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.
```

A.4 Data

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

A.5 Function Prototypes

A.5.1 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.5.2 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.5.3 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.5.4 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.5.5 bioGetBioType

```
BYTE bioGetBioType (void)
```

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

A.5.6 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.5.7 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.5.8 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.5.9 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.5.10 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.5.11 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.5.12 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.5.13 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.

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