Copyright

© Copyright 1999 – 2019 MAOSCO Limited. This document contains confidential and proprietary information. No part of this document may be reproduced, published or disclosed in whole or part, by any means: mechanical, electronic, photocopying, recording or otherwise without the prior written permission of MAOSCO Limited.

Trademarks

MULTOS is a registered trademark of MULTOS Limited.
All other trademarks, trade names or company names referenced herein are used for identification only and are the property of their respective owners.

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. The latest versions are always available from the MULTOS web site http://www.multos.com.

[MDG] mao-doc-ref-005 MULTOS Developer’s Guide
Available under the “MULTOS Application Developer Licence”. To download visit http://www.multos.com

[MIR] mao-doc-ref-010 MULTOS Implementation Report
Available under the “MULTOS Application Developer Licence”. To download visit http://www.multos.com

[OFFCARD] mao-one-off-001 MULTOS step/one Off-Card Specification
Available under license from MAOSCO Limited. For further information, please contact info@multos.com

[FIPS180-3] Secure Hash Standard
National Institute of Standards and Technology (NIST)
# Contents

## INTRODUCTION

MULTOS step/one ................................................................. 1  
Conventions and Assumptions .............................................. 1

## INSTRUCTIONS

- ADDB .................................................................................. 3  
- ADDN .................................................................................. 4  
- ADDW .................................................................................. 6  
- ANDN .................................................................................. 7  
- BRANCH .............................................................................. 9  
- CALL ................................................................................... 11  
- CLEARN ............................................................................ 12  
- CMPB .................................................................................. 14  
- CMPN .................................................................................. 16  
- CMPW .................................................................................. 18  
- DECN .................................................................................. 20  
- INCN .................................................................................. 21  
- INDEX .................................................................................. 21  
- JUMP ................................................................................... 24  
- LOAD .................................................................................. 25  
- LOADA .............................................................................. 26  
- LOADI .................................................................................. 28  
- NOTN .................................................................................. 30  
- ORN .................................................................................... 31  
- PRIMRET ............................................................................ 33  
- SETB .................................................................................. 36  
- SETW .................................................................................. 37  
- STACK ................................................................................ 38  
- STORE ............................................................................... 39  
- STOREI ............................................................................. 41  
- SUBB .................................................................................. 43  
- SUBN .................................................................................. 43  
- SUBW .................................................................................. 46  
- SYSTEM ............................................................................ 46  
- TESTN ............................................................................... 49  
- XORN .................................................................................. 51

## PRIMITIVES

- Add BCDN ............................................................................ 54  
- AES ECB Decipher .............................................................. 56  
- AES ECB Encipher .............................................................. 58  
- Bit Manipulate Byte ............................................................ 60  
- Bit Manipulate Word ............................................................ 63  
- Block Decipher ................................................................. 66  
- Block Encipher ................................................................. 71  
- Call Codelet ...................................................................... 76  
- Call Extension 0,1,2,3,4,5,6 ............................................... 78  
- CardBlock ......................................................................... 80  
- CardUnBlock .................................................................... 82  
- Check BCD ......................................................................... 84
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Case</td>
<td>85</td>
</tr>
<tr>
<td>Checksum</td>
<td>87</td>
</tr>
<tr>
<td>Configure READ BINARY</td>
<td>90</td>
</tr>
<tr>
<td>Control Auto Reset WWT</td>
<td>93</td>
</tr>
<tr>
<td>Convert BCD</td>
<td>95</td>
</tr>
<tr>
<td>Delegate</td>
<td>97</td>
</tr>
<tr>
<td>DES ECB Decipher</td>
<td>99</td>
</tr>
<tr>
<td>DES ECB Encipher</td>
<td>101</td>
</tr>
<tr>
<td>DivideN</td>
<td>103</td>
</tr>
<tr>
<td>ECC Addition</td>
<td>105</td>
</tr>
<tr>
<td>ECC Convert Representation</td>
<td>108</td>
</tr>
<tr>
<td>ECC ECIES Decipher</td>
<td>110</td>
</tr>
<tr>
<td>ECC ECIES Encipher</td>
<td>112</td>
</tr>
<tr>
<td>ECC Elliptic Curve Diffie Hellman</td>
<td>114</td>
</tr>
<tr>
<td>ECC Equality Test</td>
<td>116</td>
</tr>
<tr>
<td>ECC Generate Key Pair</td>
<td>118</td>
</tr>
<tr>
<td>ECC Generate Signature</td>
<td>120</td>
</tr>
<tr>
<td>ECC Inverse</td>
<td>122</td>
</tr>
<tr>
<td>ECC Scalar Multiplication</td>
<td>124</td>
</tr>
<tr>
<td>ECC Verify Point</td>
<td>126</td>
</tr>
<tr>
<td>ECC Verify Signature</td>
<td>128</td>
</tr>
<tr>
<td>Exchange Data</td>
<td>130</td>
</tr>
<tr>
<td>Exit to MULTOS and Restart</td>
<td>132</td>
</tr>
<tr>
<td>Flush Public</td>
<td>134</td>
</tr>
<tr>
<td>Generate Asymmetric Hash General</td>
<td>136</td>
</tr>
<tr>
<td>Generate Asymmetric Signature General</td>
<td>140</td>
</tr>
<tr>
<td>Generate DES CBC Signature</td>
<td>142</td>
</tr>
<tr>
<td>Generate MAC</td>
<td>144</td>
</tr>
<tr>
<td>Generate Random Prime</td>
<td>146</td>
</tr>
<tr>
<td>Generate RSA Key Pair</td>
<td>149</td>
</tr>
<tr>
<td>Generate Triple DES CBC Signature</td>
<td>151</td>
</tr>
<tr>
<td>Get Configuration Data</td>
<td>153</td>
</tr>
<tr>
<td>Get AID</td>
<td>154</td>
</tr>
<tr>
<td>Get Available Interface Types</td>
<td>155</td>
</tr>
<tr>
<td>Get Data</td>
<td>156</td>
</tr>
<tr>
<td>Get Delegator AID</td>
<td>158</td>
</tr>
<tr>
<td>Get DIR File Record</td>
<td>160</td>
</tr>
<tr>
<td>Get FCI State</td>
<td>162</td>
</tr>
<tr>
<td>Get File Control Information</td>
<td>163</td>
</tr>
<tr>
<td>Get Manufacturer Data</td>
<td>165</td>
</tr>
<tr>
<td>Get Memory Reliability</td>
<td>167</td>
</tr>
<tr>
<td>Get MULTOS Data</td>
<td>169</td>
</tr>
<tr>
<td>Get PIN Data</td>
<td>171</td>
</tr>
<tr>
<td>Get Process Event</td>
<td>173</td>
</tr>
<tr>
<td>Get Purse Type</td>
<td>174</td>
</tr>
<tr>
<td>Get Random Number</td>
<td>176</td>
</tr>
<tr>
<td>Get Replaced Application State</td>
<td>178</td>
</tr>
<tr>
<td>Get Session Size</td>
<td>179</td>
</tr>
<tr>
<td>Get Static Size</td>
<td>180</td>
</tr>
<tr>
<td>Get Transaction State</td>
<td>181</td>
</tr>
<tr>
<td>GSM Authenticate</td>
<td>182</td>
</tr>
<tr>
<td>Initialise PIN</td>
<td>184</td>
</tr>
</tbody>
</table>
APDU COMMANDS ................................................................. 300

Usage Notes ........................................................................ 300
CARD UNBLOCK ................................................................. 301
CHECK DATA ..................................................................... 302
CREATE MEL APPLICATION .............................................. 304
DELETE MEL APPLICATION .............................................. 306
FREEZE ............................................................................ 308
GET CONFIGURATION DATA .............................................. 309
GET DATA ......................................................................... 312
GET MANUFACTURER DATA ................................................ 314
GET MULTOS DATA ............................................................ 315
GET PURSE TYPE .............................................................. 317
GET RESPONSE .................................................................. 318
LOAD APPLICATION SIGNATURE ......................................... 320
LOAD CODE ........................................................................ 321
LOAD DATA ........................................................................ 322
LOAD DATA (Extended) ..................................................... 323
LOAD DIR FILE RECORD .................................................... 325
LOAD FCI RECORD ............................................................. 326
LOAD KTU CIPHERTEXT ..................................................... 327
OPEN MEL APPLICATION .................................................. 328
READ BINARY .................................................................... 331
READ RECORD(S) .............................................................. 333
SELECT FILE ..................................................................... 334
SET MSM CONTROLS ........................................................... 337

MULTOS STATUS CODES ..................................................... 338

INSTRUCTION MAP .......................................................... 340

PRIMITIVE SET LISTING .................................................... 346
Introduction
The MULTOS Developer’s Reference Manual is intended to be a concise presentation of the MULTOS low-level API and associated information. All of the instructions and primitives are defined without reference to any implementation. The MULTOS Implementation Report should be consulted for any specific implementation requirements.

MULTOS step/one
A new MULTOS specification product is now available, MULTOS step/one. This product is intended to provide issuers a low cost, high security, MULTOS compatible platform that can be used to deploy EMV applications using Static Data Authentication only. MULTOS step/one platforms support all instructions described in this document. However, for MULTOS step/one all primitives are considered to be optional. If implemented they will support the API described in this document. For further information regarding primitive support for MULTOS step/one, see www.multos.com.

Conventions and Assumptions
When reading this document there are some conventions and assumptions in place. They are:

- Hexadecimal numbers are indicated using a prefix of ‘0x’. For example, 0x16 is an hexadecimal value equal to 22 decimal.
- MULTOS is big-endian; i.e., the most significant byte is found at the lowest segment address and the least significant byte at the highest.
- Byte-blocks are always treated as unsigned.
- The stack operates on the principle of “last in, first out”.

This document attempts to avoid development tool specific syntax. If you wish to try the examples given, you may need to modify the code to work within your particular development environment.
Instructions

The following sub-sections define the instructions available to an application. In addition to the conventions and assumptions given in the introduction there are some additional points to take into account. They are:

- If a label can be used, but is not specified, then the instruction will execute using a value of the appropriate size found on the stack. For example, “ADDB , 5” adds five to the byte on top of the stack.
- A label can be a named memory location that the assembler will translate into an address or it can be an address. For example, “ADDB myVar, 5” adds five to the variable held at the named location myVar, while “ADDB PB[0], 5” adds five to the byte value found at the base of public memory.
**ADDB**

This instruction adds the literal byte to either the byte at the top of the stack or the byte held at the location specified by the label.

**Syntax**

```
ADDB [label], byte
```

**Remarks**

The result of the addition is written to either the byte at the top of the stack or, if specified, the label. If the result of the addition is greater than 255 the condition code register is updated as below and the value returned is truncated to one byte.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **C**: Set if result of the addition is greater than 255, cleared otherwise
- **V**: Unchanged
- **N**: Unchanged
- **Z**: Set if the result of the addition is zero, cleared otherwise

**Example**

The following line adds 32 to the tenth byte of static memory:

```
ADDB SB[9], 0x20
```

The following example adds 16 to the value held at a named location:

```c
// declare session variable
myNumber DYNAMIC BYTE
ADDB myNumber, 0x10
```

The next example adds 27 to the value placed on the top of the stack:

```
ADDB , 0x1B
```

The following examples show how the CCR is updated upon completion of the addition:

- 0xFF + 0x01 = 0x00; Carry is set and Zero is set
- 0xFF + 0x02 = 0x01; Carry is set and Zero is cleared
- 0x10 + 0x20 = 0x30; Carry and Zero are cleared
- 0x00 + 0x00 = 0x00; Carry is cleared and Zero is set
**ADDN**

This instruction adds the byte-block at the top of the stack to a byte-block specified by the label. If the label is omitted then the top two byte-blocks on the stack are used.

**Syntax**

```
ADDN [label], block_length
```

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the operands of size block_length will be taken from the stack.

The result of the addition will be written to the address corresponding to the label or, if no label is given, to the byte block immediately below the topmost block. In no case is the top byte block changed by the operation.

The operation will work if the two blocks overlap.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>
```

- **C**: Set if a carry occurs, cleared otherwise
- **V**: Unchanged
- **N**: Unchanged
- **Z**: Set if the result is zero, cleared otherwise.

**Example**

The following example adds the four bytes at the top of the stack to the four bytes at the base of the static area:

```
ADDN SB[0], 4
```

The following example is the same as the previous example, but uses a label to identify the variable instead of using a register/offset pair directly.

```
myVar STATIC BYTE 4
ADDN myVar, 4
```
The following example adds the four bytes at the top of the stack to the four bytes immediately below them on the stack.

```
//Stack = (bottom) 10,00,00,00,12,34,56,78 (top)
ADDN ,4
//Stack = (bottom) 22,34,56,78,12,34,56,78 (top)
```

The following example performs the addition of 0x00FF and 0x1001 on the stack and then adds the result, 0x1100, to a variable held in the static segment.

```
sResult STATIC Word = 0x1111
PUSHW 0x00FF       //Stack = 00,FF
PUSHW 0x1001       //Stack = 00,FF,10,01
ADDN ,2            //Stack = 11,00,10,01
POPW              //Stack = 11,00
ADDN sResult,2     //sResult now equals 0x2211
POPW              //Leave stack as found.
```

The following examples show how the CCR flags are set:

0xFFFFFFFF + 0x00000001 = 0x00000000; Carry is set and Zero is set
0xFFFFFFFF + 0x00000002 = 0x00000001; Carry is set and Zero is cleared
0x10000000 + 0x200000000 = 0x300000000; Carry and Zero are cleared
0x00000000 + 0x000000000 = 0x000000000; Carry is cleared and Zero is set
**ADDW**
This instruction adds the literal word to either the word at the top of the stack or the word held at the location specified by the label.

**Syntax**

```
ADDW [label], word
```

**Remarks**

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the operands will be taken from the stack.

The result of the addition will be written to the address corresponding to the label or, if no label is given, to the topmost word.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>
```

- **C**: Set if result is greater than 65535, cleared otherwise
- **V**: Unchanged
- **N**: Unchanged
- **Z**: Set if the result is zero, cleared otherwise

**Example**

The following line adds 0x2020 to the word at the bottom of the static segment:

```
ADDW SB[0000], 0x2020
```

The following line adds 0x1010 to the variable declared as myNum:

```
ADDW myNum, 0x1010
```

The following line adds 0x1010 to the current stack word.

```
ADDW , 0x1010
```

The following examples show how the CCR is set:

- 0xFF00 + 0x0100 = 0x0000; Carry is set and Zero is set
- 0xFF00 + 0x0200 = 0x0100; Carry is set and Zero is cleared
- 0x1000 + 0x2000 = 0x3000; Carry and Zero are cleared
- 0x0000 + 0x0000 = 0x0000; Carry is reset and Zero is set
**ANDN**

This instruction performs a bit-wise AND on a byte-block at the top of the stack with another byte-block specified by a label. If the label is omitted then the top two byte-blocks on the stack are used.

**Syntax**

```
ANDN [label], block_length
```

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the operands of size block_length will be taken from the stack.

The result of the AND will be written to the address corresponding to the label or, if no label is given, to the byte block immediately below the topmost block. In no case is the top byte block changed by the operation.

The operation will work if the two blocks overlap.

The Carry Flag is not affected by this instruction.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Set if the result is zero, cleared otherwise
Example

The following example performs a bit-wise AND operation between the four bytes at the top of the stack to the four bytes at the base of the static area. The result is written to SB[0].

```
ANDN SB[0],4
```

The following example is the same as the previous example, but uses a label to identify the variable instead of using a register/offset pair directly:

```
myVar STATIC BYTE 4
ANDN myVar,4
```

The following example performs a bit-wise AND operation between the four bytes at the top of the stack to the four bytes immediately below it on the stack.

```
//Stack = F0,F0,F0,F0,12,34,56,78
ANDN ,4
//Stack = 10,30,50,70,12,34,56,78
```

The following example pushes two blocks of four bytes onto the stack and uses them as operands in a bit-wise AND operation. To further illustrate the use of the instruction a two byte bit-wise AND is then performed with a static variable.

```
sResult STATIC WORD = 0xF0F0
PUSHW 0xFF00    //Stack = FF,00
PUSHW 0xFF00    //Stack = FF,00,FF,00
PUSHW 0x1234    //Stack = FF,00,FF,00,12,34
PUSHW 0x5678    //Stack = FF,00,FF,00,12,34,56,78
ANDN ,4        //Stack = 12,00,56,00,12,34,56,78
// the operation is: F0F0 AND 5678
ANDN sResult,2 //sResult = 0x5070
```

The following examples show how the CCR is set:

0xFF00 & 0x00FF = 0x0000; Zero is set
0xFF0F & 0x00FF = 0x00F0; Zero is cleared
**BRANCH**

The branch instruction is used to move the code pointer to a location in the application relative to the current location. The branch may be made conditional on the current values of the condition register.

**Syntax**

```
BRA offset //Branch always
BEQ offset //Branch if Equal
BLE offset //Branch if Less Than or Equal
BLT offset //Branch if Less Than
BGT offset //Branch if Greater Than
BGE offset //Branch if Greater Than or Equal
BNE offset //Branch if Not Equal
```

**Remarks**

The offset value refers to a location relative to the current instruction within the application’s code space. It can be expressed as a fixed numeric value or as a named label within the application’s source code. The latter case relies on the assembler to calculate the appropriate relative offset.

The BRANCH instruction has a range of -128 to 127 bytes inclusive; i.e., the branch can refer to a location 128 bytes prior to the current location of the code pointer or 127 bytes ahead of that location. If the branch destination lies outside of this range then a JUMP instruction should be used. Note that if the destination is within the given range, a branch is preferred because of the reduced code size produced.

The Code Pointer will point to the next instruction to execute and therefore a branch to relative address zero will have no effect, whilst a branch of -2 will branch back to the branch instruction. However, in most applications the value of the relative jump will be calculated automatically by the assembler.

The following table shows the condition code flags which are checked for each of the conditional branch instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition - description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>Z – CCR Z flag set</td>
</tr>
<tr>
<td>BLT</td>
<td>C – CCR C flag set</td>
</tr>
<tr>
<td>BLE</td>
<td>C or Z – either CCR C or CCR Z flag set</td>
</tr>
<tr>
<td>BGT</td>
<td>!(C or Z) – neither CCR C nor CCR Z flag set</td>
</tr>
<tr>
<td>BGE</td>
<td>!C – CCR C flag not set</td>
</tr>
<tr>
<td>BNE</td>
<td>!Z – CCR Z flag not set</td>
</tr>
<tr>
<td>BRA</td>
<td>always</td>
</tr>
</tbody>
</table>

The conditions based on the state of the carry flag (LT, LE, GT, and GE) are determined using unsigned comparisons. MULTOS does not interrogate the V or N flags and, therefore, does not directly support conditional program flow based on signed comparisons. If required, the application developer could implement such a flow by interrogating the N and V flags in the application source code. These values are made available using the primitive ‘Load CCR’. Note that the underlying platform may not set the N or V flags.
**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

The following example is a fragment of code that uses a conditional branch instructions to implement a loop.

```c
loopCounter DYNAMIC BYTE

   // put zero value on stack
   PUSHZ  1
   // store it to counter
   STORE  loopCounter, 1

labelStartofLoop

   // carry out processing
   // increment loop counter
   INCN  loopCounter, 1
   // compare counter to literal maximum loop value
   CMPB  loopCounter, 5
   // if loop counter < 5 go around again
   BLT  labelStartofLoop
```

The next example illustrates the use of BRANCH when handling the instruction byte of an APDU command.

```c
pINS   EQU   PT[-12]
    CMPB  pINS, 0xA4
    BEQ   cmdSelectFile
    CMPB  pINS, 0x10
    BEQ   cmdINS10
    CMPB  pINS, 0x20
    BEQ   cmdINS20

errNoINS
    EXITSW 0x6D,0x00
```
**CALL**

This instruction is used to call a function.

**Syntax**

```plaintext
CALL [function]   //Call always
CEQ  function     //Call if Equal
CLE  function     //Call if Less Than/Equal
CLT  function     //Call if Less Than
CGT  function     //Call if Greater Than
CGE  function     //Call if Greater Than/Equal
CNE  function     //Call if Not Equal
```

**Remarks**

The following table shows the condition code flags which are checked for each of the conditional call instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition - description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEQ</td>
<td>Z – CCR Z flag set</td>
</tr>
<tr>
<td>CLT</td>
<td>C – CCR C flag set</td>
</tr>
<tr>
<td>CLE</td>
<td>C or Z – either CCR C or CCR Z flag set</td>
</tr>
<tr>
<td>CGT</td>
<td>!(C or Z) – neither CCR C nor CCR Z flag set</td>
</tr>
<tr>
<td>CGE</td>
<td>!C – CCR C flag not set</td>
</tr>
<tr>
<td>CNE</td>
<td>!Z – CCR Z flag not set</td>
</tr>
<tr>
<td>CALL</td>
<td>always</td>
</tr>
</tbody>
</table>

The conditions based on the state of the carry flag (LT, LE, GT, and GE) are determined using unsigned comparisons. MULTOS does not interrogate the V or N flags and, therefore, does not directly support conditional program flow based on signed comparisons. If required, the application developer could implement such a flow by interrogating the N and V flags in the application source code. These values are made available using the primitive 'Load CCR'. Note that the underlying platform may not set the N or V flags.

The CALL instruction is used to call a function. This would be written in assembler as a label and in MEL as a code address. It is also possible to omit the function to call from the instruction. In this case the code address to call is taken as the top two bytes of the stack, which the instruction will pop from the stack. Furthermore, the top two bytes of the stack must be a valid code address and the call must not be conditional.

Prior to the execution of the called function this instruction pushes four bytes of data on to the stack, the current local base register followed by the current code pointer register. The current code pointer register will point to the next instruction after the call, i.e., the location where execution will resume once the function returns. The value of the Local Base register is set to match the new Dynamic Top.

Provided that the called function has not changed the default value of the previous code pointer address execution continues from the instruction directly after the CALL instruction. Otherwise, execution resumes at the code address given.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C: Unchanged  
V: Unchanged  
N: Unchanged  
Z: Unchanged

**Example**

The following example shows a simple function call. A function called fnBiggest accepts two words as input parameters and returns the largest of the two words. After the CALL instruction the value of 0x1000 will be left on the stack.

```assembly
start
    PUSHW 0x0100
    PUSHW 0x1000
    CALL fnBiggest
    EXIT

//================================
// fnBiggest
//================================

// Input Param = wValue1, wValue2
// Output Param = wBiggest
//
// WBiggest is the larger of the two input words
//================================
// lWord1  IN  WORD
// lWord2  IN  WORD
// The negative address relative to this function’s
// Lower Base include the 4 bytes of data that the
// call instruction places on the stack
lWord1 EQU LB[-8]
lWord2 EQU LB[-6]
    LOAD lWord1,2
    LOAD lWord2,2
    CMPN ,2
    BLT fnBiggest_leave
    POPW
fnBiggest_leave
    RET 4,2
```

**CLEARN**

This instruction sets a byte-block to zero.

**Syntax**

`CLEARN [label],block_length`
Remarks

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the data of size block_length on the stack will be cleared.

The Condition Code Register is not affected by this instruction.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Unchanged

Example

The following example clears the first 255 bytes of the public area and the top ten bytes of the stack.

```
pBase EQU PB[0000]
CLEARN pBase, 0xFF
CLEARN , 0x0A
```
**CMPB**

This instruction compares a literal byte with either the byte at the top of the stack or the byte held at the location specified by the label.

**Syntax**

```
CMPB [label], byte
```

**Remarks**

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the instruction will compare the literal byte with the byte value at the top of the stack.

The comparison is performed by subtracting the literal byte from the value given. The result is discarded but the condition code register is updated. Note that the byte values are treated as unsigned values.

The result of the comparison is held in the Condition Code Register based on the criteria given in the table below.

<table>
<thead>
<tr>
<th>Carry</th>
<th>Zero</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Target Byte &gt; Literal Byte</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Target Byte = Literal Byte</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Target Byte &lt; Literal Byte</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

**Condition Code**

```
C V N Z
- - - - X - - X
```

- C: See table above
- V: Unchanged
- N: Unchanged
- Z: See table above

**Example**

The following example compares the byte held at PT[-12], to the literal value 0x90.

```
CMPB   PT[-12], 0x90
```

The following example is the same as the above example but uses a label to define the byte’s location.

```
pINS EQU PT[-12]
CMPB   pINS, 0x90
```

The following example compares the byte held at the top of the stack to the literal byte 0x90.

```
CMPB   ,0x90
```
The following example compares the class byte of the current APDU and jumps if it does not match what is expected.

```
pCLA EQU PT[-13]
CMPB pCLA, 0x90
JNE errWrongClass
//continue processing
errWrongClass
  //insert error code
EXITSW 0x6E,0x00
```
**CMPN**

This instruction compares a byte-block of size n with another of the same size.

**Syntax**

```
CMPN [label], block_length
```

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the two blocks of data of size block_length on the stack will be compared.

The result of the comparison is held in the Condition Code Register based on the criteria given in the table below.

```
Carry  | Zero | Description
------ | ---- | ------------------------------------------
  0    |  0   | Labelled Byte-block > Stack Top Byte-block
  0    |  1   | Labelled Byte-block = Stack Top Byte-block
  1    |  0   | Labelled Byte-block < Stack Top Byte-block
  1    |  1   | Not possible
```

The byte-block occupying the top of the stack is used as the basis of the comparison. The labelled byte-block is the block of size block_length found at the location given. If no label is given, then labelled byte-block is the data of size block_length below the block occupying the top of the stack.

The operation will work if the two blocks overlap.

**Condition Code**

```
```

```
C V N Z
- - X - X
```

C  See table above  
V  Unchanged  
N  Unchanged  
Z  See table above
Example

The following example compares the four bytes at the top of the stack to the four bytes at the base of public.

    CMPN PB[0], 4

The following example is the same as the previous example, but uses a label to identify the four bytes of public.

    pPIN EQU PB[0]
    CMPN pPIN, 4

The following examples compare the four bytes at the top of the stack with the four bytes immediately below them on the stack.

```
PUSHW 0x1234
PUSHW 0x5678
PUSHW 0x1234
PUSHW 0x5678
CMPN   , 4
POPN   8             // clean up stack
BEQ    someLabel     // conditional branch will fire
PUSHW 0x1234
PUSHW 0x5678
PUSHW 0x1234
PUSHW 0x6789
CMPN   , 4
POPN   8             // clean up stack
BEQ    someLabel     // conditional branch will not fire
```

The following example compares a block of bytes held at the base of public to a byte-block held in static memory. This could typically be used to perform PIN verification.

```
sPIN STATIC BYTE 04 = 1,2,3,4
pPIN EQU PB[0]
    LOAD  pPIN, 4
    CMPN  sPIN, 4
    BNE   PinDoesNotMatch
        //Pin Matches
        //Insert code to flag a valid pin
    EXITSW  0x90, 0x00
PinDoesNotMatch
    EXITSW  0x64, 0x00
```
**CMPW**

This instruction compares a word value against a word literal.

**Syntax**

```plaintext
CMPW [label], word
```

**Remarks**

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the word on top of the stack will be compared to the literal word.

The result of the comparison is held in the Condition Code Register based on the criteria given in the table below.

<table>
<thead>
<tr>
<th>Carry</th>
<th>Zero</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Target Word &gt; Literal Word</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Target Word = Literal Word</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Target Word &lt; Literal Word</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

**Condition Code**

```
C  V  N  Z
--- --- --- ---
X  X  X  X
```

- C  See table above
- V  Unchanged
- N  Unchanged
- Z  See table above
**Example**

The following example compares the literal word 0x0000 with the word whose starting address is eleven bytes from the top of the public area, PT[-11].

```
CMPW   PT[-11], 0x0000
```

The following example is the same as the previous example except that a label is used to identify the start address, which corresponds to the P1 and P2 parameter bytes.

```
pP1P2 EQU PT[-11]
CMPW pP1P2, 0x0000
```

The following example compares the word at the top of the stack with the literal word 0x000.

```
CMPW   , 0x0000
```

The following example compares the parameter bytes of the current APDU and jumps if they do not match what is expected.

```
pP1P2 EQU PT[-11]

CMPW   pP1P2, 0x01FF
JNE    errWrongParameters
// if equal processing continues

errWrongParameters
//insert error code
EXIT
```
**DECN**
This instruction performs a block decrement; i.e., it subtracts one from the value of a byte-block.

**Syntax**

DECN [label], block_length

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register/offset pair, or an explicit register/offset pair. If a label is not specified, then the data of size block_length on the stack will be decremented.

The CCR zero flag is updated by this instruction. For example,

DECN (0x000001) = 0x00000000; Zero is set
DECN (0x000011) = 0x00000010; Zero is reset

However, DECN does not modify the carry flag of the condition register. Decrementing a zero value by one results in 0xFFFF...FF. This does not set zero flag.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the result is zero, cleared otherwise

**Example**

The following example decrements the four byte-block at the base of the static area by one.

DECN  SB[0],4

The following example is the same as the previous example except that it uses a label to identify the variable to be decremented.

sVar  STATIC BYTE 4
DECN sVar,4
**INCN**
This instruction performs a block increment; i.e., it adds one to the value of a byte-block.

**Syntax**

\[ \text{INCN} \ [\text{label}], \text{block\_length} \]

**Remarks**

The block\_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the data of size block\_length on the stack will be incremented.

The CCR zero flag is updated by this instruction. For example,

\[
\text{INCN} \ (0x000001) = 0x00000000; \text{Zero is set}
\]
\[
\text{INCN} \ (0x000011) = 0x00000010; \text{Zero is reset}
\]

However, INCN does not modify the carry flag of the condition register.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C \hspace{1em} Unchanged
V \hspace{1em} Unchanged
N \hspace{1em} Unchanged
Z \hspace{1em} Set if the result is zero, cleared otherwise

**Example**

The following example increments the four byte-block at the base of the static area by one.

\[
\text{INCN} \ SB[0], 4
\]

The following example is the same as the previous example except that it uses a label to identify the variable to be incremented

\[
\text{sVar} \hspace{1em} \text{STATIC BYTE 4}
\]
\[
\text{INCN} \ sVar, 4
\]

The following example decrements the word stored at the top of the stack by one.

\[
\text{INCN} \ ,2
\]

**INDEX**

This instruction calculates the address of a record within an array of fixed length records.
**Syntax**

INDEX label, record_length

**Remarks**

This instruction also uses the top byte of the stack to indicate which record index is required. As this value is held in a single byte the maximum number of records is 256. Note also that the array index value uses zero based counting; e.g., the first record is at offset 0.

The record_length value is specified using a single byte. Therefore, the maximum length of a record is 255 bytes.

The label, which must be present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair.

The result of the index instruction is a two byte value, which indicates the starting address of the record requested. Note, however, there is no requirement for the resulting address be valid. This instruction will calculate a two byte value based solely on the values passed to it.

The Index instruction performs the following calculation:

\[
result = \text{address(label)} + (\text{record_length} \times \text{record_indicator})
\]

where, address(label) is the segment address of the label or register/offset pair, record_length is a literal byte representing the record length, and record_indicator is the top byte of the stack.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- C: Set if a carry occurs, cleared otherwise
- V: Unchanged
- N: Unchanged
- Z: Set if the result is zero, cleared otherwise
Example

The following extended example

```c
// sByteArray assumed to start at Static Bottom;
// i.e., start address is SB[0]
sByteArray STATIC BYTE = 0xFF, 0xEF, 0xDF, 0xCF

// sWord Array assumed to follow previous array;
// i.e., start address is SB[4]
sWordArray STATIC WORD = 0x1010, 0x0101, 0xA5A5, 0x5A5A

// Get address of 3rd byte of sByteArray
PUSHB   2
INDEX   sByteArray, 1

// calculation would be result = 0 + (1 * 2) = 2.
// Therefore address of third byte is SB[2]

POPN    3 // clean stack of record_index and result

// Get 4th word of sWordArray
PUSHB   3
INDEX   sWordArray, 2

// calculation would be result = 4 + (2 * 3) = 10
// Therefore starting address of third word is SB[10]
```

The following example calculates the address of the twelfth record of an array of records, then copies this onto the stack. Unlike the previous example no assumption regarding the starting address of the array is made.

```c
recNumber EQU 16
recLength EQU 32
sArray    STATIC BYTE recNumber*recLength

PUSHB   11 //the twelfth record
INDEX   sArray, recLength
LOADI   ,recLength
```
JUMP

This instruction causes execution to continue from a different location in the application’s code space. The jump may be made conditional on the current values of the condition register.

Syntax

- JMP [label] //Jump always
- JEQ label //Jump if Equal
- JLE label //Jump if Less Than/Equal
- JLT label //Jump if Less Than
- JGT label //Jump if Greater Than
- JGE label //Jump if Greater Than/Equal
- JNE label //Jump if Not Equal

Remarks

This instruction differs from BRANCH in that the specified instruction is absolute from the start of the code rather than relative to the current instruction. The resulting machine code for JUMP is also one byte larger than that for BRANCH.

The label can be expressed as a fixed numeric value or as a named label within the application’s source code. The latter case relies on the assembler to calculate the appropriate offset.

Similar to CALL, it is possible to omit the label from the instruction. In this case the code address to which to jump is taken to be the top two bytes of the stack, which the instruction will pop from the stack. Furthermore, the top two bytes of the stack must be a valid code address and the jump must not be conditional.

The following table shows the condition code flags which are checked for each of the conditional branch instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition - description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEQ</td>
<td>Z – CCR Z flag set</td>
</tr>
<tr>
<td>JLT</td>
<td>C – CCR C flag set</td>
</tr>
<tr>
<td>JLE</td>
<td>C or Z – either CCR C or CCR Z flag set</td>
</tr>
<tr>
<td>JGT</td>
<td>!(C or Z) – neither CCR C nor CCR Z flag set</td>
</tr>
<tr>
<td>JGE</td>
<td>!(C or Z) – neither CCR C nor CCR Z flag set</td>
</tr>
<tr>
<td>JNE</td>
<td>!Z – CCR Z flag not set</td>
</tr>
<tr>
<td>JRA</td>
<td>always</td>
</tr>
</tbody>
</table>
The conditions based on the state of the carry flag (LT, LE, GT, and GE) are determined using unsigned comparisons. MULTOS does not interrogate the V or N flags and, therefore, does not directly support conditional program flow based on signed comparisons. If required, the application developer could implement such a flow by interrogating the N and V flags in the application source code. These values are made available using the primitive ‘Load CCR’. Note that the underlying platform may not set the N or V flags.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

In this example the labels ‘SetID’ and ‘QueryID’ are the designations of two functions. To execute a function, the proper APDU command instruction byte must be sent.

```assembly
pINS EQU PT[-12]

LOAD pINS,1

// if pINS = 0x10 then goto SetID
CMPB ,0x10
JEQ SetID

// if pINS = 0x20 then goto QueryID
CMPB ,0x20
JEQ QueryID

// if neither if statement applies
// then exit and return 6D00
UnrecognisedInstruction
EXITSW 0x6D,0x00

SetID
  //SetID command processing
  EXIT

QueryID
  //QueryID command processing
  EXIT
```

**LOAD**

This instruction pushes a byte-block onto the stack from either the current top of the stack or a location specified by the label.
Syntax

LOAD [label], block_length

Remarks

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

If the label is omitted then the byte-block at the top of the stack is pushed; i.e., the top of the stack is duplicated.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Example

The following example pushes the eight bytes held at the base of public onto the stack.

LOAD  PB[0],8

The following example is the same as the previous example except that it uses a label to identify the location of the eight bytes at the base of public

pKey  EQU PB[0]
LOAD  pKey,8

The following example pushes the top four bytes of the stack back onto the stack; that is it duplicates the top four bytes of the stack.

LOAD  ,4

The following example loads a four byte number from the bottom of the public segment onto the stack, doubles it by adding the value to itself, and leaves the result on the stack.

LOAD  PB[0000],4
LOAD  ,4
ADDN  ,4
POPN  4

LOADA

This instruction pushes the address of a variable or register/offset onto the stack.

Syntax

LOADA  label
**Remarks**

The label may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair.

There is no requirement that the address is a valid address.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C: Unchanged  
- V: Unchanged  
- N: Unchanged  
- Z: Unchanged

**Example**

The following example pushes the address of the top byte of static onto the stack.

```
LOADA ST[-1]
```

The following example pushes the address of a variable onto the sack.

```
sMyVar STATIC WORD
LOADA sMyVar
```
**LOADI**

This instruction pushes a block of bytes to the stack using indirect addressing.

**Syntax**

```plaintext
LOADI  [label],length
```

**Remarks**

If the label is given then the two byte address held at the label are used as the address of the byte-block to push onto the stack. If the label is omitted then the two bytes on the top of the stack are used as the address of the byte-block to push onto the stack.

The bytes stored at the label are not loaded. They are interpreted as the address of the byte-block to push onto the stack.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Unchanged

**Example**

The following example pushes a variable onto the stack using indirect addressing. The variable `sAddrVar` is used to hold the address of the variable to push onto the stack. In this case the variable is assumed to be eight bytes long.

```plaintext
sAddrVar STATIC WORD
LOADI sAddrVar,8
```

The following example pushes a block of bytes onto the stack using the top two bytes of the stack as an address.

```plaintext
LOADI ,2
```
The following example calculates the address of the twelfth record of an array of records and then copies this record onto the stack. The Index instruction is used to calculate the address of the record and leave this on the stack; the LOADI instruction is used to push the record indirectly using the address on the top of the stack.

```assembly
recNumber EQU 16
recLength EQU 32
sArray    STATIC BYTE recNumber*recLength
  PUSHB   11  //the twelfth record
  INDEX   sArray, recLength
  LOADI   ,recLength
```

The following example uses a variable, recAddr, to hold the address of the current record which is then moved to the base of public.

```assembly
recAddr  DYNAMIC BYTE 2
  // Put address of sArray on stack
  LOADA   sArray
  // Move value to recAddr
  STORE   recAddr, 2
  // Copy current record to public
  LOADI   recAddr, recLength
  STORE   PB[0], recLength
```
**NOTN**

This instruction performs a bit-wise NOT on a byte-block.

**Syntax**

```
NOTN [label],block_length
```

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair that gives the address of the block to be inverted bit-wise. If a label is not specified, then the data of size block_length on the stack will be inverted.

The result is written to the label or the byte-block on the stack.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Unchanged</td>
<td>Unchanged</td>
<td>Unchanged</td>
<td>Set if the result is zero, cleared otherwise</td>
</tr>
</tbody>
</table>

**Example**

The following example performs a bit-wise NOT operation on the first two bytes of the static area.

```
NOTN SB[0],2
```

The following example is the same as the previous example except that it uses a label to identify the bytes to perform the bit-wise NOT operation on.

```
myVar STATIC WORD
NOTN myVar,2
```

The following example performs a bit-wise NOT on the top eight bytes of the stack.

```
NOTN ,8
```
**ORN**

This instruction performs a bit-wise OR on a byte-block at the top of the stack with another byte-block specified by a label. If the label is omitted then the top two byte-blocks on the stack are used.

**Syntax**

\[
\text{ORN} \ [\text{label}], \text{block\_length}
\]

**Remarks**

The block\_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the data of size block\_length on the top of the stack and the byte block of size block\_length below it on the stack will be the OR operands.

The result of the OR operation is written to the lower byte-block on the stack, or if specified, the byte-block held at the label. The byte-block at the top of the stack is not changed by this instruction.

The result of the operation updates the zero flag in the Condition Code Register. For example,

- \(0x0000 \text{ OR } 0x0000 = 0x0000\); Zero is set
- \(0xF0F0 \text{ OR } 0xFF = 0xF0FF\); Zero is reset

The Carry Flag is not affected by this instruction.

The operation will work if the two blocks overlap.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C: Unchanged
- V: Unchanged
- N: Unchanged
- Z: Set if the result is zero, cleared otherwise
**Example**

The following example performs a bit-wise OR operation on the first two bytes of the static area.

```
ORN SB[0],2
```

The following example is similar to the previous example except that it uses a label to identify the bytes to perform the bit-wise OR operation on.

```
myVar STATIC WORD
ORN myVar,2
```

The following example uses two 8-byte blocks on the stack as operands.

```
ORN ,8
```

The following example the top stack byte and the byte below it on the stack are the OR operands. Note that the result is written to the lower stack byte.

```
PUSHB 0x10     // Stack = 10
PUSHB 0x02     // Stack = 10,02
ORN , 1      // Stack = 12,02
```
**PRIMRET**

This instruction is used to call a primitive or return from a function call.

**Syntax**

```
PRIM primitive [,byte1 [,byte2 [,byte3]]]
RET [ [inBytes] [,outBytes]]
```

**Remarks**

This instruction performs two different operations depending upon the syntax used:

- The PRIM mnemonic is used to call a primitive with up to three bytes of arguments.
- The RET mnemonic is used to return from a function. The inBytes and outBytes are used to specify the number of bytes used by input parameters and the number of bytes which are to be returned as result bytes respectively. These are used by MULTOS to clean up the stack following the function’s return.

Both the inBytes and outBytes values are specified using a single byte. Therefore, the maximum number of parameter bytes or returned bytes is 255 bytes.

After returning from a function the stack will be cleaned up. The upper part of the following diagram shows the state of the stack while a function is executing. The lower portion illustrates the stack after the RET instruction executes.

![Stack diagram](image)

The size of the input parameters is given in inBytes while the size of the result bytes is given in outBytes. For details on the control bytes see the CALL instruction. Any stack used by the function, shown as Function Stack in the above diagram, is removed along with any local variables declared by...
the function. Local variables are shown as Function Locals in the above diagram. The overall effect of this is to remove the input parameters, return the output bytes and restore the LB and CP registers.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

Although this instruction does not change any of the condition code flags the primitive called may itself have an effect on the flags.
**Example**

The following examples illustrate the use of the mnemonic PRIM. See the primitives section of this document for explanations of the primitives used.

```
// Call CheckCase for ISO APDU Command Case3
PUSHB 3
PRIM 0x01

// Call MultiplyN to multiply two 2-byte values
PUSHW 6
PUSHW 1289
PRIM 0x10, 2

// Call Shift Right - shift value placed on stack 2 bits to the left
BLOCKSIZE EQU 8
MULTIPLYBYFOUR EQU 2

    LOAD PB[0], BLOCKSIZE
    PRIM 0x03, BLOCKSIZE, MULTIPLYBYFOUR
```

The following examples illustrate the uses of the RET mnemonic.

```
// Return from function with no Input or Output
RET
// Return from function with two Input and no Output bytes
RET 2
// Return from function with no Input and three Output bytes
RET , 3
// Return from function with two Input and three Output bytes
RET 2, 3
```
**SETB**

This instruction copies the literal byte to either the byte at the top of the stack or the byte held at the location specified by the label.

**Syntax**

```
SETB [label], byte
```

**Remarks**

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the byte set will be that on top of the stack.

The SETB instruction overwrites the top stack byte, it does not push a value onto the stack. So if there is no byte on the stack, the AAM will abend if the SETB instruction is executed.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

C Unchanged
V Unchanged
N Unchanged
Z Unchanged

**Example**

The following example sets the byte at the top of the stack to 10

```
SETB ,10
```

The following example sets the byte held at pSW2 to 0

```
pSW2 EQU PT[-1]
SETB pSW2,0
```
**SETW**

This instruction copies the literal word to either the word at the top of the stack or the word held at the location specified by the label.

**Syntax**

```
SETW [label], word
```

**Remarks**

If a label is given then the word stored at the label is set to the literal word. The assembler will translate this into the corresponding register/offset pair during assembly, or alternatively the register/offset pair may be given explicitly.

If the label is omitted then the literal word is copied to the byte at the top of the stack.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged

**Example**

The following example sets the word held at PT[-4] to 16.

```
SETW   PT[-4], 16
```

The following example is the same as the previous example, but as PT[-4] is the location of the La value it uses a label to identify the variable pLa instead of using a register/offset pair directly.

```
pLa    EQU PT[-4] 
SETW   pLa, 16
```

The following example sets the word at the top of the stack.

```
SETW   , 0xFFFF
```
STACK
This instruction allows bytes or words to be pushed onto and popped from the stack.

Syntax

```
PUSHZ block_length   // Pushs block of zeros onto the stack
PUSHB byte           // Pushes a byte onto the stack
PUSHW word           // Pushes a word onto the stack
POPN  block_length   // Pops block of bytes from the stack
POPB                 // Pops a byte from the stack
POPW                 // Pops a word from the stack
```

Remarks

The action performed by this instruction depends upon the mnemonic used. There are six operations:

- Push Zero (PUSHZ): A block of zeros is pushed onto the stack. The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.
- Push Byte (PUSHB): A literal byte is pushed onto the stack
- Push Word (PUSHW): A literal word is pushed onto the stack
- Pop Byte-block (POPN): A block of bytes is popped from the stack. The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.
- Pop Byte (POPB): A single byte is popped from the stack.
- Pop Word (POPW): A single word, two bytes, is popped from the stack.

If an attempt is made to pop more bytes off the stack than are present on the stack then the MULTOS device will abend. Likewise, if there is insufficient space in dynamic memory to hold any bytes pushed onto the stack then the MULTOS device will also abend.

Condition Code

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>C</th>
<th>Unchanged</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Unchanged</td>
</tr>
<tr>
<td>N</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Z</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>
Example

The following example uses the different mnemonics to manipulate the stack. The stack is empty at the start.

```
PUSHZ 3 //Stack = 00,00,00
PUSHB 10 //Stack = 00,00,00,0A
PUSHW 0x1234 //Stack = 00,00,00,0A,12,34
POPN 3 //Stack = 00,00,00
POPB //Stack = 00,00
POPW //Stack = EMPTY
```

STORE

This instruction moves a block of bytes from the stack to a given location.

Syntax

```
STORE [label], block_length
```

Remarks

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the data of size block_length on the stack will be moved to the byte block below that on the top of the stack.

This instruction will pop the byte-block from the stack; i.e., the operation is a move and not a copy.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Unchanged

Example

The following example copies the first eight bytes of the public area to the first eight bytes of the static area.

```
LOAD    PB[0], 8
STORE   SB[0], 8
```

The next example illustrates how the instruction functions when the label is not specified.

```
PUSHW 0x0000 // Stack = 00,00
PUSHW 0x1234 // Stack = 00,00,12,34
```
STORE , 2 // Stack = 12,34
**STOREI**

This instruction moves a block of bytes from the stack to a given location using indirect addressing.

**Syntax**

```
STOREI [label], length
```

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the destination address will be taken to be the two bytes below the block of size block_length on the top of the stack. In other words, the destination address must be placed onto the stack followed by the bytes to move.

This instruction will pop the byte-block from the stack; i.e., the operation is a move and not a copy.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

C: Unchanged
V: Unchanged
N: Unchanged
Z: Unchanged
**Example**

The following example copies the bytes 0x12 and 0x34 to the base of static, which has an address of 0.

```
LOADA SB[0]    // Stack = 00,00
PUSHW 0x1234   // Stack = 00,00,12,34
STOREI , 2     // Stack = 00,00
```

The following example copies the word 0x1234 to the memory location in sValue as calculated using the INDEX instruction.

```
dAddr DYNAMIC WORD
sValue STATIC WORD = 0xABCD, 0x4567, 0x0000, 0xEF89

// use INDEX to get address of 3rd word in sValue array
PUSHB 2
INDEX sValue, 2
// move address to session variable
STORE dAddr, 2
// remove pushed byte
POPB
// now push literal word and store at calculated address
PUSHW 0x1234   // Stack = 12,34
STOREI sAddr,2 // Stack = {empty}
```
**SUBB**

This instruction subtracts the literal byte from either the byte at the top of the stack or the byte held at the location specified by the label.

**Syntax**

```
SUBB [label], byte
```

**Remarks**

The result of the operation is written to either the byte at the top of the stack or, if specified, the label.

If a label is specified then the literal byte is subtracted from the byte held at the label. The assembler will translate the label into the corresponding register/offset pair during assembly, or alternatively the register/offset pair may be given explicitly. If the label is omitted then the literal byte is subtracted from the byte at the top of the stack.

The result of the subtraction updates the Condition Code Register. The carry flag is set if the result of the operation would be less than zero, while the zero flag is set if the result of the operation is equal to zero. For example,

- \(0x10 - 0x20 = 0xF0\); Carry is set and Zero is reset
- \(0x10 - 0x10 = 0x00\); Carry is reset and Zero is set
- \(0x20 - 0x10 = 0x10\); Carry and Zero are reset

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- C  Set if a borrow occurs, cleared otherwise
- V  Unchanged
- N  Unchanged
- Z  Set if the result is zero, cleared otherwise

**Example**

The following line subtracts 32 from the tenth byte of the Static Area.

```
SUBB  SB[9] ,32
```

The following example is similar to the previous example, but uses a label to identify the variable instead of using a register/offset pair directly.

```
myNum1 STATIC BYTE
SUBB  myNum1, 32
```

**SUBN**

This instruction subtracts the byte-block at the top of the stack from a byte-block specified by the label. If the label is omitted then the top two byte-blocks on the stack are used.
### Syntax

```
SUBN [label], block_length
```

### Remarks

The `block_length` value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then both operands of size `block_length` will be taken from the stack.

The result of the subtraction is written to the lower byte-block on the stack, or if specified, the byte-block held at the label. The byte-block at the top of the stack is not changed by this instruction.

The subtraction of the byte-blocks is performed as though the entire byte-block represents a single unsigned number and bits may be carried over from the least significant bytes to the most significant bytes.

The result of the addition updates the Condition Code Register. The carry flag is set if the result of the operation would be less than zero, while the zero flag is set if the result of the operation is equal to zero. For example,

- \(0x20000000 - 0x20000000 = 0x00000000\); Carry is reset and Zero is set
- \(0x20000000 - 0x40000000 = 0xE0000000\); Carry is set and Zero is reset
- \(0x20000000 - 0x100000000 = 0x10000000\); Carry and Zero are reset

The operation will work if the two blocks overlap.

### Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- **C**: Set if a borrow occurs, cleared otherwise
- **V**: Unchanged
- **N**: Unchanged
- **Z**: Set if the result is zero, cleared otherwise
Example

The following example subtracts the four bytes at the top of the stack from the four bytes at the base of the static area.

```
SUBN SB[0],4
```

The following example is similar to the previous example, but uses a label to identify the variable instead of using a register/offset pair directly.

```
myVar STATIC BYTE 4
SUBN myVar,4
```

The following example subtracts the four bytes at the top of the stack from the four bytes immediately below them on the stack.

```
//Stack = BB,BB,BB,BB,11,11,11,11
SUBN ,4
//Stack = AA,AA,AA,AA,11,11,11,11
```

The following example performs a subtraction on the stack and then subtracts the result from a variable held in the static segment.

```
sResult STATIC Word = 0x3000
PUSHW 0x00FF        //Stack = 21,00,10,01
PUSHW 0x1001        //Stack = 21,00,10,01
SUBN , 2            //Stack = 10,FF,10,01
POPW                  //Stack = 10,FF
SUBN sResult,2      //sResult now equals 0x1F01
POPW                  //Leave stack as found.
```
**SUBW**

This instruction subtracts the literal word from either the word at the top of the stack or the word held at the location specified by the label.

**Syntax**

```plaintext
SUBW [label], word
```

**Remarks**

The result of the operation is written to either the word at the top of the stack or, if specified, the label.

If a label is specified then the literal word is subtracted from the word held at the label. The assembler will translate the label into the corresponding register/offset pair during assembly, or alternatively the register/offset pair may be given explicitly. If the label is omitted then the literal word is subtracted from the word at the top of the stack.

The result of the subtraction updates the Condition Code Register. The carry flag is set if the result of the operation is less than zero and the zero flag is set if the result of the operation is equal to zero. For example,

- $0x1000 - 0x2000 = 0xF000$; Carry is set and Zero is reset
- $0x1000 - 0x1000 = 0x0000$; Carry is reset and Zero is set
- $0x2000 - 0x1000 = 0x1000$; Carry and Zero are reset

**Condition Code**

```
C  V  N  Z
-  -  -  X  -  -  X
```

- **C** Set if a borrow occurs, cleared otherwise
- **V** Unchanged
- **N** Unchanged
- **Z** Set if the result is zero, cleared otherwise

**Example**

The following line subtracts $0x2020$ from the word at the bottom of the Static Segment.

```plaintext
SUBB SB[0], 0x2020
```

The following example is similar to the previous example, but uses a label to identify the variable instead of using a register/offset pair directly.

```plaintext
myNum1 STATIC WORD
SUBW myNum1, 0x2020
```

**SYSTEM**

With the exception of NOP, system instructions perform an operation relating to setting the response that the application will return to the IFD and exiting an application.
Syntax

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>SETSW</td>
<td>Set the status word</td>
</tr>
<tr>
<td>SETLA</td>
<td>Set the length of response data returned</td>
</tr>
<tr>
<td>SETSWLA</td>
<td>Set the status word and length of response data returned</td>
</tr>
<tr>
<td>EXIT</td>
<td>Exit from the application</td>
</tr>
<tr>
<td>EXITSW</td>
<td>Set the status word and exit from the application</td>
</tr>
<tr>
<td>EXITLA</td>
<td>Set the length of response data returned and exits from the application</td>
</tr>
<tr>
<td>EXITSWLA</td>
<td>Set the status word, length of response data and exit from the application</td>
</tr>
</tbody>
</table>

Remarks

The notation SW1 refers to the most significant byte and SW2 refers to the least significant byte of the status word. The notation La corresponds to the actual length of response data value.

When a MEL application exits the response returned to the terminal consists of two bytes, the Status Word. The default value is ‘0x9000’, which indicates successful execution of an application function.

An application may also return response data back to the IFD. If data is to be sent, then the La, Length of Actual Response, should be set to the number of bytes that are to be returned. The default value is 0x00.

The operation of this instruction depends upon the mnemonic used.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Unchanged

**Example**

The following extended example shows how an application could handle an APDU command INS. Note the use of EXITSW to exit the application and return the relevant status word.

```
pINS EQU PT[-12]  // pINS is a label for Public Top - 12
chkCLA90 // Code block chkCLA90
  LOAD pINS,1 // Load the 1 byte instruction to stack
  CMPB ,0x10 // Compare with the hex value 0x10
  BEQ cmd10 // If equal jump to code block cmd10
  CMPB ,0x20 // Compare with the hex value 0x20
  BEQ cmd20 // If equal branch to code block cmd20
UnrecINS // Instruction not recognised by the class
  EXITSW 0x6D,0x00 // Set SW1 to 0x6D and Sw2 to 0x00
```

Continuing from the previous example, the code snippet below illustrates a memory copy. Note that instruction EXITLA uses the default status word and set the actual length of response data.

```
cmd10 // Code block cmd10
  // Pop pCLA & pINS bytes off stack (1 WORD)
  POPW
  // assume sData of size 8 exists
  // copy to public using memory copy fixed length
  LOADA PB[0]
  LOADA sData
  PRIM 0x0E, 8
  // default 9000 SW used and LA set
  EXITLA 8

cmd20 // Code block cmd20
  POPW // Pop pCLA & pINS bytes off stack
  // Command processing for Instruction 20
  EXIT // Exit the application
**TESTN**

This instruction compares a byte-block with zero and sets the zero flag in CCR accordingly.

**Syntax**

```c
TESTN [label], block_length
```

**Remarks**

The block_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the data of size block_length on the stack will be tested.

The result of the operation updates the Condition Code Register. The zero flag is set if the operation is performed a byte-block with a value of zero. For example,

- TESTN 0x000000; Zero is set
- TESTN 0x000010; Zero is cleared

The TESTN instruction does not modify the carry flag of the condition register.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the byte-block equals zero, cleared otherwise
**Example**

The following example tests the four bytes at the top of the stack to determine if they are equal to zero.

```assembly
TESTN SB[0], 4
```

The following example is the same as the previous example, but uses a label to identify the variable instead of using a register/offset pair directly.

```assembly
myVar STATIC BYTE 4
TESTN myVar, 4
```

The following example tests the four bytes at the top of the stack then the top five bytes of the stack to determine if they are all equal to zero.

```assembly
// Stack = FF,00,00,00,00
TESTN , 4  // CCR Z = Set
TESTN , 5  // CCR Z = Cleared
```
**XORN**

This instruction performs a bit-wise XOR on two byte blocks of the same size.

**Syntax**

\[ \text{XORN} \ [\text{label}], \text{block\_length} \]

**Remarks**

The block\_length value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The label, if present, may be either a named memory location, which the assembler will translate into a register / offset pair, or an explicit register / offset pair. If a label is not specified, then the both operands of size block\_length are taken from the stack.

The result of the XOR operation is written to the lower byte-block on the stack, or if specified, the byte-block held at the label. The byte-block at the top of the stack is not changed by this instruction.

The result of the operation updates the zero flag condition code register. The flag is set if the result of the operation is equal to zero. For example,

- \(0xF0F0 \ XOR 0xF0F0 = 0x0000\); Zero is set
- \(0xF0F0 \ XOR 0x00FF = 0xF00F\); Zero is reset

The operation will work if the two blocks overlap

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Set if the result is zero, cleared otherwise
Example

The following example performs a bit-wise XOR operation on the four bytes at the top of the stack with the four bytes at the base of the static area.

```
XORN SB[0], 4
```

The following example is similar to the previous example, but uses a label to identify the variable instead of using a register/offset pair directly.

```
myVar STATIC BYTE 4
XORN myVar, 4
```

The following example pushes two words onto the stack and performs a bit-wise XOR on them before performing a two byte bit-wise XOR with a static variable. Note the way in which the stack and static variable change during the XORN operations.

```
sResult STATIC WORD = 0x1111
PUSHW 0xFF00 // Stack = FF,00
PUSHW 0x1234 // Stack = FF,00,12,34
XORN , 2 // Stack = ED,34,12,34
XORN sResult,2 // sResult = FC, 25
```
Primitives

MULTOS defines built-in functions, known as primitives, are available for use by any application. The MULTOS specification states whether a primitive is mandatory or optional for implementation and a type approved MULTOS implementation must comply with a stated specification. Each sub-section lists the availability and mandatory / optional status of the primitive. Unavailable primitives are identified thus, ❌, optional are identified thus, ☑, and available are identified thus, ✔.

Deprecated primitives, marked ❌, should no longer be used in new applications because either a) they have been superseded by a higher level primitive or b) are little used; they are earmarked for removal in future releases.

As mentioned in the introduction, for MULTOS step/one products, all primitives available in MULTOS are considered optional and a developer should check the specific implementation.

The conventions and assumptions given in the introductory section apply here. There are also further points of note:

- Primitives are divided into sets: Set Zero, Set One, Set Two and Set Three. The classification is based on the number of arguments included in line with the primitive call. For example, “PRIM 0x01” is part of Set Zero as no arguments are present. However, “PRIM 0x01, 1” is part of Set One as there is a single in line argument.
- All arguments are 1 byte in size and must be compile time constants. Stack based parameters are used for variable values used by the primitives.
- The stack operates on the principle “last in, first out”. In the subsections that follow stack usage is illustrated using diagrams such as:

  Stack In: OperandA, OperandB  
  Stack Out: Output

The Stack In values are referred to as input parameters and those in Stack Out are referred to as output parameters. The leftmost value is considered to be the first in. In the example above, OperandA is placed on the stack first followed by OperandB. In terms of addressing the rightmost value is below dynamic top (DT) and each value can be located using negative offsets. If the size of the operands is 2 bytes, then OperandA starts at DT[-4] and OperandB at DT[-2].
- The illustrations use relative sizes to show which operand is larger. So, a 1-byte value should be shown as smaller than a 2-byte value. The actual size of the operands is given in the description that follows the illustration.
Add BCDN

This primitive adds two stack resident unsigned byte blocks of the same size, where the blocks hold binary coded decimal (BCD) values. The result is placed on the stack.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax

//Stack holds Operand1, Operand2 bytes
PRIM 0x11, length

Arguments

The argument length gives the size of the byte blocks to be added.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand1</td>
<td>Operand2</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
</tbody>
</table>

The parameters Operand1 and Operand2 are both of size length and these are the values that will be added. The parameter Output is of size length and holds the result of the addition.

Remarks

The value designated by an operand should be in BCD format. If not in BCD format, the processing in MULTOS device will abnormally end the application.

The CCR C flag is set if the result of the operation is greater than that which can be held in length bytes. The Z flag is set if the result is zero.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set if a carry occurs, cleared otherwise.
V  Unchanged
N  Unchanged
Z  Set if the result is zero, cleared otherwise.

Primitive set and number

Set one, number 0x11
Examples

The following examples are meant to demonstrate how the primitive may be used as well as indicate how the CCR C and Z bit flags are set.

// 99 + 1 = 100
PUSHB 0x99 // stack = 0x99
PUSHB 0x01 // stack = 0x99, 0x01
PRIM 0x11, 1 // stack = 0x00 and CCR C and Z are set
    // NOTE: as length = 1, the normal addition
    // result of 100 is truncated to 00

// 99 + 2 = 101
PUSHB 0x99 // stack = 0x99
PUSHB 0x02 // stack = 0x99, 0x02
PRIM 0x11, 1 // stack = 0x01 and CCR C set and Z cleared
    // NOTE: as length = 1, the normal addition
    // result of 101 is truncated to 01

// 101 + 150 = 251
PUSHW 0x0101 // stack = 0x01, 0x01
PUSHW 0x0150 // stack = 0x01, 0x01, 0x01, 0x50
PRIM 0x11, 2 // stack = 0x02, 0x51 both CCR C and Z are cleared
    // NOTE: as length = 2, the normal addition
    // result of 251 is expressed as 0251

// 0 + 0 = 0
PUSHW 0x0000 // stack = 0x00, 0x00
PUSHW 0x0000 // stack = 0x00, 0x00, 0x00, 0x00
PRIM 0x11, 2 // stack = 0x00, 0x00 and CCR C cleared, Z set
    // NOTE: as length = 2, the normal addition
    // result of 0 is expressed as 0000
AES ECB Decipher
This primitive performs AES ECB Decipher on a sixteen byte block of memory in accordance with [FIPS197].

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0xD6

Arguments

None.

Stack Usage

Stack In

<table>
<thead>
<tr>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
</table>

Stack Out

{empty}

The 2 byte parameter KeyAddr is the starting address of the AES key to be used.
The 1 byte parameter KeyLen is the length in bytes of the AES key at address KeyAddr.
The 2 byte parameter OutputAddr is the starting address of the resultant 16-bytes of plaintext.
The 2 byte parameter InputAddr is the starting address of the 16-bytes of ciphertext.

Remarks

This primitive performs the AES ECB decipher operation on a 16-byte block of memory. The AES key is held in a block of length KeyLen.

Valid key lengths are 16, 24 and 32 bytes.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input.

This primitive is only available to an application if “Strong Cryptography” is set on in the application’s access_list when loaded.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
Primitive set and number

Set one, number 0xD6

Example

The following example declares 16 bytes of static memory to hold the 16 byte (128-bit) length AES Key, the ciphertext is held as session data, while the resulting plaintext will be written to public. The address for each of these is loaded onto the stack and the AES Decipher primitive is called.

```
prmAESDecipher EQU 0xD6
KEYLEN EQU 16

sKey STATIC BYTE 16 =
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,
0x0E,0x0F,0x10
dCiphertext DYNAMIC BYTE 16
pPlaintext PUBLIC BYTE 16

LOADA sKey
PUSHB KEYLEN
LOADA pPlaintext
LOADA dCiphertext
PRIM prmAESDecipher
```
AES ECB Encipher

This primitive performs AES ECB Encipher on a sixteen byte block of memory in accordance with [FIPS197].

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xD7

Arguments

None.

Stack Usage

Stack In

<table>
<thead>
<tr>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
</table>

Stack Out

{empty}

The 2 byte parameter KeyAddr is the starting address of the AES key to be used.
The 1 byte parameter KeyLen is the length in bytes of the AES key at address KeyAddr.
The 2 byte parameter OutputAddr is the starting address of the resultant 16-bytes of ciphertext.
The 2 byte parameter InputAddr is the starting address of the 16-bytes of plaintext.

Remarks

This primitive performs the AES ECB encipher operation on a 16-byte block of memory. The AES key is held in a block of length KeyLen.

Valid key lengths are 16, 24 and 32 bytes.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input.

This primitive is only available to an application if “Strong Cryptography” is set on in the application’s access_list when loaded.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
Primitive Set and Number

Set zero, number 0xD7

Example

The following example declares 24 bytes of static memory to hold the 24-byte (192-bit) AES Key, the plaintext is held as session data, while the resulting ciphertext will be written to public. The address for each of these is loaded onto the stack and the AES Encipher primitive is called.

prmAESEncipher EQU 0xD7
KEYLEN EQU 24

sKey STATIC BYTE 24 =
0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,
0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18
dPlaintext DYNAMIC BYTE 16
pCiphertext PUBLIC BYTE 16

LOADA sKey
PUSHB KEYLEN
LOADA pPlaintext
LOADA dCiphertext
PRIM prmAESEncipher
**Bit Manipulate Byte**

This primitive performs bit-wise operations on the top byte of the stack.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Syntax**

```plaintext
//Stack holds ByteIn parameter bytes
PRIM 0x01, Option, MaskByte
```

**Arguments**

The argument `Option` is a bitmap controlling what logical operation is performed and `MaskByte` is a literal byte holding the mask to use for operation.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>ByteIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>ByteOut</td>
</tr>
</tbody>
</table>

The 1-byte parameter `ByteIn` is the byte value that will be manipulated according to the binary operation specified by `Option` using the literal `MaskByte` as the second operand. The 1-byte value `ByteOut` depends on the `Options` argument. It may be the original byte or the result of the logical operation.

**Remarks**

Depending on the `Option` argument this primitive performs one of four binary logical operations. They are:

- **AND**: which returns a true bit only if both corresponding bits in the input and mask are true
- **OR**: which returns a true bit if either of the corresponding bits in the input or mask are true
- **XOR**: This is a logical Exclusive OR operation, which returns a true bit only if either of the corresponding bits in the input or mask are true, but false if both are true
- **EQU**: This logical operation is also known as a Exclusive NOR (XNOR), which returns a true bit only if both corresponding bits in the input and mask are of the same value
The following table shows how the Option argument is coded. The numbers in the top row correspond to the bit offset, where the most significant bit occupies offset 7.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Do not modify ByteIn</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Overwrite ByteIn with result of operation</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Any other values are undefined</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>Calculate ByteIn XOR MaskByte</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>Calculate ByteIn EQU MaskByte</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>Calculate ByteIn OR MaskByte</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>Calculate ByteIn AND MaskByte</td>
</tr>
</tbody>
</table>

Regardless of whether the ByteIn value is modified, the Condition Code Register reflects the result of the operation.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Set if the result is zero, cleared otherwise

**Primitive Set and Number**

Set two, number 0x01

**Example**

The following table lists acceptable values for the Option argument.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ByteIn remains unchanged after XOR; i.e. ByteOut = ByteIn</td>
</tr>
<tr>
<td>0x01</td>
<td>ByteIn remains unchanged after EQU; i.e. ByteOut = ByteIn</td>
</tr>
<tr>
<td>0x02</td>
<td>ByteIn remains unchanged after OR; i.e. ByteOut = ByteIn</td>
</tr>
<tr>
<td>0x03</td>
<td>ByteIn remains unchanged after AND; i.e. ByteOut = ByteIn</td>
</tr>
<tr>
<td>0x80</td>
<td>ByteOut holds result of ByteIn XOR MaskByte</td>
</tr>
<tr>
<td>0x81</td>
<td>ByteOut holds result of ByteIn EQU MaskByte</td>
</tr>
<tr>
<td>0x82</td>
<td>ByteOut holds result of ByteIn OR MaskByte</td>
</tr>
<tr>
<td>0x83</td>
<td>ByteOut holds result of ByteIn AND MaskByte</td>
</tr>
</tbody>
</table>
The following code snippet uses the primitive to ascertain the value of a bit flag held at bit offset six. The primitive number and option value have been defined using the assembler directive EQU, which is similar in function to the C expression #define and should not be confused with the exclusive NOR operation.

```assembly
prmBitManipByte EQU 0x01
optANDwithResult EQU 0x83
flagValueSet EQU 0x40 // bit 6 set
dynFlags DYNAMIC BYTE 1

LOAD dynFlags, 1
PRIM prmBitManipByte, optANDwithResult, flagValueSet
CMPB , flagValueSet
POPB
BEQ label_flag_set
```

The next example tests the value of the flag variable and will set bit 0 if it is equal to an expected value.

```assembly
optEQUnoResult EQU 0x01
optXORwithResult EQU 0x80
flagExpectedValue EQU 0x5A
flagEVCheckOK EQU 0x01

LOAD dynFlags, 1
PRIM prmBitManipByte, optEQUnoResult, flagExpectedValue
BNE label_expected_value_check_failed
// use primitive again to update flag value on stack
PRIM prmBitManipByte, optXORwithResult, flagEVCheckOK
// move new value back to session variable
STORE dynFlags, 1
```
**Bit Manipulate Word**
This primitive performs bit-wise operations on the top word of the stack.

**Availability**
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Availability Checkmarks]

**Syntax**

```plaintext
//Stack holds WordIn parameter bytes
PRIM 0x01, Option, MaskWord
```

**Arguments**
The argument `Option` is a bitmap controlling what logical operation is performed and `MaskWord` is a literal word holding the mask to use for operation.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordIn</td>
<td>WordOut</td>
</tr>
</tbody>
</table>

The 2-byte parameter `WordIn` is the value that will be manipulated according to the binary operation specified by `Option` using the literal `MaskWord` as the second operand. The 2-byte value `WordOut` depends on the `Options` argument. It may be the original word or the result of the logical operation.

**Remarks**
Depending on the `Option` argument this primitive performs one of four binary logical operations. They are:

- **AND:** which returns a true bit only if both corresponding bits in the input and mask are true
- **OR:** which returns a true bit if either of the corresponding bits in the input or mask are true
- **XOR:** This is a logical Exclusive OR operation, which returns a true bit only if either of the corresponding bits in the input or mask are true, but false if both are true
- **EQU:** This logical operation is also known as a Exclusive NOR (XNOR), which returns a true bit only if both corresponding bits in the input and mask are of the same value
The following table shows how the Option argument is coded. The numbers in the top row correspond to the bit offset, where the most significant bit occupies offset 7.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Do not modify WordIn</td>
</tr>
<tr>
<td>6</td>
<td>Overwrite WordIn with result of operation</td>
</tr>
<tr>
<td>5</td>
<td>Any other values are undefined</td>
</tr>
<tr>
<td>4</td>
<td>Calculate WordIn XOR MaskWord</td>
</tr>
<tr>
<td>3</td>
<td>Calculate WordIn EQU MaskWord</td>
</tr>
<tr>
<td>2</td>
<td>Calculate WordIn OR MaskWord</td>
</tr>
<tr>
<td>1</td>
<td>Calculate WordIn AND MaskWord</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Regardless of whether the WordIn value is modified, the Condition Code Register reflects the result of the operation.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
</table>
```

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Set if result is zero, cleared otherwise

**Primitive Set and Number**

Set three, number 0x01

**Example**

The following table lists acceptable values for the Option argument.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>WordIn remains unchanged after XOR; i.e. WordOut = WordIn</td>
</tr>
<tr>
<td>0x01</td>
<td>WordIn remains unchanged after EQU; i.e. WordOut = WordIn</td>
</tr>
<tr>
<td>0x02</td>
<td>WordIn remains unchanged after OR; i.e. WordOut = WordIn</td>
</tr>
<tr>
<td>0x03</td>
<td>WordIn remains unchanged after AND; i.e. WordOut = WordIn</td>
</tr>
<tr>
<td>0x80</td>
<td>WordOut holds result of WordIn XOR MaskWord</td>
</tr>
<tr>
<td>0x81</td>
<td>WordOut holds result of WordIn EQU MaskWord</td>
</tr>
<tr>
<td>0x82</td>
<td>WordOut holds result of WordIn OR MaskWord</td>
</tr>
<tr>
<td>0x83</td>
<td>WordOut holds result of WordIn AND MaskWord</td>
</tr>
</tbody>
</table>
The following code snippet uses the primitive to set the four least significant bytes of the word on top of the stack. The primitive number and option value have been defined using the assembler directive EQU, which is similar in function to the C expression #define and should not be confused with the exclusive NOR operation.

```assembly
prmBitManipByte EQU 0x01
optORwithResult EQU 0x82

dynFlags DYNAMIC WORD

LOAD dynFlags, 2
PRIM prmBitManipByte, optORwithResult, 0x000F
```
**Block Decipher**

This primitive performs a Block Decipher on a block of memory. The algorithms that may be used are DES, Triple DES, SEED and AES in ECB and CBC modes of operation.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xDA, AlgorithmID, ChainingMode

**Arguments**

The 1-byte argument AlgorithmID indicates the type of decipher algorithm to be used.

<table>
<thead>
<tr>
<th>AlgorithmID</th>
<th>Algorithm</th>
<th>4.2</th>
<th>4.3</th>
<th>4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>DES [FIPS46-3]</td>
<td>Optional</td>
<td>Mandatory</td>
<td>Mandatory</td>
</tr>
<tr>
<td>0x04</td>
<td>Triple DES [FIPS46-3]</td>
<td>Optional</td>
<td>Mandatory</td>
<td>Mandatory</td>
</tr>
<tr>
<td>0x05</td>
<td>SEED [KISA]</td>
<td>Optional</td>
<td>Optional*</td>
<td>Mandatory</td>
</tr>
<tr>
<td>0x06</td>
<td>AES [FIPS197]</td>
<td>Optional</td>
<td>Optional*</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

*mandatory if the algorithm is supported by an implementation

The 1-byte argument Chaining Mode indicates the block cipher mode of operation to be used.

<table>
<thead>
<tr>
<th>ChainingMode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>ECB</td>
</tr>
<tr>
<td>0x02</td>
<td>CBC</td>
</tr>
<tr>
<td>0x03</td>
<td>CTR</td>
</tr>
<tr>
<td>0x04</td>
<td>CFB</td>
</tr>
</tbody>
</table>

**Stack Usage**

**ECB mode**

<table>
<thead>
<tr>
<th>InputLen</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The 2-byte parameter InputLen specifies the number of bytes to decipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key(s) at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
- Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
- SEED: one 16-byte key.
- AES: one 16, 24 or 32 byte AES key.
- The 1-byte parameter KeyLen is the length of the key(s) to be used.
- The 2-byte parameter OutputAddr is the start address of the resultant plaintext.
- The 2-byte parameter InputAddr is the start address of the ciphertext to be deciphered.

**CBC mode**

CBC mode requires the addition of an Initialisation Vector of length equal to the block size for the selected algorithm. The stack for this mode is:

<table>
<thead>
<tr>
<th>Stack In</th>
<th>IVLen</th>
<th>IVAddr</th>
<th>InputLen</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
</table>

- The 1-byte parameter IVLen specifies the size of the Initialisation Vector.
- The 2-byte parameter IVAddr is the address of the Initialisation Vector. The size of the Initialisation Vector depends upon the specified algorithm, as follows.
  - DES: 8 bytes.
  - Two key triple DES: 8 bytes.
  - Three key triple DES: 8 bytes.
  - SEED: 16 bytes.
  - AES: 16 bytes.

- The 2-byte parameter InputLen specifies the number of bytes to decipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
  - Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
  - SEED: one 16-byte key.
  - AES: one 16, 24 or 32 byte AES key.

- The 1 byte parameter KeyLen is the length in bytes of the key at address KeyAddr.
- The 2-byte parameter OutputAddr is the starting address of the resultant plaintext.
- The 2-byte parameter InputAddr is the start address of the ciphertext to be deciphered.

**CTR chaining mode**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>IVLen</th>
<th>IVAddr</th>
<th>InputLen</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
<th>Counter Width</th>
</tr>
</thead>
</table>

CTR mode is specified in ISO/IEC-10116.
- The 1-byte parameter IVLen specifies the size of the Initialisation Vector.
- The 2-byte parameter IVAddr is the address of the Initialisation Vector. The size of the Initialisation Vector depends upon the specified algorithm, as follows.
  - DES: 8 bytes.
  - Two key triple DES: 8 bytes.
  - Three key triple DES: 8 bytes.
  - SEED: 16 bytes.
  - AES: 16 bytes.

- The 2-byte parameter InputLen specifies the number of bytes to decipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
  - Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
  - SEED: one 16-byte key.
  - AES: one 16, 24 or 32 byte AES key.

- The 1 byte parameter KeyLen is the length in bytes of the key at address KeyAddr.
- The 2-byte parameter OutputAddr is the starting address of the resultant plaintext.
- The 2-byte parameter InputAddr is the starting address of the ciphertext to be deciphered.
- The 1-byte parameter CounterWidth refers to the width of the counter to be used. This can be up to the IVLen.

Examples of different counter widths:
AES CTR mode, CounterWidth = 4, IVLen = 16.
IV = FFEEDDCCBBAA99887766554433221100

| Round | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 00 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Round 1 | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 00 |
| Round 2 | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 01 |
| Round 3 | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 02 |

AES CTR mode, CounterWidth = 4, IVLen = 16.
IV = FFEEDDCCBBAA998877665544FFFFFFFF

<table>
<thead>
<tr>
<th>Round</th>
<th>IV</th>
<th>FF</th>
<th>EE</th>
<th>DD</th>
<th>CC</th>
<th>BB</th>
<th>AA</th>
<th>99</th>
<th>88</th>
<th>77</th>
<th>66</th>
<th>55</th>
<th>44</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td>FF</td>
<td>EE</td>
<td>DD</td>
<td>CC</td>
<td>BB</td>
<td>AA</td>
<td>99</td>
<td>88</td>
<td>77</td>
<td>66</td>
<td>55</td>
<td>44</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>Round 2</td>
<td>FF</td>
<td>EE</td>
<td>DD</td>
<td>CC</td>
<td>BB</td>
<td>AA</td>
<td>99</td>
<td>88</td>
<td>77</td>
<td>66</td>
<td>55</td>
<td>44</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Round 3</td>
<td>FF</td>
<td>EE</td>
<td>DD</td>
<td>CC</td>
<td>BB</td>
<td>AA</td>
<td>99</td>
<td>88</td>
<td>77</td>
<td>66</td>
<td>55</td>
<td>44</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

AES CTR mode, CounterWidth = 16, IVLen = 16.
IV = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

<table>
<thead>
<tr>
<th>Round</th>
<th>IV</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>Round 2</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>Round 3</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
</tbody>
</table>

**CFB mode**

CFB mode requires the addition of an Initialisation Vector of length equal to the block size for the selected algorithm. The stack for this mode is:

Stack In | IVLen | IVAddr | Input | KeyAddr | KeyLen | Output | Input | FeedbackSi
- The 1-byte parameter IVLen specifies the size of the Initialisation Vector.
- The 2-byte parameter IVAddr is the address of the Initialisation Vector. The size of the Initialisation Vector depends upon the block length of the specified algorithm, as follows.
  - DES: 8 bytes.
  - Two key triple DES: 8 bytes.
  - Three key triple DES: 8 bytes.
  - SEED: 16 bytes.
  - AES: 16 bytes.

- The 2-byte parameter InputLen specifies the number of bytes to decipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
  - Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
  - SEED: one 16-byte key.
  - AES: one 16, 24 or 32 byte AES key.

- The 1-byte parameter KeyLen is the length in bytes of the key at address KeyAddr.
- The 2-byte parameter OutputAddr is the starting address of the resultant plaintext.
- The 2-byte parameter InputAddr is the start address of the ciphertext to be deciphered.
- The 1-byte parameter FeedbackSize refers to the number of bits to be used as feedback. This can be upto the block length of the algorithm.

Remarks

This primitive performs the block decipher operation on a block of memory of InputLen bytes. The algorithm used may be DES, Two Key Triple DES, Three Key Triple DES, SEED and 128/192/256-bit Key AES. The key is held in a block of the appropriate length for the algorithm.

In ECB and CBC chaining mode, DES algorithms require that the ciphertext length is a multiple of 8 bytes and the SEED and AES-128/192/256 algorithms require that the ciphertext is a multiple of 16 bytes. If the ciphertext length does not meet these restrictions then the primitive will abend. Padding is not removed during the block decipher operation.

In CTR and CFB mode there is no restriction that the plaintext be a multiple of the algorithms block length.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input. However, the output plaintext cannot partially overlap the input ciphertext. If the primitive is called with partially overlapping input and output memory areas then it abends. If an Initialisation Vector is used then this can be at any segment address, including in the input ciphertext or output plaintext memory areas.
Refer to the relevant MULTOS Implementation Report for the platform you are developing on as not all combinations of algorithm and chaining modes may be supported by the platform.

CTR mode may not be available for all algorithms and the implementation may restrict the maximum CounterSize supported. Refer to the MULTOS Implementation Report for the platform.

This primitive is only available to an application if “Strong Cryptography” are set on in the application’s access_list when loaded.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C       Unchanged  
V       Unchanged  
N       Unchanged  
Z       Unchanged.

**Primitive Set and Number**

Set two, number 0xDA
**Block Encipher**

This primitive performs a Block Encipher on a block of memory. The algorithms that may be used are DES, Triple DES, SEED and AES in ECB and CBC modes of operation.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xDB, AlgorithmID, ChainingMode

**Arguments**

The 1-byte argument AlgorithmID indicates the type of encipher algorithm to be used.

<table>
<thead>
<tr>
<th>AlgorithmID</th>
<th>Algorithm</th>
<th>4.2</th>
<th>4.3</th>
<th>4.4 / 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>DES [FIPS46-3]</td>
<td>Optional</td>
<td>Mandatory</td>
<td>Mandatory</td>
</tr>
<tr>
<td>0x04</td>
<td>Triple DES [FIPS46-3]</td>
<td>Optional</td>
<td>Mandatory</td>
<td>Mandatory</td>
</tr>
<tr>
<td>0x05</td>
<td>SEED [KISA]</td>
<td>Optional</td>
<td>Optional*</td>
<td>Optional*</td>
</tr>
<tr>
<td>0x06</td>
<td>AES [FIPS197]</td>
<td>Optional</td>
<td>Optional*</td>
<td>Optional*</td>
</tr>
</tbody>
</table>

*mandatory if the algorithm is supported by an implementation

The 1-byte argument Chaining Mode indicates the block cipher mode of operation to be used.

<table>
<thead>
<tr>
<th>ChainingMode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>ECB</td>
</tr>
<tr>
<td>0x02</td>
<td>CBC</td>
</tr>
<tr>
<td>0x03</td>
<td>CTR</td>
</tr>
<tr>
<td>0x04</td>
<td>CFB</td>
</tr>
</tbody>
</table>

**Stack Usage**

**ECB mode**

Stack In: InputLen, KeyAddr, KeyLen

Stack Out: {empty}

- The 2-byte parameter InputLen specifies the number of bytes to encipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key(s) at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
- Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
- SEED: one 16-byte key.
- AES: one 16, 24 or 32 byte AES key.

- The 1-byte parameter KeyLen is the length of the key(s) to be used.
- The 2-byte parameter OutputAddr is the start address of the resultant ciphertext.
- The 2-byte parameter InputAddr is the start address of the plaintext to be enciphered.

**CBC chaining mode**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>IVLen</th>
<th>IVAddr</th>
<th>InputLen</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The 1-byte parameter IVLen specifies the size of the Initialisation Vector.
- The 2-byte parameter IVAddr is the address of the Initialisation Vector. The size of the Initialisation Vector depends upon the specified algorithm, as follows.
  - DES: 8 bytes.
  - Two key triple DES: 8 bytes.
  - Three key triple DES: 8 bytes.
  - SEED: 16 bytes.
  - AES: 16 bytes.

- The 2-byte parameter InputLen specifies the number of bytes to encipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
  - Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
  - SEED: one 16-byte key.
  - AES: one 16, 24 or 32 byte AES key.

- The 1-byte parameter KeyLen is the length in bytes of the key at address KeyAddr.
- The 2-byte parameter OutputAddr is the starting address of the resultant ciphertext.
- The 2-byte parameter InputAddr is the start address of the plaintext to be enciphered.

**CTR chaining mode**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>IVLen</th>
<th>IVAddr</th>
<th>InputLen</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
<th>Counter Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CTR mode is specified in ISO/IEC-10116.
- The 1-byte parameter IVLen specifies the size of the Initialisation Vector.
- The 2-byte parameter IVAddr is the address of the Initialisation Vector. The size of the Initialisation Vector depends upon the specified algorithm, as follows.
- The 2-byte parameter InputLen specifies the number of bytes to encipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
  - Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
  - SEED: one 16-byte key.
  - AES: one 16, 24 or 32 byte AES key.
- The 1-byte parameter KeyLen is the length in bytes of the key at address KeyAddr.
- The 2-byte parameter OutputAddr is the starting address of the resultant ciphertext.
- The 2-byte parameter InputAddr is the start address of the plaintext to be enciphered.
- The 1-byte parameter CounterWidth refers to the width of the counter to be used. This can be up to the IVLen.

Examples of different counter widths:
AES CTR mode, CounterWidth = 4, IVLen = 16.
IV = FFEEDDCCBBAA99887766554433221100

| Round | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 00 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Round 1 | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 01 |
| Round 3 | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 | 11 | 02 |

AES CTR mode, CounterWidth = 4, IVLen = 16.
IV = FFEEDDCCBBAA998877665544FF00000000

| Round 1 | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | FF | FF | FF | FF |
| Round 2 | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 00 | 00 | 00 | 00 |
| Round 3 | IV | FF | EE | DD | CC | BB | AA | 99 | 88 | 77 | 66 | 55 | 44 | 00 | 00 | 00 | 01 |

AES CTR mode, CounterWidth = 16, IVLen = 16.
IV = FFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000001

| Round 1 | IV | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF |
| Round 2 | IV | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| Round 3 | IV | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 01 |

**CFB chaining mode**

- The 1-byte parameter IVLen specifies the size of the Initialisation Vector.
- The 2-byte parameter IVAddr is the address of the Initialisation Vector. The size of the Initialisation Vector depends upon the block length of the specified algorithm, as follows.
  - DES: 8 bytes.
  - Two key triple DES: 8 bytes.
  - Three key triple DES: 8 bytes.
  - SEED: 16 bytes.
  - AES: 16 bytes.

- The 2-byte parameter InputLen specifies the number of bytes to encipher.
- The 2-byte parameter KeyAddr is the address of the key(s) to be used. The size and format of the key at this address depends upon the specified algorithm, as follows.
  - DES: one 8-byte DES key.
  - Two key triple DES: two 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8.
  - Three key triple DES: three 8-byte DES keys. The first key is located at address KeyAddr and the second key is located at address KeyAddr+8 and the third key is located at address KeyAddr+16.
  - SEED: one 16-byte key.
  - AES: one 16, 24 or 32 byte AES key.

- The 1 byte parameter KeyLen is the length in bytes of the key at address KeyAddr.
- The 2-byte parameter OutputAddr is the starting address of the resultant ciphertext.
- The 2-byte parameter InputAddr is the start address of the plaintext to be enciphered.
- The 1-byte parameter FeedbackSize refers to the number of bits to be used as feedback. This can be upto the block length of the algorithm.

Remarks

This primitive performs the block encipher operation on a block of memory of InputLen bytes. The algorithm used may be DES, Two Key Triple DES, Three Key Triple DES, SEED and 128/192/256-bit Key AES. The key is held in a block of the appropriate length for the algorithm.

In ECB and CBC chaining mode, DES algorithms require that the plaintext length is a multiple of 8 bytes and the SEED and AES-128/192/256 algorithms require that the plaintext is a multiple of 16 bytes. If the plaintext length does not meet these restrictions then the primitive will abend.

In CTR and CFB mode there is no restriction that the plaintext be a multiple of the algorithms block length.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input. If an implementation has any restrictions around this then it will be documented in the [MIR]. If an Initialisation Vector is used then this can be at any segment address, including in the input plaintext or output ciphertext memory areas.

Refer to the relevant MULTOS Implementation Report for the platform you are developing on as not all combinations of algorithm and chaining modes may be supported by the platform.

CTR mode may not be available for all algorithms and the implementation may restrict the maximum CounterSize supported. Refer to the MULTOS Implementation Report for the platform.
This primitive is only available to an application if “Strong Cryptography” are set on in the application’s access_list when loaded.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged.

**Primitive Set and Number**

Set two, number 0xDB
Call Codelet
This primitive is used to access a code held in a codelet.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x
✓  ✓  ✓  ✓  ✓  ✓

Syntax
PRIM 0x83

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>CodeletID</th>
<th>codeAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>LinkageData</td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte CodeletID identifies which codelet to execute, while the 2-byte codeAddr is the start address within the codelet’s code area at which to start execution. The 6-byte LinkageData field is automatically handled by the operating system. These values are present so that when the codelet uses the primitive ‘Return from Codelet’ MULTOS can continue the normal execution of the application.

Remarks
This primitive is used to call a codelet that is stored on the MULTOS device. When the codelet is called it is considered to be part of the executing application’s memory space. Therefore, the codelet is able to read from and write to any data area of the application’s memory.

The 2-byte codelet ID is a unique MULTOS KMA registered value, which identifies a particular codelet. If a codelet with ID of 0 is called, the AAM will execute the currently selected application from the code segment offset specified by the second parameter placed on the stack. Use of this primitive with a codelet ID which is not stored on the device will result in the abnormal end of application execution.

The value CodeAddr is the code address of the entry point within the Codelet where execution will begin. Valid Codelet Entry Addresses for the codelet must be obtained from the provider of the codelet.

This primitive is used in conjunction with the ‘Return from Codelet’ primitive. That primitive will return control to the application code.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

**Primitive set and number**

Set zero, number 0x83

**Example**

The following example checks that a particular codelet exists and if so proceeds to call the codelet. Note that the codelet ID used below is fictitious.

```asm
prmCallCodelet EQU 0x83
prmQueryCodelet EQU 0x84
CODELETID EQU 0xF1F2

PUSHW CODELETID
PRIM prmQueryCodelet
  // CCR Z flag cleared if does not exist
BEQ warning_CodeletUnsupported
  // otherwise call the codelet from start
  // codelet ID remained on stack
PUSHZ 2
PRIM prmCallCodelet
```

MULTOS is a registered trademark of MULTOS Limited.
Call Extension 0,1,2,3,4,5,6
These primitives call an proprietary extension primitive.

Availability

MULTOS 4   MULTOS 4.2   MULTOS 4.3.1   MULTOS 4.3.2   MULTOS 4.4   MULTOS 4.5.x
✔✔✔✔✔✔

Syntax

PRIM 0x80, PrimType_LSB, PrimType_MSB, ParamByte
PRIM 0x81, PrimType_LSB, PrimType_MSB, ParamByte
PRIM 0x82, PrimType_LSB, PrimType_MSB, ParamByte
PRIM 0x83, PrimType_LSB, PrimType_MSB, ParamByte
PRIM 0x84, PrimType_LSB, PrimType_MSB, ParamByte
PRIM 0x85, PrimType_LSB, PrimType_MSB, ParamByte
PRIM 0x86, PrimType_LSB, PrimType_MSB, ParamByte

Arguments

The 1-byte arguments PrimType_MSB and PrimType_LSB represent the most significant byte and least significant byte of the primitive type. The 1-byte argument ParamByte is an optional parameter that may be passed to the primitive.

Stack Usage

Stack In       {empty}
Stack Out      {empty}

Remarks

These primitives are intended to permit up to six implementors to introduce proprietary extension primitives; i.e., primitives that are not described in this document. As these are proprietary the exact usage of these primitives are dependent upon the implementation.

MAOSCO will assign each MULTOS implementor one of the Call Extension Primitive Numbers, from zero to six, and each is able to add up to 65535 new primitives to their MULTOS implementations.
### Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C: Unchanged
- V: Unchanged
- N: Unchanged
- Z: Unchanged

### Primitive Set and Number

Set Three, Numbers 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86
CardBlock
This primitive blocks the MULTOS device.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0x05, MSB_StartAddress_MAC, LSB_StartAddress_MAC

Arguments
The 1-byte arguments MSB_StartAddress_MAC and LSB_StartAddress_MAC represent the most significant and least significant byte of the offset in Public memory where the MAC value is held.

In MULTOS 4.3 and above, the 2 arguments are ignored and may contain any value.

Stack Usage
Stack In   {empty}
Stack Out  {empty}

Remarks
For MULTOS 4.2 and below:
In order for an application to use this primitive to block a MULTOS device a Card Block MAC (CBMAC) must be supplied to the application to authenticate the card block operation.

For MULTOS 4.3 and above:
This primitive is successful if the “card_block” bit in the application’s access_list is set.

If successful the zero flag is set, and the device is blocked. A blocked device will not allow any applications to be selected either implicitly, as is the case with default and shell mode, or explicitly through the Select File command. However, during the session in which a device is blocked the application that called the primitive is still operational and may continue to process commands. Once the application session ends it and all applications can not be selected. Once a device is blocked MULTOS will return a response status of "6A81, Function Not Supported" if an attempt is made to select an application, unless that application has the “card_unblock” bit set in the application’s access_list.

If unsuccessful the zero flag is reset, and the device’s blocked status is not changed. The application may continue processing as normal and the MULTOS device will continue to process as before.
The Card Block primitive is closely associated with the EMV command Card Block. Please note that the EMV MAC supplied is intended to permit an application to verify the authenticity of the APDU command data. On the other hand the CBMAC allows the MULTOS 4.2 and below device to authenticate that the request to block the device.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the device is successfully blocked, and cleared otherwise.

**Primitive Set and Number**

set two, number 0x05

**Examples**

The following example illustrates the use of this primitive when the CBMAC value is held at the base of public memory. The incoming data is structured as given in the example.

```assembly
prmCardBlock EQU 0x05
// assumes public data starts here
pCBMAC PUBLIC BYTE 8
pEMVMACPUBLIC BYTE 8

// assumes a CheckEMVMAC function with 8-byte MAC as parameter
LOAD pEMVMAC, 8
CALL CheckEMVMAC
// assumes function cleans stack and sets CCR.Z
BNE label_failed_EMVMAC_check
// otherwise, do card block with CBMAC at PB[0] = PB[0000]
PRIM prmCardBlock, 0, 0
// check result
BEQ label_card_blocked
```

This second example assumes that the CBMAC is placed at different offsets within public and that the first two bytes of incoming command data correspond to the most significant and least significant bytes respectively. No EMV MAC handling is shown.

```assembly
// assumes public starts here
pMSB PUBLIC BYTE 1
pLSB PUBLIC BYTE 1
pVariableData PUBLIC BYTE 32

// just do card block
PRIM prmCardBlock, pMSB, pLSB
```
**CardUnBlock**
This primitive unblocks the MULTOS device.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Availability Status](image)

**Syntax**

PRIM 0x13

**Arguments**

None.

**Stack Usage**

Stack In  {empty}
Stack Out  {empty}

**Remarks**

This primitive is successful if the device is currently blocked and the “card_unblock” bit in the application’s access_list is set.

If successful the zero flag is set, and the device is unblocked.
If unsuccessful the zero flag is reset, and the device’s blocked status is not changed.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the device is successfully unblocked, and cleared otherwise.

**Primitive Set and Number**

set zero, number 0x13

**Examples**

The following example illustrates the use of this primitive to unblock a device.
prmCardBlock   EQU 0x13

PRIM prmCardUnBlock
// check result
BEQ label_card_unblocked
**Check BCD**

This primitive returns whether the number provided is in binary decimal format.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xDA

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Address</td>
</tr>
<tr>
<td>Result</td>
<td></td>
</tr>
</tbody>
</table>

The Length parameter is one byte and the Address parameter is two bytes in size. The Length is the length of the block and Address is the segment address of the block containing the number to be tested. Result is one byte and holds the result of the operation as follows:

- 0 = Not a BCD number
- 1 = BCD number

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive set and number**

Set zero, number 0xDA
Check Case

This primitive performs three vital functions in that it:

- instructs the operating system how to interpret incoming command APDU
- validates the incoming command as far as is possible under the transport protocol
- permits MULTOS to handle low level communication between the device and the terminal

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓ ✓ ✓ ✓ ✓ ✓

Syntax

//Stack holds ISOCase parameter byte
PRIM 0x01

Stack Usage

Stack In  ISOCase
Stack Out  {empty}

The 1-byte parameter ISOCase indicates which ISO Command Case is expected.

Arguments

None.

Remarks

ISO/IEC 7816 – 4 describes the four possible command cases. In brief, they are:

<table>
<thead>
<tr>
<th>Case</th>
<th>Command Data Sent</th>
<th>Response Data Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Once an incoming command has been identified by the application as being one that it can process, Check Case should be called using the expected ISO command case as the stack based parameter. If the data in public is consistent with the expected command case, the CCR Z flag will be set and cleared otherwise. If the ISOCase parameter is not a valid command case indicator the primitive will consider this to be an inconsistency and clear the CCR Z flag.

The operating system’s handling of a Case 1 command is such that only a status word is returned. There are, however, some interface devices that expect an acknowledgement byte to be transmitted prior to the status word. In order to cater for these devices both MULTOS 4 and MULTOS 4.2 support...
an ISOCase parameter value of 5. The handling of this case value is exactly the same as that for Case 1 with the exception that an acknowledgement byte is transmitted.

The amount of APDU command checking that can be performed by the primitive is based on the transport protocol in use. In most cases an application does not need to be aware of the low level transport handling that occurs as MULTOS ensures that it is takes place. In those cases where more information is required, please see [MDG].

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Set if the specified case is consistent with the data in Public, cleared if it is not

**Primitive Set and Number**

Set zero, number 0x01

**Example**

The following example checks that the ISO Case of the current APDU is ISO Case 2 and jumps to an error handler if the wrong case is detected.

```assembly
prmCheckCase EQU 0x01
PUSHB 0x02
PRIM prmCheckCase
JNE errWrongCase
//continue processing ISO Case 2
```
Checksum
This primitive generates a four byte checksum over a block of memory of variable size.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
//Stack holds Length, BlockAddr parameter
PRIM 0x82

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Length</th>
<th>BlockAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Out</th>
<th>CheckSum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter Length is the size in bytes of the area over which to calculate a checksum value. The BlockAddr parameter is 2 bytes in size and indicates the start address of the input block. Both of these are overwritten by 4-byte result of the checksum algorithm given as CheckSum above.

Remarks
The following C code illustrates an implementation of the checksum algorithm:

```c
unsigned byte message[Length]
unsigned byte checksum[4];

checksum[0] = 0x5A;
checksum[1] = 0xA5;
checksum[2] = 0x5A;
checksum[3] = 0xA5;
for ( j = 0; j < Length; ++ j )
{
    // add data byte into most significant byte of checksum
    checksum[0] += message[j];
    // add each byte of checksum into the next byte
    checksum[1] += checksum[0];
    checksum[2] += checksum[1];
    checksum[3] += checksum[2];
}
```
The result of each byte addition is held in single byte, where any result greater than 255 is truncated in such a way that the least significant byte of the result is maintained. That is to say the carries are dropped from each addition. For example, 0xFF + 0x02 = 0x01.

If the block is in Static and transaction protection is on, the checksum calculation takes pending writes into account. This is an exception to the general rule that pending writes are not visible to the application until they are committed.

It is valid to calculate the checksum of a block of length zero; the result is 0x5AA5AA5.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Unchanged

**Primitive Set and Number**

Set zero, number 0x82

**Example**

The first example code fragment calculates the checksum of the word 0x9988:

```assembly
CheckSumMe DYNAMIC BYTE 2 = 0x99, 0x88
PUSHW 0x0002    // length of block to check sum
LOADA CheckSumMe // 2 byte address of block to check sum
                   // stack now has 4 bytes
PRIM 0x82        // invoke CheckSum primitive
                   // result on stack = 0x7B, 0x13, 0x05, 0x9C
                   // overwrites 4 byte input
```

The result, where all the operands and results are hexadecimal values, is calculated as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5A</td>
<td>A5</td>
<td>5A</td>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>1st Byte (0x99)</td>
<td>5A + 99 = F3</td>
<td>A5 + F3 = 98</td>
<td>5A + 98 = F2</td>
<td>A5 + F2 = 97</td>
</tr>
<tr>
<td>2nd Byte (0x88)</td>
<td>F3 + 88 = 7B</td>
<td>98 + 7B = 13</td>
<td>F2 + 13 = 05</td>
<td>97 + 05 = 9C</td>
</tr>
</tbody>
</table>

The following example performs a check sum over a block of the static area. This may be used to verify that data has been loaded into the variables correctly. It is assumed that the correct value for the checksum is held in the bottom four bytes of public

```assembly
prmCheckSum EQU 0x82
sName STATIC BYTE 10
```
PUSHW 10
LOADA sName
PRIM prmCheckSum
CMPN PB[0000],4
JNE InvalidCheckSum
Configure READ BINARY

This primitive configures/deactivates the accelerated MULTOS READ BINARY command to directly access a specified application Static memory space with optional secure messaging.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Syntax

\[\text{PRIM} \ 0xDC, \ \text{Options}\]

Arguments

The 1 byte argument \(\text{Options}\) is used to configure or deactivate the accelerated READ BINARY command as follows.

<table>
<thead>
<tr>
<th>b7-b4</th>
<th>b3-b0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Deactivate the accelerated READ BINARY command</td>
</tr>
<tr>
<td>0</td>
<td>1-F</td>
<td>RFU</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Activate accelerated ICAO READ BINARY command with no BAC</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Activate accelerated ICAO READ BINARY command with optional BAC</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Activate accelerated ICAO READ BINARY command with mandatory BAC</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Activate accelerated ICAO READ BINARY command with optional BAC lite (no MAC)</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>Activate accelerated ICAO READ BINARY command with mandatory BAC lite (no MAC)</td>
</tr>
<tr>
<td>1</td>
<td>5-F</td>
<td>Reserved for future ICAO Read Binary secure messaging modes (e.g. BAC Lite)</td>
</tr>
<tr>
<td>2-F</td>
<td>x</td>
<td>RFU</td>
</tr>
</tbody>
</table>

Stack Usage (Options = 0x00)

If \(\text{Options} = 0x00\) then the stack will contain the following:

- Stack In: \{empty\}
- Stack Out: \{empty\}

Stack Usage (Options = 0x10)

If \(\text{Options} = 0x10\) then the stack will contain the following:
The 1-byte *Channel* value identifies the channel containing the data to be directly accessed by the accelerated READ BINARY command. A value of 0 specifies the applications Static memory; all other values are RFU.

The 2-byte *DataAddr* value identifies the segment address of the parameter block for the specified channel. For channel 0 the parameter block contains the following parameters, from low address to high address:

- 4-byte offset from the start of Static of the data that is to be directly accessed by the accelerated READ BINARY command.
- 4-byte length of the Static data that is to be directly accessed by the accelerated READ BINARY command.

**Stack Usage (Options = 0x11 or 0x12)**

If *Options* = 0x11 or 0x12 then the stack will contain the following:

Stack In  
---
**Channel** | **DataAddr**
---

Stack Out  
---
(empty)

The 2-byte values *SSCAddr*, *KeyMacAddr* and *KeyEncAddr* identify the segment address of the 8-byte counter, 16-byte MAC key and 16-byte encryption key that are used by the secure messaging.

The 1-byte *Channel* value identifies the channel containing the data to be directly accessed by the accelerated READ BINARY command. A value of 0 specifies the applications Static memory; all other values are RFU.

The 2-byte *DataAddr* value identifies the segment address of the parameter block for the specified channel. For channel 0 the parameter block contains the following parameters, from low address to high address:

- 4-byte offset from the start of Static of the data that is to be directly accessed by the accelerated READ BINARY command.
- 4-byte length of the Static data that is to be directly accessed by the accelerated READ BINARY command.

**Stack Usage (Options = 0x13 or 0x14)**

If *Options* = 0x13 or 0x14 then the stack will contain the following:

Stack In  
---
**KeyEncAddr** | **Channel** | **DataAddr**
---

Stack Out  
---
(empty)

The parameter descriptions are the same as above for options 0x11 and 0x12.
Remarks

The Z flag is set if the configuration/deactivation is successful. It is cleared if the Options argument contains an unsupported value. Invalid segment or Static addresses will cause an abend.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Set if successful, cleared otherwise.

Primitive Set and Number

Set one, number 0xDC
Control Auto Reset WWT
This primitive controls the MULTOS automatic requesting of the Work Waiting Time extensions.

Availability
MULTOS 4   MULTOS 4.2   MULTOS 4.3.1   MULTOS 4.3.2   MULTOS 4.4   MULTOS 4.5.x
✓   ✓   ✓   ✓   ✓   ✓

Syntax
// Stack holds Flag which indicates whether the reset WWT
// functionality should be disabled or enabled
PRIM 0x10

Arguments
None.

Stack Usage

Stack In   Flag
Stack Out   {empty}

The 1-byte parameter Flag can take one of two values. If it has a value of 0x00, the function is enabled and if the value is 0x01 it is disabled.

Remarks
By default, MULTOS causes a message to be automatically sent to the terminal to inform it that more time is required for processing to complete during the execution of long commands. An application can, if it wishes, send these messages manually by calling the Control Auto Reset WWT primitive with Flag = 0x01 and then calling the Reset WWT primitive periodically during the execution of the command. The time during which MULTOS does not automatically reset the WWT is from when the primitive is called to when MULTOS sends the command response back to the IFD or to when the application calls the Control Auto Reset WWT primitive again with Flag = 0x00, whichever comes first.

The Control Auto Reset WWT primitive completely disables the MULTOS automatic generation of reset WWT messages, even during computationally intensive primitives. MULTOS only guarantees that no reset WWT messages are generated by MULTOS if the Control Auto Reset WWT primitive is called in the first two MEL instructions executed following the reception of an application command. The first pushes the byte value 0x01 the second calls the primitive.

The status of the auto reset functionality of MULTOS is maintained when one application delegates to another application or when a delegate application exits. For example, (1) if an application disables the MULTOS automatic generation of reset WWT messages and then delegates to a second application, the automatically generate reset WWT messages will remain disabled. (2) if an application delegates to a second application which then disables the MULTOS automatic generation of reset
WWT messages, on exit back to the first application, automatic generation of reset WWT messages will remain disabled.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C Unchanged
- V Unchanged
- N Unchanged
- Z Unchanged

**Primitive Set and Number**

Set zero, number 0x10

**Example**

The following code fragment shows the recommended declarations and usage for this primitive.

```plaintext
prmControlAutoResetWWT EQU 0x10
```

The following example makes a call to the Control Auto Reset WWT primitive to stop MULTOS automatically resetting the WWT.

```plaintext
PUSHB 0x01
PRIM prmControlAutoResetWWT
```

The following example makes a call to the Control Auto Reset WWT primitive to start MULTOS automatically resetting the WWT.

```plaintext
PUSHB 0x00
PRIM prmControlAutoResetWWT
```
**Convert BCD**

This primitive converts a BCD-encoded value to the equivalent binary value and a binary value to the equivalent BCD-encoded value.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- ❌
- ✓
- ✓
- ✓
- ✓
- ✓

**Syntax**

PRIM 0x14, Mode

**Arguments**

The argument Mode specifies the BCD conversion to be performed and must be equal to one of the following values.

- 0x00 to convert a BCD-encoded value to the equivalent binary value.
- 0x01 to convert a binary value to the equivalent BCD-encoded value.

The primitive abends if the argument Mode is not equal to one of these values.

**Stack Usage**

Stack In

<table>
<thead>
<tr>
<th>DestLength</th>
<th>SourceLength</th>
<th>DestAddr</th>
<th>SourceAddr</th>
</tr>
</thead>
</table>

Stack Out

[empty]

The DestLength and SourceLength parameters are one byte in size and the DestAddr and SourceAddr parameters are two bytes in size. The values SourceLength and DestLength are the length of the source and destination data in bytes. The values SourceAddr and DestAddr are the segment addresses of the source and destination data.

**Remarks**

When converting from a BCD-encoded value the source data must have a valid BCD format for the conversion to succeed. If the format is invalid then the destination data is undefined and the Condition Code Register’s Z flag becomes cleared.

When performing a conversion the number of bytes needed to hold the converted value may be larger than the destination length. If the converted value is too large to be held in the destination area then the destination data is undefined and the Condition Code Register’s Z flag becomes cleared.

If the conversion succeeds then the Condition Code Register’s Z flag becomes set.

This primitive works correctly even if the source and destination blocks overlap.
Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the conversion succeeds, cleared otherwise.

Primitive Set and Number

Set One, number 0x14

Example

The following converts a BCD-encoded value held in Public into the equivalent binary value held in Static.

```assembly
prmConvertBCD EQU 0x14
PUSHB 0x06        //Length of destination area
PUSHB 0x06        //Length of source area
LOADA SB[0000]    //Address of destination data
LOADA PB[0000]    //Address of source data
PRIM prmConvertBCD, 0x00
```
Delegate
This primitive allows an application to invoke another application on the MULTOS device; that is the current application temporarily ceases to execute and the delegate application is executed.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x
✔️  ✔️  ✔️  ✔️  ✔️  ✔️

Syntax
PRIM 0x80

Arguments
None.

Stack Usage
Stack In  AIDAddr
Stack Out  [empty]

The 2-byte parameter AIDAddr is the starting address of the application ID to which the executing application wishes to delegate.

Remarks
The delegate application must be specified by an AID field, which is defined as a one-byte length followed by an Application Identifier of the length given. For example, to delegate to an application with an Application ID of F000000000AB the AID field, sAID below, would be:

sAID STATIC BYTE 6 = 0x05, 0xF0, 0x00, 0x00, 0x00, 0xAB

The Delegation primitive supports partial Application IDs; that is to say if the Application ID to delegate to is shorter than an applications AID then they are considered to match if the most significant bytes match. For example, if an attempt is made to delegate to an application with AID of 0xF000 then the application with AID given above will be considered as a match since the most significant bytes of the application’s AID match the given AID.
Delegation fails, and 0x6A83 is placed in SW1-SW2, if:

- there is no application whose AID matches the AID specified by the delegator.
- the AID length is outside the permissible range of 1 to 16 bytes inclusive.
- the delegate is already active; i.e., an attempt is made to delegate recursively
- the implementation defined maximum number of delegations has been exceeded

If the delegate application abends then the MULTOS device goes mute and all execution of application ceases.

If transaction protection is on, Delegate rolls back any uncommitted writes and turns transaction protection off. Delegate always has this effect regardless of whether delegation succeeds.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C: Unchanged
- V: Unchanged
- N: Unchanged
- Z: Unchanged

**Primitive Set and Number**

Set zero, number 0x80

**Example**

The following example writes an APDU into the Public area and delegates to an application with AID of 0xF000000000001.

```assembly
prmDelegate EQU 0x80
// pSW1 is the 2 byte SW1 SW2 of the Status Word
sAID STATIC BYTE 8 = 7,0xF0,0,0,0,0,0,1

// Delegation
LOADA sAID
PRIMprmDelegate
// Check if SW = 6A83
LOADpSW1,2
CMPWpSW1,0x6A,0x83
// if so, jump to failed delegation handling
JEQ DelegateFailed
```
DES ECB Decipher
This primitive performs DES ECB Decipher on an eight byte block of memory.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0xC5

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>KeyAddr</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each parameter is 2 bytes in size and represents the starting address of an 8-byte block of memory.

Remarks
This primitive performs the DES decipher operation on an 8-byte block of memory. The DES key is held in an 8-byte block. MULTOS ignores the parity bits.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number
Set zero, number 0xC5
Example

The following example declares 8 bytes of static memory to hold the DES Key, the plaintext is held as session data, while the resulting ciphertext will be written to public. The address for each of these is loaded onto the stack and the DES ECB Decipher primitive is called.

```asm
prmDESECBDecipher EQU 0xC5

sKey   STATIC BYTE 8 = 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08
dPlaintext DYNAMIC BYTE 8
pCiphertext PUBLIC BYTE 8

LOADA   sKey
LOADA   dPlaintext
LOADA   pCiphertext
PRIM    prmDESECBDecipher
```
**DES ECB Encipher**

This primitive performs DES ECB Encipher on an eight byte block of memory.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Syntax**

```
PRIM 0xC1
```

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyAddr</td>
<td>OutputAddr</td>
</tr>
</tbody>
</table>

Each parameter is 2 bytes in size and represents the starting address of an 8-byte block of memory.

**Arguments**

None.

**Remarks**

This primitive performs the DES encipher operation on an 8-byte block of memory. The DES key is held in an 8-byte block. MULTOS ignores the parity bits.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C: Unchanged
V: Unchanged
N: Unchanged
Z: Unchanged

**Primitive Set and Number**

Set zero, number 0xC1
Example

The following example declares 8 bytes of static memory to hold the DES Key, the plaintext is held as session data, while the resulting ciphertext will be written to public. The address for each of these is loaded onto the stack and the DES ECB Decipher primitive is called.

```
prmDESECBEncipher EQU 0xC1

sKey  STATIC BYTE 8 = 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08
dPlaintext DYNAMIC BYTE 8
pCiphertext PUBLIC BYTE 8

LOADA  sKey  
LOADA  pCiphertext
LOADA  dPlaintext
PRIM   prmDESECBEncipher
```
DivideN

This primitive performs an unsigned division of two unsigned byte blocks.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Syntax

//Stack holds block length parameter
PRIM 0x08, Length

Arguments

The argument Length expresses the number of bytes in each byte block.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Numerator</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>Quotient</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

Each parameter is of size Length.

Remarks

The argument Length is specified using a single byte. Therefore, the maximum possible length of a block is 255 bytes.

This primitive performs unsigned division of the numerator by the denominator. These values are overwritten with the resulting quotient and remainder. All of these parameters are of size Length.

If the denominator is zero, then:

- The C flag is set.
- The Z flag is unchanged.
- The data in Dynamic is unchanged.

If the denominator is non-zero, then:

- The C flag is cleared.
- The Z flag is set if the numerator is less than the denominator, and cleared otherwise.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
</table>
C    Set if the denominator is zero, cleared otherwise
V    Unchanged
N    Unchanged
Z    Set if the quotient is zero, cleared if the quotient is non-zero, remains unchanged if the
denominator is zero.

**Primitive Set and Number**

Set one, number 0x08

**Example**

The following example divides 4128 (hexadecimal 0x1020) by 256 (hexadecimal 0x0100).

```
PUSHW 0x1020
PUSHW 0x0100          // Stack = 10, 20, 01, 00
PRIM prmDivideN, 2    // Stack = 00, 10, 00, 20
                  // CCR C and Z cleared
```

The example above indicates that \((4128 \div 256) = 16\) (0x0010) with a remainder of 32 (0x0020) or,
when expressed as a fraction, \(16 \frac{32}{256}\). The result is correct as \((256 \times 16) + 32 = 4128\).
The next example reverses the previous and divides 256 by 4128.

```
PUSHW 0x0100
PUSHW 0x1020          // Stack = 01, 00, 10, 20
PRIM prmDivideN, 2    // Stack = 00, 00, 01, 00
                  // CCR C cleared, Z set
```

This new example indicates that \((256 \div 4128) = 0\) (0x0000) with a remainder of 256 (0x0100), or, as a
fraction \(0 \frac{256}{4128}\). The result is correct as \((4128 \times 0) + 256 = 256\). Here the CCR Z flag has been set
to indicate that the quotient is 0.

Division by 0 results in the CCR C flag being set and the data on the stack is left unchanged.
**ECC Addition**

This primitive adds two points on the elliptic curve specified by the supplied domain parameters.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![checkmark]  ![checkmark]  ![checkmark]  ![checkmark]  ![checkmark]  ![checkmark]

**Syntax**

PRIM 0xD0

**Arguments**

None.

**Stack Usage**

```
Stack In     domainAddr  point1Addr  point2Addr  outAddr
Stack Out    {empty}
```

All parameters are 2 bytes in size. The value held at `domainAddr` represents the elliptic curve domain parameter. Both `point1Addr` and `point2Addr` are the location of the operands. The `outAddr` is the location to which to write the result of the addition.

**Remarks**

This primitive calculates the point that is the result of the addition of two points on the elliptic curve specified by the domain parameters. If the two points are equal the primitive calculates the double of the point.

Both input points must be in the same representation, affine or projective, and the result is produced in that same representation. If the input representation is different from that of the application, then calling the primitive results in an abend.

Possible point representation values are:

0x04: Affine: X and Y values included
0x84: Projective: X, Y and Z values provided
0x0F: Affine: Use Gx and Gy values from domain parameters
0x8F: Projective: Use Gx, Gy from domain parameters with Z = 1
0x00: Affine: Use infinity as the point
0x80: Projective: Use infinity as the point

Points are structured as follows:

- BYTE representation
- BYTE X[primeLen]
- BYTE Y[primeLen]
- BYTE Z[primeLen]
Domain parameters are structures as follows:


Notes:
[1] Only supported value at present is 0x00
[2] This fixed zero byte is only required for MULTOS 4.2.1 and earlier.

If the result of the addition is infinity the Z flag is set and the representation of infinity is written to the output address specified.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Set if the result is infinity, cleared otherwise</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Primitive Set and Number**

Set zero, number 0xD0

**Example**

The following example shows how to use the ECC Addition primitive to add two points stored in Dynamic placing the result back in Dynamic and then adding the base point of the elliptic curve.

```assembly
prmEccAdd EQU 0xD0
sDomain STATIC BYTE 124 // The domain parameters for a 160 bit curve
eccBasePointAffine STATIC BYTE 0x0F // The base point in affine
//Call ECC Add to add points together
//Call ECC Add to add points together
LOADA sDomain // Load addr domain parameters
LOADA LB[0] // Load addr of first input point
LOADA LB[0x29] // Load addr of second input point
LOADA LB[0] // Load addr of output point
PRIM prmEccAdd
//Call ECC Add to add points together
//Call ECC Add to add points together
LOADA sDomain // Load addr domain parameters
LOADA LB[0] // Load addr of 1st input point
LOADA eccBasePointAffine // addr 2nd point (base point)
LOADA LB[0] // Load addr of output point
PRIM prmEccAdd
BEQ Infinity
EXIT
Infinity
```
EXITSW 0x9E, 0x20
**ECC Convert Representation**

This primitive converts the representation of an elliptic curve point.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Availability]({})

**Syntax**

PRIM 0xD1

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>domainAddr</td>
<td>pointAddr</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. The value held at `domainAddr` represents the elliptic curve domain parameters. The value found at `pointAddr` is the location of the point to convert. The `outAddr` is the location to which to write the result of the addition.

**Remarks**

This primitive converts any elliptic curve point, including the point at infinity, from affine to projective representation, or from projective to affine representation. If the input point is in affine representation the output point will be written in projective representation with a randomised Z co-ordinate.

See ECC Addition for details of domain parameters, points and point representations.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged

**Primitive Set and Number**

Set zero, number 0xD1
**Example**

The following example shows how to use the ECC Convert Representation primitive to convert a point stored in Dynamic from projective to affine representation and place the result in Public.

```assembly
prmEccConvert EQU 0xD1

sDomain STATIC BYTE 124 // The domain parameters for a 160 bit curve
//------------------------------------------------------------------
//Call ECC Convert to convert point to affine representation
// (word) Address of Domain Parameters
// (word) Address of Input Point
// (word) Address of Output Point
//------------------------------------------------------------------
LOADA sDomain        // Load addr domain parameters
LOADA LB[0]          // Load addr of input point
LOADA PB[0]          // Load addr of output point
PRIM prmEccConvert
EXITLA 0x29
```
**ECC ECIES Decipher**

This primitive performs an ECIES (Elliptic Curve Integrated Encryption Scheme) decryption of a given message.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xE9, Options

**Arguments**

The 1 byte argument *Options* is used to specify the algorithm to be used and whether the private key input will be maintained in a “protected” form.

<table>
<thead>
<tr>
<th>Options</th>
<th>Algorithm</th>
<th>Protect Private Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ECIES_KEM with DEM3 (see ISO 18033-2). Hash method is SHA-256.</td>
<td>No</td>
</tr>
<tr>
<td>0x01</td>
<td>ECIES_KEM with DEM3 (see ISO 18033-2). Hash method is SHA-512.</td>
<td>No</td>
</tr>
<tr>
<td>0x80</td>
<td>ECIES_KEM with DEM3 (see ISO 18033-2). Hash method is SHA-256.</td>
<td>Yes</td>
</tr>
<tr>
<td>0x81</td>
<td>ECIES_KEM with DEM3 (see ISO 18033-2). Hash method is SHA-512.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>domainAddr</td>
</tr>
<tr>
<td>(empty)</td>
<td></td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. *domainAddr* is the location of the elliptic curve domain parameter. *Length* is the length of the deciphered message written to location *messageAddr*. *privatekeyAddr* is the location of the private key to be used. *inputAddr* is the location of the enciphered message to be processed.

The private key is of length prime_len. The enciphered message is in the form (R, X, T) where R is prime_len x 2 bytes, X is Length bytes and T is half the hash size bytes.

**Remarks**

The Z flag is cleared on successful decipher.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Cleared if success, set if failure

**Primitive Set and Number**

Set one, number 0xE9

**Example**
**ECC ECIES Encipher**

This primitive performs an ECIES (Elliptic Curve Integrated Encryption Scheme) encryption of a given message.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xEA, Options

**Arguments**

The 1 byte argument *Options* is used to specify the algorithm to be used.

<table>
<thead>
<tr>
<th>Options</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ECIES_KEM with DEM3 (see ISO 18033-2). Hash method is SHA-256.</td>
</tr>
<tr>
<td>0x01</td>
<td>ECIES_KEM with DEM3 (see ISO 18033-2). Hash method is SHA-512.</td>
</tr>
</tbody>
</table>

**Stack Usage**

Stack In: domainAddr, Length, pubkeyAddr, messageAddr, outputAddr (empty)

Stack Out:

All parameters are 2 bytes in size. *domainAddr* is the location of the elliptic curve domain parameter. *Length* is the length of the plaintext message to be processed at location messageAddr. *publincKeyAddr* is the location of the public key to be used. *outputAddr* is the location where the enciphered message is written.

The public key consists of ecc_X followed by ecc_Y and is of length prime_len x 2. The enciphered message is in the form (R, X, T) where R is prime_len x 2 bytes, X is Length bytes and T is half the hash size bytes.

**Remarks**

The Z flag is cleared on successful encipher.

**Condition Code**

```
   C  V  N  Z
-  -  -  -  -  -  X
```
C  Unchanged
V  Unchanged
N  Unchanged
Z  Cleared if success, set if failure

**Primitive Set and Number**

Set one, number 0xEA

**Example**
ECC Elliptic Curve Diffie Hellman
This primitive performs an Elliptic Curve Diffie Hellman key agreement in accordance with ANSI X9.63.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2  MULTOS 4.5.3  MULTOS 4.5.4  MULTOS 4.5.5

Syntax

PRIM 0xE8, Options

**Arguments**

The 1 byte argument Options is used to specify whether the private key input will be maintained in a “protected” form.

<table>
<thead>
<tr>
<th>Options</th>
<th>Protect Private Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>No</td>
</tr>
<tr>
<td>0x80</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Stack Usage**

Stack In

```
domainAddr | privatekeyAddr | publickeyAddr | sharedAddr
```

Stack Out

{empty}

All parameters are 2 bytes in size. domainAddr is the location of the elliptic curve domain parameter. privatekeyAddr is the location of the private key to be used. publickeyAddr is the location of the public key to be used. sharedAddr is the location where the shared secret key is written.

The private key is of length prime_len. The public key consists of ecc_X followed by ecc_Y and is of length prime_len x 2. The shared secret key is of length prime_len.

**Remarks**

The Z flag is cleared on successful processing.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

C  Unchanged
V  Unchanged
N  Unchanged
Z  Cleared if success, set if failure

*Primitive Set and Number*

Set one, number 0xE8

*Example*
**ECC Equality Test**

This primitive tests if two points on the specified elliptic curve are equal.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- **X**  ✔  ✔  ✔  ✔  ✔

**Syntax**

PRIM 0xD2

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>domainAddr</td>
<td>point1Addr</td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. The value held at domainAddr represents the elliptic curve domain parameters. The values found at point1Addr and point2Addr are the locations of the points to test for equality.

**Remarks**

This primitive tests if two points on the elliptic curve specified by the supplied domain parameters are equal. Both input points must be in the same representation, affine or projective, or the application calling the primitive will abend.

See ECC Addition for details of domain parameters, points and point representations.

The Z flag is set to indicate that the two points are equal.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Set if the two points are equal, cleared otherwise
**Primitive Set and Number**

Set zero, number 0xD2

**Example**

The following example shows how to use the ECC Test for Equality primitive to determine if a point supplied in Public is equal to the base point of the elliptic curve.

```assembly
prmEccEqual EQU 0xD2

sDomain STATIC BYTE 124 // The domain parameters for a 160 bit curve
eccBasePointAffine STATIC BYTE 0x0F // The base point in affine

//Call ECC Equal to compare points
// (word) Address of Domain Parameters
// (word) Address of First Point
// (word) Address of Second Point

LOADA sDomain // Load addr domain parameters
LOADA eccBasePointAffine// Load addr 1st pt (base point)
LOADA PB[0] // Load addr second point
PRIM prmEccEqual
BEQ PointsEqual
EXIT
PointsEqual
EXITSW 0x9E,0x20
```
**ECC Generate Key Pair**
This primitive generates an Elliptic Curve Cryptography public and private key pair.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xEA7, Options

**Arguments**
The 1 byte argument Options is used to specify whether the private key input will be maintained in a “protected” form.

<table>
<thead>
<tr>
<th>Options</th>
<th>Protect Private Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>No</td>
</tr>
<tr>
<td>0x80</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Stack Usage**

Stack In  domainAddr  keyAddr
Stack Out  {empty}

All parameters are 2 bytes in size. domainAddr is the location of the elliptic curve domain parameter. keyAddr is the location where the key pair generated will be written.

The key pair consists of the public key followed by the private key. The public key consists of ecc_X followed by ecc_Y and is of length prime_len x 2. The private key is of length prime_len.

**Remarks**
The Z flag is cleared on successful key pair generation.

The format of the domain parameters is shown in the example below. P, A, B, Gx, Gy and N are prime_len long. The format, prime length and H are a single byte.

**Condition Code**

```
C  V  N  Z
-  -  -  -  X
```
Primitive Set and Number

Set one, number 0xE7

Example

```c
#define ECC_KEY_LEN 28   // 224 bit prime
#define PRIM_GEN_ECC_PAIR 0xE7
typedef struct
{
    BYTE x[ECC_KEY_LEN];
    BYTE y[ECC_KEY_LEN];
} ecc_public_s;
typedef struct
{
    ecc_public_s publicKey;
    BYTE privateKey[ECC_KEY_LEN];
} ecc_s;
#pragma melstatic
BYTE abDomainParams[] = {
    0x00,           // Format of domain params
    0x1C,          // Prime length in bytes
    0xAC,0x75,0xCF,0x35,0x99,0x88,0x5A,0x6A,0x26,0xB2,0x0F,0x52,0x71,0xAB,0x95,0xA3,
    0xF0,0xD2,0x4B,0x74,0x37,0x21,0x46,0xCC,0xDB,0xA0,0x5F,0xA9,          // P
    0x93,0x62,0xE8,0xF2,0x7B,0xDC,0xA9,0x6F,0xB1,0xE6,0xBF,0xA6,0x79,0x5E,0x10,0x60,
    0x9A,0x69,0x2D,0x0D,0x9F,0xE8,0x2E,0xB4,0xD8,0xE8,0xD4,0x20,        // A
    0x89,0xD8,0x66,0x9D,0x59,0x20,0x5C,0xB4,0xA3,0x6E,0xEC,0x01,0x22,0xC6,0x49,0x1C,
    0x92,0xB6,0x18,0xB8,0xFC,0x09,0xB6,0xD6,0xF3,0x24,0xAA,0xCA,        // B
    0x2E,0xDA,0x6A,0x9C,0xE8,0x53,0x3B,0xBC,0xB8,0x1D,0x49,0xF4,0x69,0xB5,0x43,0x95,
    0xD3,0x1A,0x64,0xB8,0x14,0xB8,0x92,0xB3,0x6B,0xC0,0x23,0x00,        // Gx
    0xB4,0xDA,0x69,0x9D,0xF7,0x56,0xBF,0x58,0xC9,0x50,0x76,0x7A,0xD7,0xF8,0xB4,0x62,
    0x1E,0x2F,0x5C,0xFC,0x28,0x25,0x97,0x99,0x14,0x05,0xB2,0x4D,        // Gy
    0x0F,0xAD,0x9E,0x79,0x3C,0x80,0xC2,0x66,0xBD,0xB3,0x18,0xAA,0x67,0x6C,0x9E,0xDB,
    0x4F,0xB6,0x53,0xCF,0x4F,0x67,0x92,0x37,0x13,0x37,0x56,0xA1,        // N
    0x0B          // H
};
ecc_s sEccKeyPair;

void main(void)
{
    __push (abDomainParams);
    __push (&sEccKeyPair);
    __code (PRIM, PRIM_GEN_ECC_PAIR, 0x00);

    // …etc
}
```

© 2019 MAOSCO Limited. MAO-DOC-TEC-006 v1.56
MULTOS is a registered trademark of MULTOS Limited.
**ECC Generate Signature**

This primitive generates an Elliptic Curve Cryptography signature.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xE5, Options

**Arguments**

The 1 byte argument Options is used to specify the algorithm to be used and whether the private key input will be maintained in a “protected” form.

<table>
<thead>
<tr>
<th>Options</th>
<th>Algorithm</th>
<th>Protect Private Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ECDSA</td>
<td>No</td>
</tr>
<tr>
<td>0x80</td>
<td>ECDSA</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Stack Usage**

Stack In: domainAddr | privatekeyAddr | hashAddr | sigAddr
Stack Out: {empty}

All parameters are 2 bytes in size. domainAddr is the location of the elliptic curve domain parameter. privatekeyAddr is the location of the private key to be used. hashAddr is the location of the hash code over which the signature is generated. sigAddr is the location where the signature is written.

The private key and hash code are of length prime_len. The signature produced is (R, S) and is of length 2 x prime_len.

**Remarks**

The Z flag is cleared on successful signature generation.

**Condition Code**

```
C  V  N  Z
-  -  -  -  -  -  X
```

C  Unchanged
V  Unchanged
N  Unchanged
Z  Cleared if success, set if failure

*Primitive Set and Number*

Set one, number 0xE5

*Example*
**ECC Inverse**

This primitive calculates the inverse (negation) of a point on an elliptic curve.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- **X**  ✓  ✓  ✓  ✓  ✓  ✓

**Syntax**

PRIM 0xD3

**Arguments**

None.

**Stack Usage**

Stack In
- DomainAddr
- pointAddr
- outAddr

Stack Out
- (empty)

All parameters are 2 bytes in size. The value held at domainAddr represents the elliptic curve domain parameters. The value found at pointAddr is the location of the point to convert. The outAddr is the location to which to write the result of the inversion calculation.

**Remarks**

This primitive calculates the inverse (negation) of a point on an elliptic curve.

The output point will be written in the same representation (affine or projective) as the input point.

The values 0x0F or 0x8F may be specified in the Point Representation byte of the point stored at pointAddr to indicate that the base point of the elliptic curve group is to be used as the input point. See ECC Addition for details of points and point representations.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged
**Primitive Set and Number**

Set zero, number 0xD3

**Example**

The following example shows how to use the ECC Inverse primitive with the ECC Addition primitive to subtract two points stored in Dynamic placing the result back in Dynamic.

```assembly
prmEccInv EQU 0xD3

sDomain STATIC BYTE 124 // The domain parameters for a 160 bit curve
//Call ECC Inverse to add points together
LOADA sDomain // Load addr domain parameters
LOADA LB[0x29] // Load addr of input point
LOADA LB[0x29] // Load addr of output point
PRIM prmEccInv

//Call ECC Add to add points together
LOADA sDomain // Load addr domain parameters
LOADA LB[0] // Load addr of first input point
LOADA LB[0x29] // Load addr second point
LOADA LB[0] // Load addr of output point
PRIM prmEccAdd
BEQ Infinity
EXIT
Infinity
EXITSW 0x9E, 0x20
```
ECC Scalar Multiplication
This primitive calculates a scalar multiplication of a point on the specified elliptic curve.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0xD4

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>domainAddr</td>
<td>pointAddr</td>
</tr>
<tr>
<td>(empty)</td>
<td></td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. The value held at domainAddr represents the elliptic curve domain parameters. The value found at pointAddr is the location of the input point and mAddr is the location of the multiplier. The outAddr is the location to which to write the result of the multiplication.

Remarks
This primitive performs a scalar multiplication of a point on the elliptic curve specified by the supplied domain parameters by the specified unsigned integer multiplier which is one byte longer than the length specified in the domain parameters. The result, a point on the curve, is written at the specified segment address in the same representation as the input point.

The values 0x0F or 0x8F may be specified in the Point Representation byte of the point stored at pointAddr to indicate that the base point of the elliptic curve group is to be used as the input point. See ECC Addition for details of domain parameters, points and point representations.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Set if the result is infinity, cleared otherwise

**Primitive Set and Number**

Set zero, number 0xD4

**Example**

The following example shows how to use the ECC Scalar Multiplication primitive to multiply a base point by a random number stored in Dynamic placing the result back in Dynamic.

```assembly
prmEccMult EQU 0xD4

eccBasePointAffine STATIC BYTE 0x0F // The base point in affine
prmGetRandomNumber EQU 0xC4
sDomain STATIC BYTE 124 // The domain parameters for a 160 bit curve
//---------------------------------------------
//Pad out stack
//---------------------------------------------
PUSHZ 17
//------------------------------
//Generate Random Number
//------------------------------
PRIM prmGetRandomNumber
PRIM prmGetRandomNumber
PRIM prmGetRandomNumber
//------------------------------
//Call ECC Mult
//------------------------------
LOADA sDomain // Load addr domain parameters
LOADA eccBasePointAffine // Load addr input point (base point)
LOADA DT[-25] // Load addr of 21 byte multiplier
LOADA DT[-47] // Load addr of output point
PRIM prmEccMult
POP
BEQ Infinity
EXIT
Infinity
EXITSW 0x9E, 0x20
```
**ECC Verify Point**

This primitive verifies that a point is a valid elliptic curve point.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- ✗
- ✗
- ✗
- ✗
- ✗
- ✗

**Syntax**

PRIM 0xD1, VerType

**Arguments**

The argument VerType can take one of three values depending on what type of verification is required. A value of 0x00 indicates the no group order check should be performed. A value of 0x01 means that the point should have the same order \(N\) as specified in the domain parameters. A value of 0x02 indicates that the point should not have order less than or equal to \(H\) as specified in the domain parameters.

**Stack Usage**

Stack In  | Stack Out
---|---
domainAddr | pointAddr
(empty)

All parameters are 2 bytes in size. The value held at domainAddr represents the elliptic curve domain parameters. The value found at pointAddr is the location of the point to verify. See ECC Addition for details of domain parameters, points and point representations.

**Remarks**

This primitive verifies that the specified point:

- is not infinity
- is a point on the elliptic curve defined by the specified domain parameters
- If VerType is set to 0x01: has the same order \(N\) as the group order specified in the domain parameters
- If VerType is set to 0x02: does not have order less than or equal to \(H\) the co-factor specified in the domain parameters

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
C    Unchanged  
V    Unchanged  
N    Unchanged  
Z    Set if the point is not valid, cleared otherwise

**Primitive Set and Number**

Set one, number 0xD1

**Example**

The following example shows how to use the ECC Verify primitive to verify that a point stored in Public is valid and has the same order as the base point of the elliptic curve.

```
prmEccVerify EQU 0xD1

sDomain STATIC BYTE 124 // The domain parameters for a 160 bit curve
//---------------------------------------------------------------
//Call ECC Verify to check the point
//---------------------------------------------------------------
LOADA sDomain // Load addr domain parameters
LOADA PB[0]   // Load addr input point
PRIM prmEccVerify, 0x01
BEQ Invalid
EXIT
Invalid
EXITSW 0x9E, 0x20
```
**ECC Verify Signature**

This primitive verifies an Elliptic Curve Cryptography signature.

### Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

### Syntax

PRIM 0xE6, Options

### Arguments

The 1 byte argument **Options** is used to specify the algorithm to be used.

<table>
<thead>
<tr>
<th>Options</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ECDSA</td>
</tr>
</tbody>
</table>

### Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>domainAddr</td>
<td>publickeyAddr</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. *domainAddr* is the location of the elliptic curve domain parameter. *publickeyAddr* is the location of the private key to be used. *sigAddr* is the location of the signature to be verified. *hashAddr* is the location of the hash code to be compared in the verification.

The public key consists of ecc_X followed by ecc_Y and is of length prime_len x 2. The signature is (R, S) and is of length 2 x prime_len. The hash code is of length prime_len.

### Remarks

The Z flag is cleared on successful signature verification.

### Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- C: Unchanged
- V: Unchanged
- N: Unchanged
- Z: Cleared if success, set if failure
**Primitive Set and Number**

Set one, number 0xE6

**Example**
Exchange Data
This primitive allows a MULTOS application to import data from or export data to a non-MULTOS application.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0x85

Arguments
None.

Stack Usage

Stack In
Channel
DataAddr

Stack Out
{empty}

The 1-byte Channel value identifies the non-MLTOS application with which the application wishes to exchange data. The 2-byte DataAddr is the location of data that is used to determine the direction of the exchange as well as the content to exchange. Note that the format of the data is specific to the channel.

Remarks
If parameter Channel specifies a value unknown to the implementation then the AAM will abnormally end the application.

Condition Code

C  V  N  Z
-

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number
Set zero, number 0x85
**Example**

The following example imports data from a non-MULTOS application to the address space of a MULTOS application.

```assembly
prmExchangeData EQU 0x85
SrcChannel EQU 0x01

start
  PUSHB SrcChannel // ID of exchange channel
  LOADA PB[0] // address of control data
  PRIM prmExchangeData
  EXIT // Exit
```
Exit to MULTOS and Restart

This primitive informs MULTOS that when the currently selected application exits (via the SYSTEM instruction), MULTOS should process the contents of Public as if it has been passed from an IFD rather than provide a response to the IFD. Such processing shall include the processing of a MSM command APDU placed in Public by the currently selected application (or any other APDU other than a valid SELECT command APDU). Following such processing, the response APDU from the command processing is placed in Public and the currently selected application shall be restarted.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x17

**Arguments**

None.

**Stack Usage**

Stack In: {empty}
Stack Out: {empty}

There are no input or output parameters for this primitive.

**Remarks**

This primitive sets the most significant bit, b7, of ProtocolFlags to indicate to the application that it has been restarted rather than called from the IFD. The application is responsible for clearing this bit.

As of MULTOS 4.5.4, if the calling application is currently being delegated to then it will return to the delegator before MULTOS processes the command. If the 5th (CCR5) bit of the CCR register is set, then the delegated application will not return to the delegator before MULTOS processes the command. The delegated application will then be executed after the command has been processed and will return to the delegator after it has finished executing.

This primitive will reset bit 5 to bit 8 of the CCR register after they have been checked.

**Condition Code**

```
C  V  N  Z
-  -  -  -
```
Primitive Set and Number

Set Zero, Number 0x17

Example

This is a C code fragment showing how an application could construct an OPEN MEL APP command and get MULTOS to execute it.

```c
// Construct MULTOS Open MEL APP command to run later
CLA = 0xBE;
INS = 0x12;
P1 = 0x00;
P2 = 0x00;
Lc = 0x9D;
ProtocolFlags |= 0x02; // Lc valid
ProtocolFlags &= ~0x04; // Le not used
memcpy(abPublic, abOpenMelAppData, Lc);
__c_code(PRIM, 0x17);
Exit();
```
## Flush Public

This primitive allows an application to return more bytes to the IO device than the size of the public memory area.

The application sets La to the total number of bytes to return (which is greater than the size of public), copies the first block of bytes to be returned to public then calls this primitive. The word at the top of the stack gives the number of bytes to flush to the IO from the start of public. The application calls this primitive multiple times for each block until just the last block of data is in public. When the application returns to MULTOS, MULTOS will transmit the final response data held in Public (as indicated by La) and then transmit SW12.

### Availability

MULTOS 4 | MULTOS 4.2 | MULTOS 4.3.1 | MULTOS 4.3.2 | MULTOS 4.4 | MULTOS 4.5.x
---|---|---|---|---|---
[X] | [X] | [X] | ✓ | ✓ | ✓

### Syntax

PRIM 0xEC

### Arguments

None.

### Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>BlockSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
</tr>
</tbody>
</table>

BlockSize is a 2 bytes value giving the number of bytes in public to transmit to the IO..

### Remarks

The primitive reduces La by BlockSize at the end of each call.

The primitive will cause an abend if BlockSize is larger than the size of the public memory area.

It’s important to note that if the application calls Flush Public to flush the final block of response data then MULTOS will only return SW12 when the application returns back to MULTOS and MULTOS sends the final command response.

### Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V  Unchanged
N  Unchanged
Z  Set to 1 if La > 0 following the call to this primitive.

**Primitive Set and Number**

Set Zero, Number 0xEC

**Example**

In the following fragment of ‘C’ code, the maximum public size is 1024 bytes. The application wishes to return 3200 bytes. **doFlushPublic** is a macro that calls the primitive. **buff** is an array of BYTES in static memory. **pub** is a pointer to the beginning of public memory.

La = 3200;

// Send first block
memcpy(pub,buff,1000);
doflushPublic(1000);

// Second block
memcpy(pub,buff+1000,1000);
doflushPublic(1000);

// Third block
memcpy(pub,buff+2000,1000);
doflushPublic(1000);

// Remaining bytes
memcpy(pub,buff+3000,200);
multosExit();
**Generate Asymmetric Hash General**

This primitive generates an Asymmetric Hash Digest using as input a block of memory of arbitrary size.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xC4, Mode

**Arguments**

The 1-byte argument Mode is used to indicate which stack based parameters are required.

**Stack Usage**

See the Remarks section below for details of the stack usage.

All parameters are 2 bytes in size. The possible parameters are:

- **IVAddr** indicates the location of an initial vector with a size equal to the hash chain length
- **dataLength** gives the size of the input data block
- **resAddr** is the location where the resulting asymmetric hash should be written
- **dataAddr** is the location of the input data block that is of size dataLength.
- **hmLen** indicates the size of the hash modulus supplied
- **hmAddr** indicates the location of the hash modulus supplied
- **appHcl** gives an application supplied hash chain length value

**Remarks**

For MULTOS 4 implementations the hash chain length is always 16 bytes in length. In MULTOS 4.2 this value is a fixed, platform specific value or the length can be supplied depending on the mode employed.

When an IV is not supplied, a default value is used. That value’s length is the hash chain length and each byte is 0x55. Similarly, when a hash modulus is not supplied, the hash modulus value of the platform is used. Details of this value are available by request from the KMA, but is only available to MULTOS Issuers.

The following table provides an overview of all possible Mode values, their support by MULTOS 4 and MULTOS 4.2 and what application supplied values are required. In all cases the length of data and the address of the data that serves as input must be supplied as must the result address.

<table>
<thead>
<tr>
<th>Mode</th>
<th>4?</th>
<th>4.2?</th>
<th>IV Supplied</th>
<th>HM</th>
<th>HCL Supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
For MULTOS 4 there are two *Mode* values defined, 0 and 1, that are not used in MULTOS 4.2 or later. They are included here for completeness.

A *Mode* value of 0 indicates that the default IV should be used and that no hash modulus value is supplied. The stack for this mode would then be:

```
Stack In:  dataLength | resAddr | dataAddr
Stack Out:  {empty}
```

A *Mode* value of 1 indicates that the IV is supplied and that no hash modulus value is supplied. The stack for this mode would then be:

```
Stack In:  IVAddr | dataLength | resAddr | dataAddr
Stack Out:  {empty}
```

Mode values of 2 and 3 are supported by both MULTOS 4 and MULTOS 4.2.

A *Mode* value of 2 indicates that the default IV should be used and that a hash modulus value is supplied. The stack for this mode would then be:

```
Stack In:  dataLength | resAddr | dataAddr | hmLen | hmAddr
Stack Out:  {empty}
```

A *Mode* value of 3 indicates that the IV is supplied and that a hash modulus value is supplied. The stack for this mode would then be:

```
Stack In:  IVAddr | dataLength | resAddr | dataAddr | hmLen | hmAddr
Stack Out:  {empty}
```
As of MULTOS 4.2 modes 4 and 5 have been added.

A Mode value of 4 indicates that the default IV should be used, a hash modulus value is supplied as is an application specified hash chain length. The stack for this mode would then be:

Stack In
\[\begin{array}{c}
\text{dataLength} \\
\text{resAddr} \\
\text{dataAddr} \\
\text{hmLen} \\
\text{hmAddr} \\
\text{appHcl}
\end{array}\]
\{empty\}

Stack Out
\{empty\}

A Mode value of 5 indicates that the IV, hash modulus and application specified hash chain length are supplied. The stack for this mode would then be:

Stack In
\[\begin{array}{c}
\text{IVAddr} \\
\text{dataLength} \\
\text{resAddr} \\
\text{dataAddr} \\
\text{hmLen} \\
\text{hmAddr} \\
\text{appHcl}
\end{array}\]
\{empty\}

Stack Out
\{empty\}

Any other value for Mode is undefined

If any of the required components are not present or if a combination of a component’s start address and length yields an address outside the application’s data space, the application calling this primitive will abnormally end processing.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive Set and Number**

Set one, number 0xc4

**Example**

The following code fragment shows the recommended declarations and usage for this primitive.

```
prmGenerateAHashGeneral  EQU  0xC4
```

The following example generates the asymmetrical hash of a variable sAUCode.

```
sLCode     STATIC WORD = 0x000A
sAUCode    STATIC BYTE 0x0A = "1234567890"
//Calculate the A-Hash of sAUCode
LOAD    sLCode,2
LOADA   PB[0000]
LOADA   sAUCode
PRIM    prmGenerateAHashGeneral,0
EXITLA  0x0010
```
The next example uses the default IV value, but supplies a hash modulus and hash chain length value.

```
INPUTSIZE EQU 64
HMSIZE EQU 72
APPHCL EQU 20

dDataToBeHashed DYNAMIC BYTE 64
dHashDigest DYNAMIC BYTE 20
sMyHashModulus STATIC BYTE 72

// Populate Mode 4 Stack
// size of data
PUSHW INPUTSIZE
// address of result
LOADA dHashDigest
// address of input data
LOADA dDataToBeHashed
// hash modulus length
PUSHW HMSIZE
// hash modulus address
LOADA sMyHashModulus
// application specific hash chain length
PUSHW APPHCL
// call primitive using mode 4
PRIMprmGenerateAHASHGeneral, 4
```
Generate Asymmetric Signature General
This primitive generates an asymmetric signature over a message of arbitrary length.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0xE1, Mode

Arguments

The 1-byte argument Mode indicates what type of modular exponentiation is to be used. A value of 0 indicates the full exponentiation is required, while a value of 2 indicates that the exponentiation is to be performed using Chinese Remainder Theorem (CRT).

Stack Usage

When the Mode value is set to 0, full exponentiation, the stack usage is:

Stack In

<table>
<thead>
<tr>
<th>msgLen</th>
<th>modLen</th>
<th>eAddr</th>
<th>modAddr</th>
<th>sigAddr</th>
<th>msgAddr</th>
<th>cerType</th>
<th>hmLen</th>
<th>hmAddr</th>
</tr>
</thead>
</table>

Stack Out

{empty}

The parameters are as follows:

- 2-byte msgLen indicating the length of the message to be signed
- 2-byte modLen indicating the size of the modulus used to sign the hash digest
- 2-byte eAddr indicating the address of the exponent, where the data is of size modLen
- 2-byte modAddr indicating the address of the modulus of size modLen used to sign the hash digest
- 2-byte sigAddr indicating the location where to write the resulting signature and where the data area is of size modLen
- 2-byte msgAddr indicating the location of the data to be signed, where the data is of size msgLen
- 1-byte cerType indicating the type of MULTOS certificate to produce
- 2-byte hmLen indicating the size of the hash modulus used in the calculation of the hash digest
- 2-byte hmAddr indicating the location of the hash modulus of size hmLen to use when calculating the hash digest
When the Mode value is set to 2, modular exponentiation using CRT, the stack usage is:

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgLen</td>
<td>modLen</td>
</tr>
<tr>
<td></td>
<td>dpdqAddr</td>
</tr>
<tr>
<td></td>
<td>pquAddr</td>
</tr>
<tr>
<td></td>
<td>sigAddr</td>
</tr>
<tr>
<td></td>
<td>msgAddr</td>
</tr>
<tr>
<td></td>
<td>cerType</td>
</tr>
<tr>
<td></td>
<td>hmLen</td>
</tr>
<tr>
<td></td>
<td>hmAddr</td>
</tr>
</tbody>
</table>

The parameters are as follows:

- 2-byte `msgLen` indicating the length of the message to be signed
- 2-byte `modLen` indicating the size of the modulus used to sign the hash digest
- 2-byte `dpdqAddr` indicating the address of the concatenation of `dp` and `dq`, where the data is of size `modLen / 2`
- 2-byte `pquAddr` indicating the address of the concatenation of the values `p`, `q`, and `u` each of size `modLen / 2` used to sign the hash digest
- 2-byte `sigAddr` indicating the location where to write the resulting signature and where the data area is of size `modLen`
- 2-byte `msgAddr` indicating the location of the data to be signed, where the data is of size `msgLen`
- 1-byte `cerType` indicating the type of MULTOS certificate to produce
- 2-byte `hmLen` indicating the size of the hash modulus used in the calculation of the hash digest
- 2-byte `hmAddr` indicating the location of the hash modulus of size `hmLen` to use when calculating the hash digest

In this mode it is assumed that the modulus length given in `modLen` is even. The factors `p` and `q` are prime, can both be expressed in `modLen / 2` bytes and the relationship `p < q` holds.

**Remarks**

The hashing algorithm used by this primitive is the MULTOS asymmetric hash and the certificates produced are in a MULTOS format. The format is given in the 1-byte `cerType` parameter, where a value of 3 indicates that a MULTOS 3 certificate should be produced and a value of 4 means that the certificate should be in a MULTOS 4 format.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- `C` Unchanged
- `V` Unchanged
- `N` Unchanged
- `Z` Unchanged

**Primitive Set and Number**

Set one, number 0xE1
Generate DES CBC Signature
This primitive generates an 8-byte DES CBC Signature over a block of memory.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x
✓  ✓  ✓  ✓  ✓  ✓

Syntax

//Stack Length, IVAddr, KeyAddr, SigAddr, InputAddr
PRIM 0xC6

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>IVAddr</td>
</tr>
<tr>
<td></td>
<td>KeyAddr</td>
</tr>
<tr>
<td></td>
<td>SigAddr</td>
</tr>
<tr>
<td></td>
<td>InputAddr</td>
</tr>
<tr>
<td></td>
<td>{empty}</td>
</tr>
</tbody>
</table>

All the parameters are 2-bytes in size. The parameter Length is the size of the plaintext used as input to the signature generation process. The value IVAddr is the location of an 8-byte initial vector, KeyAddr is the location of an 8-byte DES key, SigAddr is the location where the 8-byte signature is written and InputAddr is the location of data of size Length to be signed.

Remarks
This primitive uses a single 8-byte DES key and operates in CBC mode. At each step the DES encipher operation is performed.

The primitive operates only on complete 8-byte blocks in the plaintext. If Length is not an integer multiple of 8, trailing bytes are ignored. For example, if Length was 17 bytes, the 16 most significant bytes would serve as input to the algorithm and the last byte would be ignored.

The parity bits of the key are ignored.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged
Primitive Set and Number

Set zero, number 0xC6

Example

The following example generates the DES CBC Signature over the contents of sAUCode and writes it to the base of public.

```
prmGenDESCBCSignature          EQU 0xC6

sLCode   STATIC WORD    = 16
sIV      STATIC BYTE 8  = 1,2,3,4,5,6,7,8
sDESKey  STATIC BYTE 8  = 1,2,3,4,5,6,7,8
sAUCode  STATIC BYTE 16 = "1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16"

//Calculate the signature sAUCode
LOAD    sLCode,2
LOADA   sIV
LOADA   sDESKey
LOADA   PB[0000]
LOADA   sAUCode
PRIM    prmGenDESCBCSignature
EXITLA  0x008
```
Generate MAC
This primitive generates a MAC according to the required algorithm.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

Syntax
PRIM 0xC6, Algorithm

Arguments
The 1 byte argument Algorithm is used to specify the MAC algorithm as follows.


Algorithm = 1: DES CBC MAC according to ISO9797-1 algorithm 1. If the input data is a multiple of the DES block length (8-bytes) then no padding is applied. This can be used to provide an IV to the Generate MAC primitive Algorithm 0. This could also be used as a substitute for the Generate DES CBC Signature or Generate Triple DES CBC signature primitives.

Algorithm = 2: AES CMAC according to ISO9797-1 MAC Algorithm 5 and in NIST SP 800-38B.

Algorithm = 3: 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 the Generate MAC primitive Algorithm 2.

Algorithm = 4: HMAC according to ISO9797-2 MAC Algorithm 2.

Stack Usage

Algorithm 0:
Stack In  | PadByte | MsgLength | IVAddr | KeyAddr | MACAddr | MsgAddr
Stack Out | {empty}  

Algorithm 1, 2, & 3:
Stack In  | PadByte | MsgLength | IVAddr | KeyAddr | MACAddr | MsgAddr | KeyLength
Stack Out | {empty}  

Algorithm 4:
Stack In  | HashAlgo | MsgLength | KeyAddr | MACAddr | MsgAddr | KeyLength
Stack Out | {empty}  

MAO-DOC-TEC-006 v1.56 © 2019 MAOSCO Limited. MULTOS is a registered trademark of MULTOS Limited.
Where:

- **PadByte**: 1-byte parameter specifying the value of padding byte used to pad out the input message up to a multiple of the algorithm block length.
  
  Eg: for a padding byte = 0x80
  
  DES MAC: Msg | 0x80, 0x00..0x00 making the total message a multiple of 8-bytes.
  
  AES CMAC: Msg | 0x80, 0x00..0x00 making the total message a multiple of 16-bytes.

- **MsgLength**: 2-byte parameter specifying the input message length.

- **IVAddr**: 2-byte parameter specifying the location of the Initial Vector. The IV Length is dependent on the algorithm, i.e., 8-bytes for DES, 16-bytes for AES.

- **KeyAddr**: 2-byte parameter specifying the address of the key.

- **MACAddr**: 2-byte parameter specifying the address to store the result. The length of the result will be 8-bytes for a DES based algorithm, 16-bytes for AES, and dependent on the hash result length for HMAC.

- **MsgAddr**: 2-byte parameter specifying the address of the input message.

- **KeyLength**: 1-byte parameter specifying the keylength. Valid values are:
  
  o Algorithm 0: 16-byte 3DES key assumed. This parameter should not be supplied for Algorithm 0.
  
  o Algorithm 1: 8/16/24
  
  o Algorithm 2: 16/24/32
  
  o Algorithm 3: 16/24/32
  
  o Algorithm 4: upto the hash block size. If less than the hash block size is used then the key is padded with 00’s.

- **HashAlgo**: The following hash algorithms can be used for HMAC
  
  o HashAlgo = 0: SHA-1; hash block length = 64; hash result length = 20
  
  o HashAlgo = 1: SHA-256; hash block length = 64; hash result length = 32
  
  o HashAlgo = 2: SHA-512; hash block length = 128; hash result length = 64

---

**Remarks**

Consult the MULTOS Implementation Report for the device you are developing for as not all implementations support all algorithms.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C Unchanged
- V Unchanged
- N Unchanged
- Z Unchanged

**Primitive set and number**

Set one, number 0xC6
**Generate Random Prime**

This primitive generates a random prime value.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- ✗  ✔  ✔  ✔  ✔  ✔  ✔

**Syntax**

```c
//Stack holds flag, conf, timeout, rgExp, rgMinAddr, rgMaxAddr
outAddr
PRIM 0xCC
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcdFlag</td>
<td>conf</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

The most significant bit of the 1-byte parameter `gcdFlag` is set if the prime to generate must meet the condition that 3 and the prime value minus one are co-prime; i.e., the greatest common divisor of 3 and `prime -1` must be 1.

The 2-byte parameter `conf` is used to set the level of confidence that the number generated is prime. The value must be greater than zero and indicates that the probability that the result is composite is less than or equal to $2^{-\text{conf}}$. For example, if `conf` was set to 4, then the probability that the number is composite would be $2^{-4} = 1/16$.

The 2-byte parameter `timeout` is the approximate time in hundredths of a second within which the primitive should return a value. If the value is zero, then the primitive will not return until a prime number is found.

The 2-byte parameter `rgExp` is the desired length of the prime expressed in bytes, while `rgMinLoc` is the address of a 4-byte value giving the minimum value of the four most significant bytes of the modulus and `rgMaxLoc` is the address of a 4-byte value giving the maximum value of the four most significant bytes of the modulus.

The 2-byte parameter `outAddr` is the location where the generated prime of size `rgExp` is to be written.
Remarks

This primitive generates a random prime in the range

\[ [rgMin \times 256^{\text{rgExp}-4}, \, rgMax \times 256^{\text{rgExp}-4}] \].

If the parameter \( \text{conf} \) is set to zero the application calling the primitive will abnormally end if the implementation uses a probabilistic primality test.

It is expected that implementations will select a candidate number and check if it prime. If it is not, a new candidate is chosen and the process is repeated. The implementation will translate the timeout value to a maximum number of candidates to try, which must be at least one, based on an average time to check a candidate.

If no prime is generated the CCR Z flag is cleared and no value is written to \( \text{outAddr} \).

Condition Code

\[
\begin{array}{cccccc}
C & V & N & Z \\
- & - & - & - & - & X
\end{array}
\]

- C Unchanged
- V Unchanged
- N Unchanged
- Z Z is set if the prime is returned and cleared on timeout.
**Primitive Set and Number**

Set zero, number 0xCC

**Example**

The following code fragment generates a 576-bit prime for an RSA key set with a public exponent of 3. The chance that the result is not actually prime is less than 1 in a billion. The operation cancels if it takes longer than 1 minute.

```assembly
PrmGenerateRandomPrime EQU 0xCC

abyPrime STATIC BYTE 72
abyRgMin STATIC BYTE 4 = { 0x80, 0x00, 0x00, 0x00 }
abyRgMax STATIC BYTE 4 = { 0xFF, 0xFF, 0xFF, 0xFF }

//------------------------------
//Call generate random prime
// (byte) flag
// (word) conf confidence of prime being composite
// (word) timeout approximate maximal time to search
// (word) length of prime
// (word) Address of abyRgMin
// (word) Address of abyRgMax
// (word) Address of abyPrime (output)
//------------------------------

PUSHB 0x80        //gcd(3, prime-1)=1
PUSHW 30          //confidence 10-9 ≈ 2-30
PUSHW 6000        //timeout
PUSHW 64
LOADA abyRgMin   //Address of rgMin
LOADA abyRgMax   //Address of rgMax
LOADA abyPrime   //Address of result
PRIM prmGenerateRandomPrime
```
**Generate RSA Key Pair**

This primitive generates an RSA key pair for application usage.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- ✗  ✓  ✓  ✓  ✓  ✓

**Syntax**

PRIM 0xE0, Method, Mode

**Arguments**

The 1-byte parameter method specifies the algorithm used to generate the key pair.

- 0x00: The key pair is generated using the default method defined by the MULTOS implementation. This method and algorithm used is decided upon by the MULTOS Implementer to ensure quality of keys generated and performance. This method may use proprietary mechanisms dependent upon the hardware platform and cryptographic co-processor features.

- 0x01: The key pair is generated using the method defined by [X9.31].

- 0x80: The protected key pair is generated using the default method.

- 0x81: The protected key pair is generated using the method defined by [X9.31].

The 1-byte parameter mode specifies the manner in which the key pair will be generated. This parameter applies only to method 0x00. For method 0x01, any value is ignored.

- 0x00: Performance. A key pair will be generated in a manner that will optimise performance of the generation process.

- 0x01: Balanced. A key pair will be generated in a manner that balances performance against confidence in the prime numbers used to generate the key pair.

- 0x02: Confidence. A key pair will be generated in a manner that maximises the confidence in the prime numbers used to generate the key pair.

**Stack Usage**

Stack In

- keyLen
eLen
eAddr
dpdqpuAddr
mAddr
mLen

Stack Out

- (empty)

- The 2-byte parameter keyLen is the length in bytes of the key to be generated.
- The 2-byte parameter eLen is the length of the public exponent.
- The 2-byte parameter eAddr is the segment address of the public exponent.
- The 2-byte parameter dpdqpuAddr is the segment address of dp concatenated to dq, p, q and u.
- The 2-byte parameter mAddr is the segment address of the modulus, if to be returned.
- The 2-byte parameter mLen is the length of the modulus (equal to keyLen), or zero if the modulus is not to be returned.
Remarks

This primitive calculates an RSA key pair.
Implementations may only support a selection of method and mode arguments. Please see the MIR
for details.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set if the generation of the key pair fails, cleared if the generation of the key pair succeeds.
V  Unchanged
N  Unchanged
Z  Unchanged.

Primitive Set and Number

Set Two, number 0xE0

Example

The following example uses the Generate RSA Key Pair primitive to generate a 1024-bit / 128-byte
key pair.

```
prmGenRSAKeyPair   EQU 0xE0

sMod   STATIC BYTE
sDPDQ  STATIC BYTE 96
sP     STATIC BYTE 32
sQ     STATIC BYTE 32
sU     STATIC BYTE 32
sBase  STATIC BYTE 64

//Call primitive to protect the keys
//  (word) Length of Modulus
//  (word) Address of dp|dp
//  (word) Address of p|q|u
//  (word) Address of dp|dq
//  (word) Address of p|q|u

PUSHW 0x0080   //Length of key to be generated
PUSHW 0x0002   //Length of public exponent length
LOADA sE      //Address of public exponent
LOADA sDPDQPQU //Address of dp|dq|p|q|u
LOADA sM      //Address of modulus
PUSHW 0x0080   //Length of Modulus
PRIM   prmGenRSAKeyPair  // call primitive
```

//-------------------------------------------------------------------------------
**Generate Triple DES CBC Signature**

This primitive generates an 8-byte Triple DES CBC Signature over a block of memory.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmark]

**Syntax**

```c
//Stack Length, IVAddr, KeyAddr, SigAddr, InputAddr
PRIM 0xC7
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Length</th>
<th>IVAddr</th>
<th>KeyAddr</th>
<th>SigAddr</th>
<th>InputAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All the parameters are 2-bytes in size. The parameter Length is the size of the plaintext used as input to the signature generation process. The value IVAddr is the location of an 8-byte initial vector, KeyAddr is the location of two 8-byte DES key, SigAddr is the location where the 8-byte signature is written and InputAddr is the location of data of size Length to be signed.

**Remarks**

This primitive uses two 8-byte DES keys and operates in CBC mode. The 16-byte key value assumes that the most significant 8 bytes are “Key 1” and the least significant 8 bytes are “Key 2”. At each step the DES operations performed are: encipher using the Key 1, decipher using Key 2, encipher using Key 1.

The primitive operates only on complete 8-byte blocks in the plaintext. If Length is not an integer multiple of 8, trailing bytes are ignored. For example, if Length was 17 bytes, the 16 most significant bytes would serve as input to the algorithm and the last byte would be ignored.

The parity bits of the key are ignored.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged

V  Unchanged
Primitive Set and Number

Set zero, number 0xC7

Example

The following example generates the DES CBC Signature over the contents of sAUCode and writes it to the base of public.

```
prmGenTripleDESCBCSignature     EQU 0xC7
LCODE    EQU 16

sIV      STATIC BYTE 8  = 1,2,3,4,5,6,7,8
sDESKeys  STATIC BYTE 8  = 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
             0x07, 0x08,
             0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
sAUCode  STATIC BYTE 16 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
             13, 14, 15, 16

//Calculate the signature sAUCode
PUSHW   LCODE
LOADA   sIV
LOADA   sDESKeys
LOADA   PB[0000]
LOADA   sAUCode
PRIM    prmGenTripleDESCBCSignature
EXITLA  0x008
```
Get Configuration Data

This primitive allows applications to access exactly the same configuration as can be accessed via the Get Configuration Data APDU command.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

Syntax

PRIM 0x15

Arguments
None.

Stack Usage

Stack In  outAddr  Token
Stack Out  bytesRead

All the parameters are two bytes in size.

Token takes the same values defined in the Remarks table for the Get Configuration Data command. outAddr points to a buffer to contain the requested data. bytesRead returns the number of bytes written by the primitive to outAddr or zero if an error condition occurred (invalid token or attempt to write to invalid address).

Condition Code

C  V  N  Z
-  -  -  -  -

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number

Set zero, number 0x15
**Get AID**
This primitive gets the AID of the calling application or any other loaded application.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.x</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.1 / 2</th>
<th>MULTOS 4.5.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xDD

**Arguments**
None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dest</td>
<td>AppNumber</td>
</tr>
<tr>
<td>Result</td>
<td></td>
</tr>
</tbody>
</table>

The one-byte AppNumber specifies the number of the application to get the AID of. An application number of zero refers to the executing application.

The two-byte Dest defines the destination address of the 17-byte AID (one byte length followed by a 16-byte body).

The one-byte Result holds the result of the operation: 0 indicates that 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 Result equals 1.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged.
V Unchanged
N Unchanged
Z Unchanged

**Primitive set and number**
Set zero, number 0xDD
Get Available Interface Types

This primitive returns information on the interfaces supported by the MCD.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.x</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.1 / 2 / 3</th>
<th>MULTOS 4.5.4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax

PRIM 0x17, type

Arguments

type (a single byte) specifies the type of interfaces as follows.

- 0x00 - standard interfaces
- 0x01 - proprietary interfaces
- All other values - RFU

Stack Usage

Stack In: {empty}
Stack Out: SupportedInterfaces

Remarks

SupportedInterfaces is a 16-bit bit field that indicates which interfaces are supported by the MULTOS platform as follows.

- type = 0x00 (standard interfaces)
  - bit 0: 1 = contact ISO smartcard interface supported, 0 = not supported
  - bit 1: 1 = contactless ISO smartcard interface supported, 0 = not supported
  - bits 2-15: RFU
- type = 0x01 (proprietary interfaces)
  - bits 0-15: implementation-specific.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C
Unchanged.

V
Unchanged

N
Unchanged

Z
Unchanged

Primitive set and number

Set one, number 0x17
Get Data

This primitive retrieves the Data Objects (DO) of a generic MAOS device. Specifically for MULTOS, this command returns data objects to identify the platform type and other objects as agreed with Global Platform.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Syntax

```
//Stack holds Addr parameter
PRIM 0x87, ReadLength
```

Arguments

The value ReadLength specifies the maximum number of bytes to read from the Data.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>outAddr</td>
<td>BytesRead</td>
</tr>
</tbody>
</table>

The 2-byte parameter outAddr indicates the location where the data returned is to be written. While the 1-byte BytesRead values indicates the total number of bytes actually read.

Remarks

The ReadLength value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if ReadLength is zero.

The Data Object is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested and the actual length of the Data. The number of bytes copied is returned on the stack. The exact effect of this primitive is undefined if the destination area includes the top one or two bytes of the stack.

This primitive allows an application to obtain the Data Objects of the MCD. The data structure returned by this primitive is given as part of the ‘Get Data’ command in the ‘APDU Commands’ section.

Condition Code

```
C V N Z
- - - X - - -
```
C  Set if data retrieved was less than requested, cleared otherwise
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive Set and Number**

Set one, number 0x87
Get Delegator AID
This primitive permits an application to ascertain the Application ID of the application that delegated to it, if any.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓ ✓ ✓ ✓ ✓ ✓

Syntax
//Stack holds AIDAddr parameter
PRIM 0x81, ReadBytes

Arguments
The argument ReadBytes indicates the maximum number of bytes of the AID to write.

Stack Usage
Stack In    AIDAddr
Stack Out   {empty}

The 2-byte parameter AIDAddr holds the address where ReadBytes number of the Application ID is to be written.

Remarks
The ReadLength value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if ReadLength is zero.

The AID, preceded by its length expressed as a byte, is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested or the actual length of the AID plus one for the length byte.

If the application calling the primitive was not delegated to, then the CCR Z flag is set.

Condition Code

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set if data retrieved was less than requested, cleared otherwise
V  Unchanged
N  Unchanged
Z  Set if there is no Delegator, cleared otherwise
**Primitive Set and Number**

Set one, number 0x81

**Example**

The following example checks that the application has been delegated to by another application, and that the AID of that application is 0xF000000000000002; otherwise the application exits.

```
prmGetDelegatorID   EQU 0x81
sAID     STATIC BYTE 8 = 7,0xF0,0,0,0,0,0,2
dTemp    DYNAMIC BYTE 8

LOADA  dTemp
PRIM   prmGetDelegatorID,0x08
JEQ    ValidAID
LOAD   sAID,7
CMPN   dTemp,7
JNE    InvalidAID
JMP    Continue

ValidAID
EXIT

InvalidAID
EXIT

Continue
  //Continue processing
```
Get DIR File Record

This primitive retrieves a record from the Directory File, also referred to as the DIR File, stored in a root directory of the MULTOS device.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Syntax

// Stack holds Addr, RecNo parameters
PRIM 0x09, ReadLength

Arguments

The argument `ReadLength` specifies the number of bytes to read from the DIR File record.

Stack Usage

Stack In

<table>
<thead>
<tr>
<th>Addr</th>
<th>RecNo</th>
</tr>
</thead>
</table>

Stack Out

<table>
<thead>
<tr>
<th>Addr</th>
<th>RecNo</th>
<th>BytesRead</th>
</tr>
</thead>
</table>

The 2-byte parameter `Addr` indicates the address where the retrieved DIR File record should be written. The 1-byte parameter `RecNo` is the record number to be retrieved and `BytesRead` is the actual number of bytes read from the DIR File record.

A `RecNo` of zero indicates the current application’s DIR file record (supported from MULTOS 4.5).

Remarks

The `ReadLength` value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if `ReadLength` is zero.

The DIR File record numbers are indexed from 1.

The DIR file record is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested and the actual length of the record.

If a record does not exist the CCR Z flag is set.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>
C  Set if data retrieved was less than requested, cleared otherwise.
V  Unchanged
N  Unchanged
Z  Set if the specified record does not exist, cleared otherwise

**Primitive Set and Number**

Set one, number 0x09

**Example**

The following example reads in the whole of the first DIR File record into the base of public and sets La to the number of bytes read.

```assembly
prmGetDIRFileRecord EQU 0x09  
pLa EQU PT[-4]

LOADA PB[0000]  // Load address of public base to stack
PUSHB 1           // We want record no 1

PRIM prmGetDIRFileRecord, 64

STORE pLa,1      // Copy bytes read into La
```
Get FCI State
This primitive returns whether the currently selected application has a normal or dual FCI.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

Syntax
PRIM 0x87

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>{empty}</td>
<td>Result</td>
</tr>
</tbody>
</table>

Result is one byte and holds the result of the operation as follows:

- 0 = The executing application has a normal FCI
- 1 = The executing application has a dual FCI

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive set and number
Set zero, number 0x87
**Get File Control Information**
This primitive retrieves the File Control Information corresponding to an application loaded onto a MULTOS device.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmarks]

**Syntax**

```c
// Stack holds Addr,RecNo parameter
PRIM 0x0A, ReadLength
```

**Arguments**

The argument `ReadLength` specifies the number of bytes to read from the FCI record.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>RecNo</td>
</tr>
<tr>
<td>Addr</td>
<td>RecNo</td>
</tr>
</tbody>
</table>

The 2-byte parameter `Addr` indicates the address where the retrieved FCI record should be written. The 1-byte parameter `RecNo` is the record number to be retrieved and `BytesRead` is the actual number of bytes read from the FCI record.

A `RecNo` of zero indicates the current application’s DIR file record (supported from MULTOS 4.5).

**Remarks**

The `ReadLength` value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if `ReadLength` is zero.

The FCI record numbers are indexed from 1.

The FCI record is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested and the actual length of the record.

If a record does not exist the CCR Z flag is set.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>
```

© 2019 MAOSCO Limited. MAO-DOC-TEC-006 v1.56
MULTOS is a registered trademark of MULTOS Limited.
C  Set if data retrieved was less than requested, cleared otherwise.
V  Unchanged
N  Unchanged
Z  Set if the specified record does not exist, cleared otherwise

**Primitive set and number**

Set one, number 0x0A

**Example**

The following example reads in the whole of the first FCI record into the base of public and sets pLa to the number of bytes read.

```assembly
prmGetFCIControlInformation EQU 0x0A
pLa EQU PT[-4]

LOADA PB[0000]  //Load address of public base to stack
PUSHB 1           //We want record no 1
PRIM  prmGetFCIControlInformation, 64
STORE pLa, 1     //Copy bytes read into La
```
Get Manufacturer Data

This primitive retrieves the Manufacturer Data of a MULTOS device.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

// Stack holds Addr parameter
PRIM 0x0B, ReadLength

Arguments

The argument ReadLength specifies the number of bytes to read from the manufacturer data returned.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Out</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesRead</td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter Addr holds the address where the retrieved bytes of Manufacturer Data should be written. The 1-byte parameter BytesRead is the actual number of Manufacturer Data bytes read.

Remarks

The ReadLength value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if ReadLength is zero.

The Manufacturer Data is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested and the actual length of the data returned. Note that the exact effect of this primitive is undefined if the destination area includes the top one or two bytes of the stack.

The structure of the data returned is explained in the ‘APDU Commands’ section under ‘Get Manufacturer Data’.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C   Set if data retrieved was less than requested, cleared otherwise
V   Unchanged
N   Unchanged
Primitive Set and Number

Set one, number 0x0B

Example

The following example gets the Manufacturer Data, writes it to the base of public and sets La to the number of bytes read.

```
prmGetManufacturerData EQU 0x0B
pLa EQU PT[-4]

LOADA  PB[0000]   //Load public base address to stack
PRIM   prmGetManufacturerData, 22
STORE  pLa,1      //Copy bytes read into La
```

The following example defines offsets within the returned data and extracts the 6-byte MCD ID from the data held in dynamic memory. That value is then copied to public.

```
EXPECTEDLENGTH EQU 22
IC_Manufacturer_ID EQU 0x0000
IC_Type EQU 0x0001
ROM_IC_Details EQU 0x0003
MCD_ID EQU 0x0005
Initialisation_Date EQU 0x000B
Processor_Page_Size EQU 0x0012
Max_Tx_TPDU_Size EQU 0x0013
Max_Rx_TPDU_Size EQU 0x0015
prmMemoryCopy EQU 0x0C

// stack empty
LOADA  DB[0000]   // write data to stack
PRIM   prmGetManufacturerData, EXPECTEDLENGTH
// copy MCD_ID to Public via Memory Copy Primitive
PUSHW  6         // length of MCD_ID
LOADA  PB[0000]   // destination address
LOADA  DB[MCD_ID] // source address
PRIM   prmMemoryCopy
// exit setting La to length of MCD ID
EXITLA 6
```
**Get Memory Reliability**

This primitive requests the status of the current reliability of the non-volatile memory within the MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x09

**Arguments**

None.

**Stack Usage**

None.

**Remarks**

The non-volatile memory technology currently used in smart cards Electrically Erasable Programmable Read-Only Memory, sometimes referred to as EEPROM, has finite life and reliability may vary slightly from manufacturer to manufacturer. This primitive adds an additional layer of safety for end-of-card handling on devices that monitor memory reliability. Detecting that memory is only marginally reliable or unreliable allows applications to handle such situations. The actual mechanism used to monitor memory reliability varies from implementation to implementation.

There are three possible states. They are:

- Memory is reliable: C and Z are both cleared.
- Memory is marginally reliable: C is cleared and Z is set.
- Memory is unreliable: C is set and Z is cleared.

All MULTOS implementations support this primitive. However, not all implementations monitor memory reliability. In those cases were memory reliability is not monitored the implementation will always return the response, "memory is reliable".

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

C  See Remarks for information on interpreting the value of this flag.
V  Unchanged


N Unchanged
Z See Remarks for information on interpreting the value of this flag.

**Primitive Set and Number**

Set Zero, number 0x09

**Example**

The following example calls the Get Memory Reliable primitive and returns a response in SW according to the current level of reliability. This could typically be used either as a specific command, or as a self-check performed by the application.

```assembly
prmGetMemoryReliabLe field EQU 0x09

PRIM  prmGetMemoryReliable
BEQ   errMemMarginal
BLT   errMemUnreliable

MemReliable
  //Memory is reliable
  EXIT

errMemMarginal
  //Memory is marginally reliable
  EXITSW 0x65, 0x01

errMemUnreliable
  //Memory is unreliable
  EXITSW 0x65, 0x02
```
Get MULTOS Data
This primitive retrieves the MULTOS Data of a MULTOS device.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

// Stack holds Addr parameter
PRIM 0x0C, ReadLength

Arguments
The argument ReadLength specifies the number of bytes to read from the manufacturer data returned.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>BytesRead</td>
</tr>
</tbody>
</table>

The 2-byte parameter Addr holds the address where the retrieved bytes of Manufacturer Data should be written. The 1-byte parameter BytesRead is the actual number of Manufacturer Data bytes read.

Remarks
The ReadLength value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if ReadLength is zero.

The MULTOS Data is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested and the actual length of the data returned. Note that the exact effect of this primitive is undefined if the destination area includes the top one or two bytes of the stack.

The structure of the data returned is explained in the ‘APDU Commands’ section under ‘Get MULTOS Data’.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

C  Set if data retrieved was less than requested, cleared otherwise
V  Unchanged
Primitive Set and Number

Set One, number 0x0C

Example

The following example reads in the first 10 bytes of the MULTOS Data into the base of public and sets La to the number of bytes read.

```plaintext
prmGetMULTOSData fieldEQU 0x0C
PLa   EQU PT[-4]

LOADA   PB[0000]   //Load address of public base to stack
PRIM    prmGetMULTOSData, 10  
STORE   pLa,1//Copy bytes read into La
```
Get PIN Data

Gets data relating to the PIN which is either the local application PIN or the Global PIN depending on the access_list bit settings in the ALC. See Initialise PIN for details.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Availability Icon]

Syntax

PRIM 0x86, ElementId

Arguments

ElementId can take the following values:
- 0x00: PIN Try Counter
- 0x01: PIN Try Limit
- 0x02: PIN Status
- 0x03: PIN Verification Status (new in MULTOS 4.5.2)

Stack Usage

Stack In {empty}
Stack Out [Value]

Value is the one byte value of the PIN data element selected.

PIN Status has the values
- 0x00 = PIN not initialize
- 0x01 = PIN has been initialized with pin_access_level = 01
- 0x02 = PIN has been initialized with pin_access_level = 00,10 or 11
- 0x03 = PIN status error, implementer specific.

PIN Verification Status has the values
- 0x5A = PIN is unverified
- 0xA5 = PIN is verified

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged
Primitive Set and Number

Set one, number 0x86
Get Process Event

The Get Process Event primitive can be called by any application to get the number of the application process event that caused the application to be executed by MULTOS. See the [MDG] for a description of Application Process Events.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0xE8

Stack Usage

Stack In: {empty}
Stack Out: EventID

Remarks

This primitive returns the id of the Application Process Event. Valid values are between 0 and 6 where 0 is the default. 0 is the only possible value for applications that do not have the required access_list bit set.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Unchanged

Primitive Set and Number

Set Zero, Number 0xE8
Get Purse Type
This primitive returns a value indicating the type of Mondex Purse that the device can support.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
// Stack holds Addr parameter
PRIM 0x0D, ReadLength

Arguments
The argument ReadLength specifies the number of bytes to read from the manufacturer data returned.

Stack Usage
Stack In  Addr
Stack Out  BytesRead

The 2-byte parameter Addr holds the address where the retrieved bytes of Purse Type information should be written. The 1-byte parameter BytesRead is the actual number of bytes read.

Remarks
The ReadLength value is specified using a single byte. Therefore, the maximum length of a returned data is 255 bytes. Note that the effect of the primitive is undefined if ReadLength is zero.

The purse type data is copied to the segment address specified by the application. The number of bytes copied is the lesser of the number requested and the actual length of the data returned. Note that the exact effect of this primitive is undefined if the destination area includes the top one or two bytes of the stack.

The structure of the data returned is explained in the ‘APDU Commands’ section under ‘Get Purse Type’.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set if data retrieved was less than requested, cleared otherwise
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive Set and Number**

Set One, number 0x0D
Get Random Number
This primitive places an eight byte random number onto the stack.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0xC4

Arguments
None.

Stack Usage
Stack In  {empty}
Stack Out  Bytes

The output parameter Bytes holds the 8-byte block of random data returned by the primitive.

Remarks
The method of random number generation is implementation specific. So, it may be generated using a hardware assisted ‘true’ random number generator or it may be generated as a pseudo-random number from a seed value. In either case, the process is performed in such a way that the secrecy of the number is guaranteed. It is not possible for any coresident application to determine what number was provided or will be provided subsequently.

Condition Code

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number
Set zero, number 0xC4
Example

The following example calls the Get Random Number primitive and stores the eight byte random number in a variable called sDESKey

```asm
prmGetRandomNumber   EQU 0xC4
sDESKey      STATIC BYTE 8

PRIM prmGetRandomNumber
STORE sDESKey, 8
EXIT
```
**Get Replaced Application State**

This primitive returns the state of the application that the currently executing application is replacing, if any.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>4.3.x</td>
<td>4.4</td>
<td>4.5.1 / 2</td>
<td>4.5.3</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xBD

**Arguments**

None.

**Stack Usage**

Stack In  {empty}
Stack Out  State

State is one byte and holds the state of the replaced application as follows:

- 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)

**Condition Code**

C  V  N  Z
-  -  -  -

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive set and number**

Set one, number 0xBD
Get Session Size
This primitive returns the size of the application's session data. It is useful for applications that intend to call the Update Session Size primitive.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.x</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.1 / 2</th>
<th>MULTOS 4.5.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0x03

Arguments

None

Stack Usage

Stack In {empty}
Stack Out SessionSize

The primitive returns the two-byte value SessionSize being the size of the application's session data.

Condition Code

C V N Z
- - - - - - - -

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive set and number

Set zero, number 0x03
Get Static Size

This primitive returns the total size of the application’s Static memory.

Availability

MULTOS 4   MULTOS 4.2   MULTOS 4.3.1   MULTOS 4.3.2   MULTOS 4.4   MULTOS 4.5.x

Syntax

PRIM 0xDF, Options

Arguments

The 1 byte argument Options is used to specify the size of the returned Static size as follows.

- Options = 0: 32-bit (4-byte) Static size returned.
- Options = 1: 64-bit (8-byte) Static size returned.

Stack Usage

Stack In {empty}
Stack Out StaticSize

StaticSize specifies the total size of Static in bytes. StaticSize can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the value of Options.

Remarks

The Z flag is cleared to zero if the number of Static bytes is too large to be held in the returned StaticSize, otherwise it is set to 1.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Set if no overflow occurred, cleared otherwise.

Primitive set and number

Set one, number 0xDF
**Get Transaction State**

This primitive returns whether transaction protection is currently enabled.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

![Availability Icon]

**Syntax**

PRIM 0x16

**Arguments**

None.

**Stack Usage**

Stack In   {empty}
Stack Out   Result

Result is one byte and holds the result of the operation as follows:

- 0 = Transaction protection off
- 1 = Transaction protection on

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive set and number**

Set zero, number 0x16
**GSM Authenticate**

This primitive performs the A3A8 algorithm (the default algorithm implemented by MULTOS, if supported, is the Comp-128 version 2 algorithm) that is used by a SIM application whilst authenticating to the GSM network.

It is possible that a network-specific algorithm may replace the standard Comp-128 version 2 algorithm by the loading of a network-specific AMD into the MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

```
PRIM 0xCB
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDAddr</td>
<td>KeyAddr</td>
</tr>
<tr>
<td>sreskcAddr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{empty}</td>
</tr>
</tbody>
</table>

The 2 byte parameter RANDAddr is the starting address of the 16-byte random challenge to be used. The 2 byte parameter KeyAddr is the starting address of the 16-byte key to be used. The 2 byte parameter sreskcAddr is the starting address of the 12-byte result containing the 4-byte SRES and 8-byte Kc values.

**Remarks**

This primitive performs the Comp-128 version 2 algorithm as standard but may be replaced by an alternative network-specific algorithm.

This primitive is only available to an application if “GSM Authenticate” is set on in the application’s access_list when loaded. This permits a network to restrict the use of the algorithm by third-party applications.

**Condition Code**

```
C  V  N  Z
--- --- --- ---
-   -   -   -
```

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive Set and Number**

Set zero, number 0xCB

**Example**

The following example declares 16 bytes of static memory to hold the 16 byte Key, the RAND, SRES and Kc values are held in session. The address for each of these is loaded onto the stack and the GSM Authenticate primitive is called.

```
prmGSMAuthenticate  EQU  0xCB

sKey  STATIC BYTE 16 =
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D
 ,0x0E,0x0F,0x10
dsreskc DYNAMIC BYTE 16
dRAND  DYNAMIC BYTE 16

LOADA   dRAND
LOADA   sKEY
LOADA   dsreskc
PRIM    prmGSMAuthenticate
```
Initialise PIN

This primitive initialises either the application’s PIN or the Global PIN, depending on the `access_list` bits of the ALC.

<table>
<thead>
<tr>
<th>bit9*</th>
<th>bit8*</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Application PIN / Full access</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Global PIN / Basic access</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Global PIN / Write access</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Global PIN / Full access</td>
</tr>
</tbody>
</table>

* Indexed from bit0

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xE5

Arguments

None.

Stack Usage

Stack In

Stack Out

The 2 byte parameter `InitDataAddr` is the start address of the following data block:

a) PIN Reference Data (8 bytes)
b) PIN Length (1 byte)
c) PIN Try Counter (1 byte)
d) PIN Try Limit (1 byte)
e) Checksum (4 bytes).

Checksum is the MULTOS Checksum calculated over fields a-d inclusive.

Remarks

An application is only allowed to Initialise the Global PIN in “Global Basic” mode when it has not been already initialised.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive Set and Number**

Set zero, number 0xE5

**Example**

typedef struct
{
    BYTE amPinValue[8];
    BYTE bPinLength;
    BYTE bPinTryCounter;
    BYTE bPinTryLimit;
    BYTE amChecksum[4];
} INITIALISE_PIN_PARAMETERS;

#pragma melstatic
INITIALISE_PIN_PARAMETERS params;

// Default PIN value 1234
BYTE defaultPIN[4] = { 0x00, 0x01, 0x02, 0x03 };

void main(void)
{
    //... set params to required values
    memcpy(params.amPinValue,defaultPIN,4);
    params.bPinLength = 4;
    params.bPinTryCounter = 3;
    params.bPinTryLimit = 3;

    // Calculate checksum
    multosChecksum(11, params, params.amChecksum);

    // Call primitive
    __push (__typechk (INITIALISE_PIN_PARAMETERS *, params));
    __code (__PRIM, 0xE5);

    //...
**Initialise PIN Extended**

This is similar to *Initialise PIN* but allows for alternative PIN data block formats.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

![Checkmark](image)

**Syntax**

PRIM 0xE4, ElementId

**Arguments**

ElementId can take the following values:

0x00: Initialise Pin Extended

**Stack Usage**

Stack In  InitDataAddr
Stack Out  \{empty\}

The 2 byte parameter InitDataAddr is the start address of either the following data blocks:

- a) PIN Length (1 byte)
- b) PIN Reference Data (PIN Length bytes)
- c) PIN Try Counter (1 byte)
- d) PIN Try Limit (1 byte)
- e) Checksum (4 bytes).

Checksum is the MULTOS Checksum calculated over fields a-d inclusive.

**Remarks**

An application is only allowed to Initialise the Global PIN in “Global Basic” mode when it has not been already initialised.

**Condition Code**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>V</td>
<td>N</td>
<td>Z</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive set and number**

Set one, number 0xE4
**Load CCR**

This primitive pushes the Condition Code register onto the stack.

**Availability**

MULTOS 4   MULTOS 4.2   MULTOS 4.3.1   MULTOS 4.3.2   MULTOS 4.4   MULTOS 4.5.x

![checkmark](image)

**Syntax**

PRIM 0x05

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>{empty}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>CCR</td>
</tr>
</tbody>
</table>

The 1-byte output parameter CCR holds a byte whose value is the same as that of the CCR.

**Remarks**

This primitive pushes one byte to the stack that contains the same bit settings as the Condition Code Register.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

**Primitive set and number**

Set Zero, Number 0x05

**Example**

The following example performs an operation which will have different results on a MULTOS device that supports signed arithmetic than one which does not. It then loads the Condition Control register.
onto the stack and then performs a bit manipulation to determine if bit 1, the Negative Flag, is set. The code jumps to a label called exitSigned if it is.

```assembly
prmLoadCCR   EQU 0x05
prmAND EQU 0xC0  // 00000011b
prmCMP EQU 0x00  // 00000000b

PUSHB 0x00
PUSHB 0x01
SUBN ,1
PRIM prmLoadCCR
PRIM prmBitManipulateByte, prmAND + prmCMP ,0x06
BNE   isSigned

isNotSigned
   //The MULTOS device does not support signed arithmetic
isSigned
   //The MULTOS device supports signed arithmetic
```
Lookup
This primitive searches a byte array for the first instance of a specific byte value.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

// Stack holds ByteValue and Address of the search array
PRIM 0x0A

Arguments
None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteValue</td>
<td>ArrayAddr</td>
</tr>
<tr>
<td>Offset</td>
<td></td>
</tr>
</tbody>
</table>

The 1-byte parameter ByteValue is the value for which to search within the array. The 2-byte parameter ArrayAddr is the location of the array to be searched. Finally, the 1-byte output parameter Offset is the location within the array where the first instance of ByteValue was found.

Remarks
The primitive expects that the first byte of the search array indicates the total length in bytes of the remainder of the array. For example,

Length, Byte1, Byte2, … ByteN

where the value of Length is N.

The value Offset returned from this primitive is zero based and, continuing from the example above, counting begins with Byte1.

The CCR Z flag is set to indicate if an instance of ByteValue has been found.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged
N Unchanged
Z Set if the byte is found in the array, cleared otherwise

**Primitive Set and Number**

Set Zero, Number 0x0A

**Example**

The following example searches for the first instance of the byte 0x02 in the array.

```
prmLookup   EQU 0x0A

// Declare the array
// Number of bytes in the array
// Byte Values for the array
sArray      STATIC BYTE 5 = 0x04, 0x01, 0x02, 0x03, 0x55

PUSHB 0x02 // byte value to find
LOADA sArray // address of the array
PRIM prmLookup

// Stack now equals: 0x01
```

To cater for the case where a value is not found, the following could be used:

```
PUSHB 0xFF // byte value to find
LOADA sArray // address of the array
PRIM prmLookup
// CCR Z flag is cleared if the value is not found
JNE Not_Found

Not_Found
// handling here
```
**Lookup Word**

This primitive searches a byte array for the first instance of a specific 2 byte value.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

```
✘  ✘  ✘  ✔  ✔  ✔
```

**Syntax**

// Stack holds WordValue and Address of the search array
PRIM 0x14

**Arguments**

None

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordValue</td>
<td>Offset</td>
</tr>
<tr>
<td>ArrayAddr</td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter *WordValue* is the value for which to search within the array. The 2-byte parameter *ArrayAddr* is the location of the array to be searched. Finally, the 2-byte output parameter *Offset* is the location within the array where the first instance of *WordValue* was found (or the first instance of either a LSB or MSB match of *WordValue* if no full match was found).

**Remarks**

The primitive expects that the first word of the search array indicates the number of words in the remainder of the array. For example,

```
Length, Word1, Word2, ... WordN
```

where the value of Length is N.

The value *Offset* returned from this primitive is zero based and, continuing from the example above, counting begins with Word1.

The CCR Z and C flags are changed to indicate whether a full or partial match was founds as follows. Note that a full match takes precedence over a partial match.

- Full match of *WordValue*: C = 1 and Z = 1
- Partial match of LSB of *WordValue* only: C = 0 and Z = 1
- Partial match of MSB of *WordValue* only: C = 1 and Z = 0
No match (full or partial): C = 0 and Z = 0

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C: Changed as described above
V: Unchanged
N: Unchanged
Z: Changed as described above

**Primitive Set and Number**
Set Zero, Number 0x14
**Manage Stack**
This primitive manages the stack belonging to the executing application. It is intended to be used in conjunction with the *Exit to MULTOS and Restart* primitive.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.x</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.1/2</th>
<th>MULTOS 4.5.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x07, Options

**Arguments**

*Options* defines the operation to perform on the stack as follows:

- 0 = Save the application’s stack to an internal temporary buffer
- 1 = Restore the application’s stack from the saved copy

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>{empty}</td>
<td>{empty}</td>
</tr>
</tbody>
</table>

**Remarks**

An application can call this primitive after calling Exit to MULTOS and Restart but before exiting back to MULTOS to save the contents of the application’s stack. When the application restarts it can call this primitive again to restore the state of its stack. This allows the application to resume execution at the point just after when it exited to MULTOS.

To prevent stack data leakage between applications MULTOS automatically deletes the saved stack if the exit to MULTOS results in the executing application being deleted.

This primitive abends under the following conditions:

- Options = 0 (save stack) and the application has not called Exit to MULTOS and Restart previously.
- Options = 1 (restore stack) and the application has not saved the stack previously.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C: Unchanged.
V: Unchanged
N: Unchanged
Z: Unchanged

**Primitive set and number**

Set one, number 0x07
Memory Compare

This primitive compares two blocks of bytes to determine if they hold the same data.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

// Stack holds length, addr1 and addr2 parameters
PRIM 0x0B

Arguments

None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Length</th>
<th>Addr1</th>
<th>Addr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter Length gives the size of the memory areas to be compared. The 2-byte values Addr1 and Addr2 are the locations of the areas.

Remarks

The comparison performed by this primitive is based on subtraction. The second operand, the area corresponding to the address on the top of the stack, is subtracted from the first. No data is modified, but the Condition Code Register is set according to the result of the operation.

There are three possible results of the comparison of blocks of size Length. They and the CCR setting used to indicate that result are:

- When the byte block at Addr1 > the byte block at Addr2, both CCR C and CCR Z flags are cleared.
- When the byte block at Addr1 = the byte block at Addr2, the CCR C flag is cleared and CCR Z flag is set
- When the byte block at Addr1 < the byte block at Addr2, the CCR C flag is set and CCR Z flag is cleared.

Where the number of bytes to be compared is a compile time constant and Length is no more than 255 bytes the primitive Memory Compare Fixed Length may be used.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
</table>
© 2019 MAOSCO Limited. MAO-DOC-TEC-006 v1.56
MULTOS is a registered trademark of MULTOS Limited.
C Set or cleared as above
V Unchanged
N Unchanged
Z Set or cleared as above

**Primitive Set and Number**

Set Zero, Number 0x0B

**Example**

The following example declares two four byte blocks which in a real application would represent PIN numbers. sPIN is the copy of the PIN in the applications static and pPIN is the copy of the PIN in the public segment sent as part of an APDU. The memory compare primitive is called to compare whether the two PIN numbers are the same.

```assembly
prmMemoryCompare EQU 0x0B

pPIN    PUBLIC BYTE 2
sPIN    STATIC BYTE 2 = 0x12, 0x34

PUSHW   2
LOADA   sPIN
LOADA   pPIN
PRIM    prmMemoryCompare
JNE     errPINdoesNotMatch
EXIT

errPINdoesNotMatch
EXITSW  0x65,0x81
```
Memory Compare Enhanced
This primitive compares two blocks of bytes.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.x  MULTOS 4.4  MULTOS 4.5.1  MULTOS 4.5.2

Syntax
PRIM 0x05, Mode

Arguments
Mode defines the comparison mode as follows:
0 = Equality only test
1 = Equality and greater/less than test.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Length</th>
<th>Addr1</th>
<th>Addr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>Result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter Length gives the size of the memory areas to be compared. The 2-byte values Addr1 and Addr2 are the locations of the areas. The 2-byte Result is the result of the comparison.

Remarks
The operation of this primitive is controlled by the Mode value.

Equality Only Test
The two memory areas are tested for equality and the Result can be one of the following two values.

0x5555 = blocks not equal
0xAAAA = blocks equal

Equality and Greater/Less Than Test
The comparison performed by this primitive is based on subtraction. The second operand, the area corresponding to the address on the top of the stack, is subtracted from the first (no data is modified) and the Result can be one of the following two values.

0x5A5A = byte block at Addr1 > byte block at Addr2
0xA5A5 = byte block at Addr1 < byte block at Addr2
0xAAAA = blocks equal
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged

*Primitive set and number*

Set one, number 0x05
**Memory Compare Fixed Length**

This primitive is used to compare two blocks of bytes of a fixed length.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmarks](symbol.png)

**Syntax**

```c
// Stack holds Addr1, Addr2 parameter
PRIM 0x0F, Length
```

**Arguments**

The argument *Length* is the number of bytes in each of the byte blocks.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr1, Addr2</td>
<td>{empty}</td>
</tr>
</tbody>
</table>

The 2-byte values *Addr1* and *Addr2* are the locations of the areas to be compared.

**Remarks**

The *Length* value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The comparison performed by this primitive is based on subtraction. The second operand, the area corresponding to the address on the top of the stack, is subtracted from the first. No data is modified, but the Condition Code Register is set according to the result of the operation.

There are three possible results of the comparison of blocks of size *Length*. They and the CCR setting used to indicate that result are:

- When the byte block at *Addr1* > the byte block at *Addr2*, both CCR C and CCR Z flags are cleared.
- When the byte block at *Addr1* = the byte block at *Addr2*, the CCR C flag is cleared and CCR Z flag is set.
- When the byte block at *Addr1* < the byte block at *Addr2*, the CCR C flag is set and CCR Z flag is cleared.

The primitive works correctly even if the blocks overlap.
Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set or cleared as above  
V  Unchanged  
N  Unchanged  
Z  Set or cleared as above

Primitive Set and Number

Set One, Number 0x0F

Example

The following example declares two four byte blocks which in a real application would represent PIN numbers. sPIN is the copy of the PIN in the application's static and pPIN is the copy of the PIN in the public segment sent as part of an APDU. The Memory Compare Fixed Length primitive is called to compare whether the two PIN numbers are the same.

```
prmMemoryCompareFixedLength EQU 0x0F

pPIN   PUBLIC BYTE 2
sPIN   STATIC BYTE 2 = 0x12, 0x34

LOADA   sPIN
LOADA   pPIN
PRIM    prmMemoryCompareFixedLength, 2
JNE     errPINdoesNotMatch
EXIT

errPINdoesNotMatch
EXITSW  0x65, 0x81
```
**Memory Copy**
This primitive copies a block of bytes from one location to another.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Syntax**

```c
// Stack holds Length, DestAddr, SourceAddr parameters
PRIM 0x0C
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length, DestAddr, SourceAddr</td>
<td>{empty}</td>
</tr>
</tbody>
</table>

All of the parameters are 2 bytes in size. The value *Length* is the number of bytes to copy. The values *DestAddr* and *SourceAddr* are, respectively, the locations to where and from where the data is copied.

**Remarks**

Where the number of bytes to be copied is a compile time constant and *Length* is no more than 255 bytes the primitive Memory Copy Fixed Length may be used.

This primitive works correctly even if the source and destination blocks overlap.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*C* Unchanged  
*V* Unchanged  
*N* Unchanged  
*Z* Unchanged

**Primitive Set and Number**

Set Zero, number 0x0C
Example

The following example copies a byte block from the bottom of Public to a variable called sName.

```
prmMemoryCopy EQU 0x0C
pLc EQU PT[-8]
sName STATIC BYTE 0x20

LOAD pLc,2   //Length of byte block to copy
LOADA sName  //Address to copy to (destination)
LOADA PB[0000]  //Address to copy from (source)
PRIM prmMemoryCopy
```
Memory Copy Additional Static

This primitive copies a block of memory from a segment address to an area of Static, from an area of Static to a segment address or from one area of Static to another area of Static. Either 32-bit or 64-bit Static addressing is supported.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0xDD, Options

Arguments

The 1 byte argument Options is used to specify the direction of the copy, whether the copy is atomic and the Static addressing mode as follows.

<table>
<thead>
<tr>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Copy data from segment address to Static offset</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Copy data from Static offset to segment address</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Copy data from Static offset to Static offset</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>1</td>
<td>1</td>
<td>RFU</td>
<td>RFU</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>x</td>
<td>x</td>
<td>Non-atomic copy</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>x</td>
<td>x</td>
<td>Atomic copy</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>x</td>
<td>x</td>
<td>32-bit Static addressing mode</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>x</td>
<td>x</td>
<td>64-bit Static addressing mode</td>
<td></td>
</tr>
</tbody>
</table>

Stack Usage (copy from segment address to Static offset)

If Options indicates that the copy is from a segment address to a Static offset then the stack will contain the following:

```
Stack In         Length StaticOffset SegAddr
                {empty}           |
```

The 2-byte Length identifies the number of bytes to copy.

StaticOffset specifies the Static offset of the destination. StaticOffset can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the specified Static addressing mode.

The 2-byte SegAddr specifies the segment address of the source.
Stack Usage (copy from Static offset to Segment address)

If Options indicates that the copy is from a Static offset to a segment address then the stack will contain the following:

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>SegAddr</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte Length identifies the number of bytes to copy.

The 2-byte SegAddr specifies the segment address of the destination. StaticOffset specifies the Static offset of the source. StaticOffset can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the specified Static addressing mode.

Stack Usage (copy from Static offset to Static offset)

If Options indicates that the copy is from a Static offset to a Static offset then the stack will contain the following:

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>StaticOffset2</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

Length identifies the number of bytes to copy. This length can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the specified Static addressing mode.

StaticOffset2 specifies the Static offset of the destination and StaticOffset1 specifies the Static offset of the source. StaticOffset1 and StaticOffset2 can be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the specified Static addressing mode.

Remarks

Invalid segment or Static addresses will cause an abend. The copy is successful even if the source and destination areas overlap.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged.
Primitive set and number

Set one, number 0xDD
**Memory Copy Fixed Length**

This primitive copies a block of bytes of a fixed length from one location to another.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✔ ✔ ✔ ✔ ✔ ✔

**Syntax**

// Stack holds DestAddr, SourceAddr parameters
PRIM 0x0E, Length

**Arguments**

The argument *Length* is the number of bytes to copy.

**Stack Usage**

Stack In  | DestAddr | SourceAddr |
----------|----------|------------|
Stack Out | {empty}  |            |

All of the parameters are 2 bytes in size. The values DestAddr and SourceAddr are, respectively, the locations to where and from where the data is copied.

**Remarks**

The *Length* value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

This primitive works correctly even if the blocks overlap.

**Condition Code**

| C | V | N | Z |
---|---|---|---|
-  | - | - | - |

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Unchanged

**Primitive Set and Number**

Set One, Number 0x0E
Example

The following example copies the 32 bytes at the bottom of Public to a variable called sName.

```apl
prmMemoryCopyFixedLength EQU 0x0E

sName STATIC BYTE 32

LOADA sName    //Address to copy to (destination)
LOADA PB[0000] //Address to copy from (source)
PRIM   prmMemoryCopyFixedLength, 32
```
Memory Copy From Replaced Application

This primitive allows for the currently executing application to copy the Session or Static data belonging to the application that it is replacing.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>4.3.x</td>
<td>4.4</td>
<td>4.5.1 / 2</td>
<td>4.5.3</td>
<td></td>
</tr>
</tbody>
</table>

Syntax

// Stack holds Length, Destination, Offset parameters
PRIM 0x06 Options

Arguments

Options:
- Bit 0: source data (0 = session, 1 = Static)
- Bit 6: Static addressing mode (0 = 32 bit, 1 = 64 bit)
- Bit 7: atomicity (0 = non-atomic, 1 = atomic)

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Length</th>
<th>DestAddr</th>
<th>SourceAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{empty}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset is the offset into the replaced application’s session or Static data. If the session data is being read then the offset is a 16-bit value, otherwise its size depends upon the addressing mode (32-bit or 64-bit).

Destination is a 16-bit value and holds the destination segment address. Length is a 16-bit value and holds the length of the data to read.

This primitive abends if no readable replaced application exists or if the offset/length values are invalid for the replaced application.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number

Set one, number 0x06


**Memory Copy Non-Atomic**

This primitive copies a block of bytes from one location to another. If the byte block is copied into the static area, data item protection function will be disabled if possible.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>☘</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

**Syntax**

```c
// Stack holds Length, DestAddr, SourceAddr parameters
PRIM 0x0F
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Length</th>
<th>DestAddr</th>
<th>SourceAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All of the parameters are 2 bytes in size. The value `Length` is the number of bytes to copy. The values `DestAddr` and `SourceAddr` are, respectively, the locations to where and from where the data is copied.

**Remarks**

This primitive works correctly even if the source and destination blocks overlap.

Where the number of bytes to be copied is a compile time constant and `Length` is no more than 255 bytes the primitive Memory Copy Non-Atomic Fixed Length may be used.

When copying into the static memory area with this primitive, the copying will be performed more quickly than with Memory Copy primitive as the data items are not protected.

This primitive is a request for a non-atomic memory copy. Non-atomic means that the data will be written in EEPROM page size blocks (see [MIR] for page size information for a specific implementation) when complete pages are available. If the data being copied results in writing to only a part of a page, then MULTOS will revert to an atomic copy. Whilst this copy operation may be faster the data in the destination will not be protected if power-off occurred during the copying to the static area. MULTOS will always guarantee the integrity of data other than the data being copied.

**Condition Code**

```
C  V  N  Z
--- --- --- ---
-   -   -   -
```
**Primitive Set and Number**

Set Zero, number 0x0F

**Example**

The following example shows how the primitive can be used as well as explaining how the copy takes place. The example will assume a page size of 32 bytes and that the copy destination is at the start of a page.

```
prmMemoryCopyNonAtomic EQU 0x0F

sDataBlock STATIC BYTE 270
pData PUBLIC BYTE 270

PUSHW 0x010E // length of 270 bytes to be copied
LOADA sDataBlock // Address to copy to (destination)
LOADA pData field // Address to copy from (source)
PRIM prmMemoryCopyNonAtomic
```

The memory copy would then copy 8 pages of data as 32 byte blocks. The remaining 24 bytes do not constitute a full page and would be copied atomically.
**Memory Copy Non-Atomic Fixed Length**

This primitive copies a block of bytes of a fixed length from one location to another. If the byte block is copied into the static area, data item protection function will be disabled if possible.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmarks](image)

**Syntax**

```c
// Stack holds estAddr, SourceAddr parameters
PRIM 0x13, Length
```

**Arguments**

The argument *Length* is the number of bytes to copy.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>DestAddr</th>
<th>SourceAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

All of the parameters are 2 bytes in size. The values *DestAddr* and *SourceAddr* are, respectively, the locations to where and from where the data is copied.

**Remarks**

This primitive works correctly even if the source and destination blocks overlap.

When copying into the static memory area with this primitive, the copying will be performed more quickly than with Memory Copy primitive as the data items are not protected.

This primitive is a request for a non-atomic memory copy.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged
**Primitive Set and Number**

Set Zero, number 0x13

**Example**

The following example shows how the primitive can be used as well as explaining how the copy takes place. The example will assume a page size of 32 bytes and that the copy destination is at the start of a page.

```
prmMemoryCopyNonAtomicFixedLength EQU 0x13

sDataBlock STATIC BYTE 120
pData PUBLIC BYTE 120

LOADA sDataBlock // Address to copy to (destination)
LOADA pData field // Address to copy from (source)
PRIM prmMemoryCopyNonAtomicFixedLength, 0x78
```

The memory copy would then copy 3 pages of data as 32 byte blocks. The remaining 24 bytes do not constitute a full page and would be copied atomically.
Memory Fill Additional Static

This primitive fills a block of Static memory with a specific byte value. Either 32-bit or 64-bit Static addressing is supported.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xDE, Options

Arguments

The 1 byte argument Options is used to specify whether the fill is atomic and the Static addressing mode as follows.

<table>
<thead>
<tr>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>Non-atomic fill</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>Atomic fill</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>32-bit Static addressing mode</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>RFU</td>
<td>64-bit Static addressing mode</td>
</tr>
</tbody>
</table>

Stack Usage

Stack In  | Value | Length | StaticOffset |
Stack Out |       |        | [empty]      |

The 1-byte Value identifies the value to fill the specified area of Static with.

Length identifies the number of bytes to fill. This length can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the specified Static addressing mode.

StaticOffset specifies the Static offset of the destination. StaticOffset can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the specified Static addressing mode.

Remarks

Invalid Static addresses will cause an abend.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged.
V  Unchanged
N  Unchanged
Z  Unchanged.

**Primitive set and number**

Set one, number 0xDE
**Modular Exponentiation / RSA Sign**

This primitive performs a modular exponentiation operation, the basis of the RSA algorithm. This version of the primitive will execute with full countermeasures to protect the algorithm.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓ ✓ ✓ ✓ ✓ ✓

**Syntax**

```
// Stack: eLen, mLen, eAddr, mAddr, inAddr, outAddr
PRIM 0xC8
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>eLen</th>
<th>mLen</th>
<th>eAddr</th>
<th>mAddr</th>
<th>inAddr</th>
<th>outAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{empty}</td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. The values eLen and mLen represent the length of the exponent and modulus respectively. These lengths represent the size in bytes. The value eAddr is the location of the exponent of size eLen, while mAddr is the location of the modulus of size mLen. The addresses inAddr and outAddr are the location of the input to the modular exponentiation operation and the address to where the output will be written.

**Remarks**

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

Moduli with length that is not a multiple of 8 bits are padded at the least significant end with bits 0. So, a 1023-bit modulus would have the least significant bit of the least significant byte set to 0.

The size of the input and output is considered to the same as that of the modulus. They are all mLen in size.

The primitive will function normally if inAddr and outAddr point to the same memory area. That is to say the output can overwrite the input.
In order to enable modular exponentiation to operate correctly there are a number of general conditions that must be met:

- The modulus must be odd.
- The base value must be less than the modulus.
- The exponent must be less than the modulus.
- The length of the exponent must be less than or equal to the length of the modulus.

There are some implementation specifics that may impact on the usage of this primitive. For example, the most significant byte of the modulus should not be zero although some platforms may permit it. As another example, some implementations may only work on fixed key lengths. It may also be the case that an implementation may provide optimised support for an exponent length of 1 with a value of 3 and from MULTOS 4.2 one may also provide optimised support for an exponent length of 3 and a value of 65537. See the MULTOS Implementation Report [MIR] for specific information.

**Primitive set and number**

Set zero, number 0xC8

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Example**

The following example shows how to use the modular exponentiation primitive to encrypt input using the private exponent. Here the 72-byte input value is found in public and the output overwrites it.

```
prmModularExponentiation         EQU 0xC8

sD   STATIC BYTE 64  // 64-byte private exponent
sN   STATIC BYTE 72  // 72-byte modulus

PUSHW  64         // exponent size
PUSHW  72         // modulus size
LOADA  sD         // exponent location
LOADA  sN         // modulus location
LOADA  PB[0]      // input location
LOADA  PB[0]      // output location
PRIM  prmModularExponentiation

EXITLA  72
```
Modular Exponentiation CRT / RSA Sign CRT
This primitive performs a modular exponentiation using the Chinese Remainder Theorem algorithm.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓  ✓  ✓  ✓  ✓  ✓

Syntax

// Stack: dpdqLen, dpdqAddr, pquAddr, inAddr, outAddr
PRIM 0xC9

Arguments

None.

Stack Usage

Stack In

Stack Out

All of the parameters are 2 bytes in size. The value held in dpdqLen is the size in bytes of the area located at dpdqAddr, where the dp value concatenated to the dq value is held. The parameter pquAddr is the location of the memory area where the values p, q and u are concatenated in that order. The parameters inAddr and outAddr are respectively the location of the input and the location where the output of the operation is written.

Remarks

This primitive performs modular exponentiation operation, where the operands are held in CRT format, and the result is written at the specified address outAddr.

Moduli with length that is not a multiple of 8 bits are padded at the least significant end with bits 0. So, a 1023-bit modulus would have the least significant bit of the least significant byte set to 0.

The size of the input, output and public modulus is considered to the value given in dpdqLen.

The primitive will function normally if inAddr and outAddr point to the same memory area. That is to say the output can overwrite the input.
In order to enable modular exponentiation CRT to operate correctly there are a number of general conditions that must be met:

- The public modulus size as measured in bytes and given in \( dpdq\text{Len} \) must be even
- \( p \) and \( q \) must be odd primes of size \( dpdq\text{Len} / 2 \)
- The public modulus is equal to \( p \times q \)
- \( u \) is the inverse of \( q \) modulo \( p \); i.e., \((q \times u) \mod(p) \equiv 1 \mod(p)\). This means that \( u < p \). The value \( u \) is held in a memory area of size \( dpdq\text{Len} / 2 \)
- \( dp \) is the value of the secret exponent modulo \( (p - 1) \) and is of size \( dpdq\text{Len} / 2 \)
- \( dq \) is the value of the secret exponent modulo \( (q - 1) \) and is of size \( dpdq\text{Len} / 2 \)

The most significant byte of \( p \) and \( q \) should not be zero. Some platforms may permit leading zero bytes, but this cannot be guaranteed.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged  
V Unchanged  
N Unchanged  
Z Unchanged

**Primitive Set and Number**

Set Zero, Number 0xC9

**Example**

The following example uses the modular exponentiation CRT primitive to encrypt a 72-byte value held in public memory. The result of the operation overwrites the input.

```assembly
ModularExponentiationCRT EQU 0xC9

sMod STATIC BYTE 72
sDPDQ STATIC BYTE 108
// following areas considered adjacent: sP | sQ | sU
sP STATIC BYTE 36
sQ STATIC BYTE 36
sU STATIC BYTE 36

PUSHW 72        // Length of modulus
LOADA sDPDQ     // Address of dp|dq
LOADA sP        // Address of p|q|u
LOADA PB[0]     // Address of base
LOADA PB[0]     // Address of result
PRIM ModularExponentiationCRT
```
Modular Exponentiation CRT Protected / RSA Sign CRT Protected

This primitive performs a modular exponentiation using the Chinese Remainder Theorem. The keys used however are stored in an enciphered form in memory and must be deciphered before use. It also provides a means to protect plaintext keys for subsequent use.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xDC

Arguments

None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>dpdqLen</th>
<th>dpdqAddr</th>
<th>pquAddr</th>
<th>inAddr</th>
<th>outAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter dpdqLen is the length in bytes of the modulus, which is even.
The 2-byte parameter dpdqAddr is the segment address of dp concatenated to dq.
The 2-byte parameter pquAddr is the segment address of p, q and u.
The 2-byte parameter inAddr is the segment address of the base bytes OR the address of dpdq if the keys are to be protected for subsequent use by the primitive.
The 2-byte parameter outAddr is the segment address to write the result to OR the address of pqu if the keys are to be protected for subsequent use by the primitive.

Remarks

This primitive calculates an exponent modulo a modulus, using the Chinese Remainder Theorem (CRT). The result (the base to the power of the exponent, modulo the modulus) is written at the specified segment address. The values of p and q are the two large prime numbers that were originally chosen to generate the key. The size of p and q is equal to half the length of the modulus. The remaining parameters for this primitive may be calculated from the values of p and q.

\[
\begin{align*}
\text{u} & \quad \text{inverse of q modulo p, of length dpdqLen} / 2 \\
\text{dp} & \quad \text{secret exponent modulo p - 1, of length dpdqLen} / 2 \\
\text{dq} & \quad \text{secret exponent modulo q - 1, of length dpdqLen} / 2 
\end{align*}
\]

A complete description of the Chinese Remainder Theorem and Cryptography is beyond the scope of this document. Please refer to a more specialised book on cryptography for more details on Chinese Remainder Theorem.
In order for Chinese Remainder Theorem to operate correctly there are certain conditions that must be satisfied.

1. \( N = p \times q \)
2. \( q > 2 \) \& \( p > 2 \)
3. \( p \) and \( q \) must both be odd.
4. The most significant byte of \( p \) and \( q \) should not be zero. Some platforms may permit leading zero bytes, but this cannot be guaranteed on each platform. Primes with length not a multiple of 8 bits are left padded with bits 0.
5. \( u \) must be less than \( p \)
6. \( u = (q \times u) \mod (p) = 1 \)

The keys used \((p, q, u, dp, dq)\) are stored as protected data items (for example enciphered). The method used is implementation specific and not described here. Before use, the implementation will reverse the effects of this encipherment to recover the actual keys to be used. This will be done without changing the stored value of the keys.

In order to protect the keys in the first place however, it is necessary for the application to ask the implementation to perform the necessary transformation and store the keys in the protected form. This is done by calling the primitive where the \( inAddr \) is set to \( dpdqAddr \) AND \( outAddr \) is set to \( pquAddr \). When this happens, the implementation does NOT invoke the modular exponentiation function but instead simply transform the keys into their protected form and writes them back to the addresses specified by \( inAddr \) and \( outAddr \).

If the primitive is called with \( inAddr \) set to \( pquAddr \) and \( outAddr \) set to \( dpdqAddr \) and passed protected keys, the original unprotected keys are be obtained and written back over the corresponding input data.

If \( inAddr \) points to \( dpdqAddr \) (or \( pquAddr \)) but \( outAddr \) does not point to \( pquAddr \) (or \( dpdqAddr \)) or \( outAddr \) points to \( pquAddr \) (or \( dpdqAddr \)) but \( inAddr \) does not point to \( dpdqAddr \) (or \( pquAddr \)) then the primitive will perform a Modular Exponentiation using the protected keys specified in \( pquAddr \) and \( dpdqAddr \) and either exponentiating the protected input keys to produce the output result (if \( inAddr \) is \( pquAddr \) or \( dpdqAddr \)) or overwriting the protected keys with the result (if \( outAddr \) is \( pquAddr \) or \( dpdqAddr \)).

Note: Modular Exponentiation CRT in some implementations may only work on fixed key lengths. See the MULTOS Implementation Report for more details.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **C** Unchanged
- **V** Unchanged
- **N** Unchanged
- **Z** Unchanged

**Primitive Set and Number**

Set Zero, number 0xDC
Example

The following example uses the modular exponentiate CRT primitive to perform an encipher or decipher.

prmModExpCRTProtected EQU 0xDC

sMod  STATIC BYTE 64
sDPDQ STATIC BYTE 96
sP    STATIC BYTE 32
sQ    STATIC BYTE 32
sU    STATIC BYTE 32
sBase STATIC BYTE 64

//Call primitive to protect the keys
// (word) Length of Modulus
// (word) Address of dp|dp
// (word) Address of p|q|u
// (word) Address of dp|dq
// (word) Address of p|q|u

PUSH 0x64 //Length of modulus
LOADA sDPDQ //Address of dp|dq
LOADA sP //Address of p|q|u
LOADA sDPDQ //Address of dp|dq
LOADA sP //Address of p|q|u
PRIM prmModExpCRTProtected // call primitive

//Now call the primitive to perform a CRT exponentiation
//using the previously protected keys

PUSH 0x64 //Length of modulus
LOADA sDPDQ //Address of dp|dq
LOADA sP //Address of p|q|u
LOADA sBase //Address of base
LOADA PB[0] //Address of result
PRIM prmModExpCRTProtected // call primitive
**Modular Inverse**

This primitive calculates the modular inverse of a value. A modular inverse of an integer \( b \) (modulo \( m \)) is the integer \( b^{-1} \) such that \( b b^{-1} = 1 \) (mod \( m \)).

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xD0, Prime

**Arguments**

The 1-byte argument Prime is set to 1 if the modulus used is prime. Otherwise, it is set to 0.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>{empty}</td>
<td>mLen</td>
</tr>
<tr>
<td></td>
<td>modulusAddr</td>
</tr>
<tr>
<td></td>
<td>inLen</td>
</tr>
<tr>
<td></td>
<td>inAddr</td>
</tr>
<tr>
<td></td>
<td>outAddr</td>
</tr>
</tbody>
</table>

All the parameters are 2 bytes in size. The parameter \( mLen \) states the length in bytes of the modulus value, which can be found at \( modulusAddr \). The value \( inLen \) gives the length of the input data found at \( inAddr \). The result of the modular inverse calculation is written to \( outAddr \).

**Remarks**

The size of the output held at \( outAddr \) is considered to the value given in \( mLen \).

The value calculated is one such that the value stored at the segment address \( inAddr \) modulo the modulus stored at the segment address \( modulusAddr \) is congruent to 1 modulo the supplied modulus.

In order to calculate the modular inverse the input value and the modulus must be co-prime. If they are not the CCR Z flag is set and no value is written to \( outAddr \).
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Set if the modular inverse cannot be calculated, cleared otherwise

**Primitive Set and Number**

Set One, Number 0xD0

**Example**

The following example shows how to use the Modular Inverse primitive to calculate the secret key of an RSA key set with a 128-byte modulus and a public exponent of 3. The primitive call below calculates the modular inverse of 3 with respect to the modulus \((P – 1) \times (Q – 1)\).

```assembly
prmModInv EQU 0xD0
prmMultiplyN EQU 0x10

dModulus DYNAMIC BYTE 128
sPrimeP STATIC BYTE 64 // The first prime
sPrimeQ STATIC BYTE 64 // The second prime

LOAD sPrimeP, 64 // Load P
DECN ,64 // P - 1
LOAD sPrimeQ, 64 // Load Q
DECN ,64 // Q - 1
PRIM prmMultiplyN, 64 // (P - 1) * (Q - 1)
STORE dModulus, 128 // move result to variable

//-------------------------------------------------
//Call Modular Inverse to calculate secret key
//-------------------------------------------------

PUSHB 3 // Public Exponent
PUSHW 128 // Size of modulus
LOADA dModulus // Address of modulus
PUSHW 1 // Size of input
LOADA DT[-7] // Address of input
LOADA PB[0] // Address of destination
PRIM prmModInv, 0x00 // Calculate inverse
BEQ Invalid // Invalid if no inverse
EXITLA 128 // Return result

Invalid
EXITSW 0x9E, 0x20 // No inverse possible
```
Modular Multiplication

This primitive multiples two operands and reduces the result of the multiplication modulo a given modulus.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xC2

Arguments

None.

Stack Usage

Stack In

<table>
<thead>
<tr>
<th>lenMod</th>
<th>addrOp1</th>
<th>addrOp2</th>
<th>addrMod</th>
</tr>
</thead>
</table>

Stack Out

{empty}

All the parameters are 2 bytes in size. The parameter lenMod is the size of the modulus supplied and located at addrMod. The parameters addrOp1 and addrOp2 are the locations of the multiplicands.

Remarks

This primitive calculates a product modulo a modulus, that is (Operand1 * Operand2) mod modulus. The result overwrites the first operand.

Both operands must represent values that are less than that of the modulus.

The modulus and both operands are considered to be of size lenMod.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

**Primitive Set and Number**

Set Zero, Number 0xC2

**Examples**

The following example uses modular multiplication where the operands are in public memory.

```
prmModularMultiplication EQU 0xC2
MODSIZE EQU 72

sModulus STATIC BYTE MODSIZE

PUSHW MODSIZE
LOADA sModulus
LOADA PB[0]
LOADA PB[MODSIZE]
PRIM prmModularMultiplication
EXITLA MODSIZE
```
Modular Reduction
This primitive reduces an operand with respect to a modulus.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0xC3

Stack Usage
Stack In  lenOp  lenMod  addrOp  addrMod
Stack Out  {empty}

All parameters are 2 bytes in size. The parameters lenOp and lenMod state the size of the operand to be reduced and the modulus respectively. The location of the operand is given in addrOp while the modulus location is given in addrMod.

Remarks
This primitive calculates Operand mod Modulus. The result is written to addrOp and will be of length lenMod.

If lenOp is less than lenMod the result is undefined.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number
Set Zero, Number 0xC3

Example
The following example reduces a value with respect to a 96-byte modulus. The value to be reduced is sent as command data.
prmModReduction EQU 0xC3
pLC EQU PT[-8]
MODSIZE EQU 96

sModulus STATIC BYTE MODSIZE

// check that incoming data length >= 96
CMPW pLC, MODSIZE
JLT errOperandSize
// stack parameters set
LOAD pLC, 2
PUSHW MODSIZE
LOADA PB[0]
LOADA sModulus
PRIM prmModReduction
**MultiplyN**

This primitive multiplies two unsigned blocks of bytes from the stack together and leaves the result on the stack.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓ ✓ ✓ ✓ ✓ ✓ ✓

**Syntax**

// Stack holds Operand1, Operand2
PRIM 0x10, Length

**Arguments**

The argument Length indicates the size of the multiplicands.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>Output</td>
<td></td>
</tr>
</tbody>
</table>

The parameters Operand1 and Operand2 are the values of size Length that are to be multiplied. The output parameter Output holds the result of the multiplication is twice the size of Length.

**Remarks**

This primitive performs unsigned multiplication of two numbers. The result replaces the two operands at the top of stack.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the result is zero, cleared otherwise

**Primitive Set and Number**

Set One, Number 0x10
Example

The following example pushes two words onto the stack and multiplies them together. The result is left on the stack at the end of the example.

```
prmMultiplyN   EQU  0x10

PUSHW  0x0100   //Stack: 01,00
PUSHW  0x0002   //Stack: 01,00,00,02
PRIM   prmMultiplyN, 2   //Stack: 00,00,02,00
```
**Pad**
This primitive adds padding to a non-padded message.

**Availability**
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- ✔
- ✔
- ✔
- ✔
- ✔

**Syntax**
PRIM 0x15, PadScheme

**Arguments**
The 1-byte argument PadScheme specifies the padding scheme, as follows.
- 0x01: The message is appended with the byte 0x80 and it is then padded with zero or more bytes of 0x00 to the next multiple of BlockLen bytes.
- 0x02: The message is appended with the byte 0x80 and it is then padded with one or more bytes of 0x00 to the next multiple of BlockLen bytes.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockLen</td>
<td>lenMsg</td>
</tr>
<tr>
<td>lenMsgPadded</td>
<td>addrMsg</td>
</tr>
</tbody>
</table>

The 1-byte parameter BlockLen specifies the padding block length in bytes.
The 2-byte parameter lenMsg specifies the length of the message to be padded in bytes.
The 2-byte parameter addrMsg specifies the segment address of the message to be padded.
The 2-byte result lenMsgPadded is the length in bytes of the padded message.

**Remarks**
This primitive pads a message to a specific block size according to a specified padding scheme.
The padding is added to the end of the specified block.
The calling application needs to ensure that the total size of the memory area in which the message is held in sufficiently large to allow the padding to be added.
The primitive supports block lengths of 8 and 16 bytes.
The primitive abends, if an invalid PadScheme value is supplied or if BlockLen is not supported by the implementation.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number

Set One, Number 0x15
**Platform Optimised Checksum**
This primitive calculates a checksum using a platform-specific optimised algorithm.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x89

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
<tr>
<td>BlockAddr</td>
<td></td>
</tr>
</tbody>
</table>

The 2-byte parameter Length specifies the length in bytes of the block of memory to be checksummed.
The 2-byte parameter BlockAddr specifies the segment address of the block of memory to be checksummed.
The 4-byte parameter Checksum is the resultant four byte checksum.

**Remarks**

This primitive generates a four byte checksum over the block of memory starting at BlockAddr and of length Length using a performance optimised platform specific method.

If the block is in Static, and transaction protection is on, the checksum calculation takes pending writes into account. This is an exception to the general rule that pending writes are not visible to the application until they are committed.

There are no specific guarantees about the properties of the checksum algorithm, however, MULTOS implementations should aim to ensure that the checksum calculated has the full strength of a four byte checksum (i.e. there should be a 1/2^32 probability that the checksums calculated over two different random blocks of data have the same value). The exact algorithm implemented by this primitive on a particular platform may be specified in the MULTOS Implementation Report but otherwise application developers cannot assume the results of this primitive will conform to any particular algorithm and should assume that the result calculated on different platforms will be different.
The checksum is returned in Dynamic, where it overwrites the length and segment address of the checksummed area.

It is valid to calculate the checksum of a block of length zero.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unchanged</td>
<td>Unchanged</td>
<td>Unchanged</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

**Primitive Set and Number**

Set One, Number 0x89

**Example**

The following example performs a checksum over a block of the Static area. The Static area is declared as a number of variables, however, the checksum is performed over all of the variables. Typically this may be used to verify that data has been loaded into the variables correctly.

The correct value for the checksum is held in the bottom four byte of Public.

```assembly
prmPlatOptCheckSum EQU 0x17
sName STATIC BYTE 10
sVariable2 STATIC BYTE 5
sVariable3 STATIC BYTE 5

PUSHW 20
LOADA sName
PRIM prmPlatOptCheckSum
CMPN PB[0000],4
JNE InvalidCheckSum

ValidCheckSum
EXIT

InvalidCheckSum
EXIT
```
**Query0, Query1, Query2, Query3**

These primitives check that a specific primitive from the sets 0 to 3 is available.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Syntax**

There are four different primitive numbers assigned depending on which primitive set is being queried. They are:

- `PRIM 0x00, primNo` // Check Set 0 primitive
- `PRIM 0x01, primNo` // Check Set 1 primitive
- `PRIM 0x02, primNo` // Check Set 2 primitive
- `PRIM 0x03, primNo` // Check Set 3 primitive

**Arguments**

The 1-byte parameter `primNo` is the number of the primitive whose existence is being checked.

**Stack Usage**

- Stack In  {empty}
- Stack Out {empty}

There are no input or output parameters for these primitives.

**Remarks**

This group of primitives allows an application to query the availability of other primitives. Query0 is used to query the existence of primitives in set zero, Query 1 in set one, and so on.

The set of a primitive is given in the Primitive and Set Number section of each primitive documented in this document.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- C Unchanged
- V Unchanged
- N Unchanged
- Z Set if the desired primitive exists, cleared otherwise
### Primitive Set and Number

Set One, numbers 0x00, 0x01, 0x02 and 0x03

#### Example

The following example tests for the implementation of the Query2 primitive.

```assembly
prmQuery1   EQU  0x01
prmQuery2   EQU  0x02

PRIM     prmQuery1, prmQuery2
JNE     PrimNotSupported
//Continue normal execution

PrimNotSupported
EXIT
```
**Query Channel**

This primitive allows a MULTOS application to determine whether a channel is supported by the platform.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x86

**Arguments**

None.

**Stack Usage**

Stack In

Stack Out

{empty}

The 1-byte parameter ChannelID indicates which non-MULTOS application channel should be queried.

**Remarks**

If the specified channel is supported then the Z flag is set, otherwise it is cleared.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Unchanged

V  Unchanged

N  Unchanged

Z  set if the channel number is supported by the platform, cleared otherwise.

**Primitive set and number**

Set zero, number 0x86
**Query Codelet**

This primitive queries the existence of a specific codelet on the MULTOS device.

**Availability**

- MULTOS 4
- MULTOS 4.2
- MULTOS 4.3.1
- MULTOS 4.3.2
- MULTOS 4.4
- MULTOS 4.5.x

**Syntax**

```
PRIM 0x84
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeletID</td>
<td>CodeletID</td>
</tr>
</tbody>
</table>

The 2-byte parameter `CodeletID` gives the globally unique identification number of the codelet to be queried.

**Remarks**

A codelet is code that has been included in ROM during the masking process. The code, which can be a complete application or a library of functions, is available to all applications on the device. Support for any particular codelet is at the discretion of the implementor. However, all codelets are registered with the MULTOS Key Management Authority and each has a unique identifier.

The purpose of this primitive is to determine if a codelet with the indicated ID is available on the MULTOS device. If the codelet with ID `CodeletID` is present in the device, the CCR Z flag is set.

The 2-byte value `CodeletID` remains on the stack after the primitive executes. This can be used by a following ‘Call Codelet’ primitive.

Note that a codelet ID of 0 is valid and refers to the executing application. Given that the executing application must exist, the codelet exists and the CCR Z flag is set accordingly. See the remarks section of the primitive ‘Call Codelet’ for further information.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the Codelet ID was found, cleared otherwise.

**Primitive Set and Number**

Set Zero, Number 0x84

**Example**

The following example checks that a particular codelet exists and if so proceeds to call the codelet. Note that the codelet ID used below is fictitious.

```assembly
prmCallCodelet       EQU 0x83
prmQueryCodelet      EQU 0x84
CODELETID            EQU 0xF1F2

PUSHW CODELETID
PRIM       prmQueryCodelet
// CCR Z flag cleared if does not exist
BEQ    warning_CodeletUnsupported
// otherwise call the codelet from start
// codelet ID remained on stack
PUSHZ 2
PRIM       prmCallCodelet
```
**Query Cryptographic Algorithm**

This primitive allows a MULTOS application to determine whether a cryptographic algorithm is supported by the implementation. The primitive cannot be used to determine any restrictions in the use of the algorithm on any implementation.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x8A

**Arguments**

None.

**Stack Usage**

Stack In: AlgorithmID

Stack Out: [empty]

The 1-byte parameter AlgorithmID indicates which cryptographic algorithm should be queried.

<table>
<thead>
<tr>
<th>AlgorithmID</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>DES [FIPS46-3]</td>
</tr>
<tr>
<td>0x04</td>
<td>Triple DES [FIPS46-3]</td>
</tr>
<tr>
<td>0x05</td>
<td>SEED [KISA]</td>
</tr>
<tr>
<td>0x06</td>
<td>AES [FIPS197]</td>
</tr>
<tr>
<td>0x07</td>
<td>RSA</td>
</tr>
<tr>
<td>0x08</td>
<td>Comp-128</td>
</tr>
<tr>
<td>0x09</td>
<td>ECC</td>
</tr>
</tbody>
</table>

**Remarks**

If the specified algorithm is supported then the Z flag is set, otherwise it is cleared.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C  Unchanged
V  Unchanged
N  Unchanged
Z  set if the algorithm is supported by the platform, cleared otherwise.

Primitive set and number

Set zero, number 0x8A
**Query Interface Type**

This primitive indicates the type of interface is being used to communicate to the device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Syntax**

```
PRIM 0x0D
```

**Arguments**

None.

**Stack Usage**

Stack In          {empty}
Stack Out         {empty}

There are no input or output parameters for this primitive.

**Remarks**

The primitive allows an application to determine if the terminal is communicating to the device using a contact or contactless interface. The CCR Z flag is updated depending on the result of the primitive processing.

**Condition Code**

```
<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

C  Unchanged
V  Unchanged
N  Unchanged
Z  Set if the interface was contactless, cleared if contact

**Primitive Set and Number**

set zero, number 0x0D
**Example**

The following example exits indicating that a function is not supported if the interface type is contactless.

```assembly
prmQueryInterface EQU 0x0D

PRIM prmQueryInterface
JNE err_FuncNotSupported
    // normal functioning if interface is contact

err_FuncNotSupported
    EXITSW 0x6A81
```
Read PIN

Returns the clear PIN which is either the local application PIN or the Global PIN depending on the access_list bit settings in the ALC. See Initialise PIN for details.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xE6

Arguments

None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
<th>OutAddr</th>
<th>PinLength</th>
</tr>
</thead>
</table>

OutAddr is a 2 byte address of a buffer to contain the returned PIN.
PinLength (1 byte) is the length of the PIN returned in OutAddr

Remarks

This primitive will abend if the ALC Permission is either Global/Basic or Global/Write or if the PIN has not yet been initialised.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number

set zero, number 0xE6
Reject Process Event

The Reject Process Event primitive can be called by any application to request that the current application process event is rejected by MULTOS. The application continues to execute normally, with MULTOS processing the request when the application exits. The effect of calling this primitive depends upon the event that is being rejected (see [MDG] for more information).

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

PRIM 0xE9

Stack Usage

Stack In   {empty}
Stack Out  {empty}

Remarks

This primitive has no effect if the required access_list bit is not set.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number

Set Zero, Number 0xE9

Reset Session Data

This primitive allows a shell application to reset the session data of all other applications on the MULTOS device.
Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓  ✓  ✓  ✓  ✓  ✓  ✓

Syntax

PRIM 0x81

Arguments

None.

Stack Usage

Stack In   {empty}
Stack Out  {empty}

There are no input or output parameters for this primitive.

Remarks

If the calling application is not executing as a shell, this primitive has no effect.

This primitive clears the session data of all other applications. The session data of the shell application is unaffected.

Condition Code

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>V</td>
<td>N</td>
<td>Z</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

Primitive Set and Number

Set Zero, Number 0x81
**Reset WWT**

This primitive sends a work wait time extension request to the terminal.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

```
PRIM 0x02
```

**Arguments**

None.

**Stack Usage**

- Stack In: {empty}
- Stack Out: {empty}

There are no input or output parameters for this primitive.

**Remarks**

This primitive causes a message to be sent to the terminal to inform it that more time is required for processing to complete. The nature of the message is protocol-dependent in accordance with [ISO7816-3]. Under T=0, for example, a NULL byte is sent while under T=1 an S-Block, Supervisor Wait Time Extension, is sent.

Most MULTOS implementations send work wait extension requests automatically and the frequency of the resets is implementation specific. This automatic functioning can be disabled using the Control Auto Reset WWT Primitive.
### Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

### Primitive Set and Number

Set Zero, Number 0x02
Return from Codelet

This primitive is used when a codelet based function finishes executing and returns control to the calling application. It allows input to be removed from and output left on the stack.

Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓  ✓  ✓  ✓  ✓  ✓

Syntax

PRIM 0x80, BytesIn, BytesOut

Arguments

The 1-byte parameter BytesIn indicates the number of bytes to be removed from the stack upon returning from the codelet. The 1-byte sized BytesOut parameter indicates the number of bytes to be left on the stack as output from the codelet processing.

Stack Usage

Stack In    [n]
Stack Out   [m]

Both the input and out parameters n and m are variable in size. The size can range from 0 to 255 bytes. The input parameter n is of size BytesIn and m is of size BytesOut.

The input and output can also consist of several variables of differing size, but this primitive does not concern itself with the data structure, but rather the total length.

Remarks

This primitive is used to return control from a codelet to the application that invoked it. The two arguments are used to discard parameters passed to the codelet and return result bytes in the same way as the PRIMRET Return instruction operates.

This primitive is expecting linkage data to be on the stack. These bytes are automatically placed there by the primitive ‘Call Codelet’.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged

**Primitive Set and Number**

set two, number 0x80
RSA Verify
This primitive performs a modular exponentiation operation, the basis of the RSA algorithm. This version of the primitive is optimised for use with Public key operations only and platform countermeasures that protect the RSA algorithm may be disabled. For Private key operations the Modular Exponentiation / RSA Sign primitive should be used.

*IT IS STRONGLY ADVISED THIS PRIMITIVE IS USED WITH PUBLIC KEYS ONLY.*

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

|x| x| ✓| ✓| ✓| ✓|

**Syntax**

```c
// Stack: eLen, mLen, eAddr, mAddr, inAddr, outAddr
PRIM 0xEB
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>eLen</th>
<th>mLen</th>
<th>eAddr</th>
<th>mAddr</th>
<th>inAddr</th>
<th>outAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All parameters are 2 bytes in size. The values eLen and mLen represent the length of the exponent and modulus respectively. These lengths represent the size in bytes. The value eAddr is the location of the exponent of size eLen, while mAddr is the location of the modulus of size mLen. The addresses inAddr and outAddr are the location of the input to the modular exponentiation operation and the address to where the output will be written.

**Remarks**

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

Moduli with length that is not a multiple of 8 bits are padded at the least significant end with bits 0. So, a 1023-bit modulus would have the least significant bit of the least significant byte set to 0.

The size of the input and output is considered to the same as that of the modulus. They are all mLen in size.

The primitive will function normally if inAddr and outAddr point to the same memory area. That is to say the output can overwrite the input.
In order to enable modular exponentiation to operate correctly there are a number of general conditions that must be met:

- The modulus must be odd.
- The base value must be less than the modulus.
- The exponent must be less than the modulus.
- The length of the exponent must be less than or equal to the length of the modulus.

There are some implementation specifics that may impact on the usage of this primitive. For example, the most significant byte of the modulus should not be zero although some platforms may permit it. As another example, some implementations may only work on fixed key lengths. It may also be the case that an implementation may provide optimised support for an exponent length of 1 with a value of 3 and from MULTOS 4.2 one may also provide optimised support for an exponent length of 3 and a value of 65537. See the MULTOS Implementation Report [MIR] for specific information.

**Primitive set and number**

Set zero, number 0xEB

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged

**Example**

The following example shows how to use the modular exponentiation primitive to encrypt input using the private exponent. Here the 72-byte input value is found in public and the output overwrites it.

```
prmModularExponentiation EQU 0xEB
```

```
sD STATIC BYTE 64 // 64-byte private exponent
sN STATIC BYTE 72 // 72-byte modulus

PUSHW 64  // exponent size
PUSHW 72  // modulus size
LOADA sD  // exponent location
LOADA sN  // modulus location
LOADA PB[0]  // input location
LOADA PB[0]  // output location
PRIM prmModularExponentiation

EXITLA 72
```
**Secure Hash**
This primitive calculates the SHA-1, SHA-224, SHA-256, SHA-384 or SHA-512 digest of a message of arbitrary length in accordance with [FIPS180-3].

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xCF

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>lenMsg</td>
<td>lenHash</td>
</tr>
<tr>
<td>addrHash</td>
<td>addrMsg</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

Each of the input parameters is 2-bytes in size. The value \( lenMsg \) is the size in bytes of the input to the Secure Hash algorithm. The value \( lenHash \) is either 20, 28, 32, 48 or 64 and is the size of the resultant hash digest. The parameter \( addrHash \) is the location where the hash digest will be written. The parameter \( addrMsg \) is the location of the input of size \( lenMsg \).

**Remarks**

The primitive uses the appropriate Secure Hash algorithm according to the length of the hash digest requested, i.e.

<table>
<thead>
<tr>
<th>lenHash</th>
<th>Algorithm used</th>
<th>Algorithms supported (when primitive is implemented)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Algorithm used</td>
<td>MULTOS 4.2</td>
</tr>
<tr>
<td>20</td>
<td>SHA-1</td>
<td>Optional</td>
</tr>
<tr>
<td>28</td>
<td>SHA-224</td>
<td>Optional</td>
</tr>
<tr>
<td>32</td>
<td>SHA-256</td>
<td>Supported</td>
</tr>
<tr>
<td>48</td>
<td>SHA-384</td>
<td>Optional</td>
</tr>
<tr>
<td>64</td>
<td>SHA-512</td>
<td>Optional</td>
</tr>
</tbody>
</table>

Unsupported values of \( lenHash \) will cause an abend. The primitive functions properly even if \( lenMsg \) is zero.
Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

Primitive set and number

Set zero, number 0xCF
**Secure Hash IV**

This primitive calculates the SHA-1, SHA-224, SHA-256, SHA-384 or SHA-512 digest of a message of arbitrary length in accordance with [FIPS180-3] with the ability to pass a previously calculated intermediate hash value and message remainder (where the previous message was not block-aligned) to the algorithm.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

```
PRIM 0xE4
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>lenMsg</th>
<th>lenHash</th>
<th>addrHash</th>
<th>addrMsg</th>
<th>addrIntermediateHash</th>
<th>addrPrevHashedBytes</th>
<th>lenMessageRemainder</th>
<th>addrMessageRemainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>lenMessageRemainder</td>
<td>addrMessageRemainder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each of the input parameters is 2-bytes in size.
The value lenMsg is the size in bytes of the input to the Secure Hash algorithm.
The value lenHash is either 20, 28, 32, 48 or 64 and is the size of the resultant hash digest.
The parameter addrHash is the location where the hash digest will be written.
The parameter addrMsg is the location of the input message of size lenMsg.
The parameter addrIntermediateHash is the location of the previously calculated intermediate hash value input to the algorithm and output from the algorithm. It is 20, 32 or 64 bytes in length dependent upon the algorithm requested.
The parameter addrPrevHashedBytes is the location of the 4 byte (32-bit) counter indicating the number of bytes previously input to the hashing algorithm, including previous calculations.
The parameter lenMessageRemainder is the number remaining non-block aligned bytes from a previously hashed message.
The parameter addrMessageRemainder is the address of the remaining non-block aligned bytes of a previously hashed message, of length lenMessageRemainder.

**Remarks**

The primitive uses the appropriate Secure Hash algorithm according to the length of the hash digest requested, i.e. The length of the intermediate hash value passed to the algorithm and maximum length of the message remainder (if present) depends upon the algorithm to be used.
Unsupported values of lenHash will cause an abend.

The primitive functions properly even if lenMsg is zero.

If the value at addrIntermediateHash is all zeros, then the algorithm shall replace this value with the standard IV value used by the algorithm, as specified in [FIPS180-3].

The 32-bit value at addrPrevHashedBytes is the number of bytes previously hashed by a call to this primitive or an alternative calculation method. If the value at this address is zero, the primitive will start a new hash calculation and ignore the values contained at addrIntermediateHash and addrMessageRemainder. This value is updated by the primitive and may serve as input to a subsequent call to the primitive.

If lenMessageRemainder is zero, the value at addrMessageRemainder will be ignored, but value at addrIntermediateHash will still be used as the input value to the algorithm.

Following calculation, the memory at location addrIntermediateHash shall contain the last intermediate hash value H(n) calculated by the algorithm prior to any truncation when performing a SHA-224 or SHA-384 algorithm. This value may serve as input to a subsequent call to the primitive. The memory at addrHash will always contain a final hash value complete with truncation if applicable.

If the message hashed (the value at addrMessageRemainder prepended to the value at addrMsg) is not block aligned (i.e. not a multiple of either 32 or 64 bytes depending upon the hash algorithm), following calculation of the intermediate value, the remainder of the message of length lenMessageRemainder shall be pointed to by addrMessageRemainder and will be placed on the returned stack. This memory address may be within the area starting at addrMsg for length lenMsg or it may be at the address passed to the primitive.

Developers should ensure that there is sufficient memory at address addrMessageRemainder to contain the message remainder of the appropriate block size, as the returned message remainder can be longer than the input message remainder. If a developer does not allocate such a memory area, then the primitive may overwrite memory beyond addrMessageRemainder + lenMessageRemainder or abend.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
V Unchanged

© 2019 MAOSCO Limited. MAO-DOC-TEC-006 v1.56
MULTOS is a registered trademark of MULTOS Limited.
N  Unchanged
Z  Unchanged

*Primitive set and number*

Set zero, number 0xE4
**SEED ECB Decipher**

This primitive performs SEED ECB Decipher on a sixteen byte block of memory using a sixteen byte key.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- Unchanged
- Unchanged
- Unchanged
- Unchanged
- Unchanged
- Unchanged

**Syntax**

PRIM 0xCD

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyAddr</td>
<td>OutputAddr</td>
</tr>
<tr>
<td>{empty}</td>
<td></td>
</tr>
</tbody>
</table>

Each input parameter is 2 bytes in size. The value `KeyAddr` is the location of the key used in the SEED algorithm, `InputAddr` is the location of the data block that serves as input to the decipher operation and `OutputAddr` is the location where the output should be written.

**Remarks**

SEED is a 128-bit symmetric key block cipher that had been developed by KISA, the Korea Information Security Agency.

This primitive recovers 16-byte plaintext from a SEED ECB ciphertext.

The result may overwrite the input; i.e., `OutputAddr` and `InputAddr` may point to the same location.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C Unchanged
- V Unchanged
- N Unchanged
- Z Unchanged

**Primitive Set and Number**

Set Zero, number 0xCD
Example

The following example declares 48 bytes of static memory to hold the SEED Key, the plaintext block and the ciphertext block. The address for each of these is loaded onto the stack and the SEED ECB Decipher primitive called.

```asm
prmSEEDECBDecipher EQU 0xCD

sSEEDKey STATIC BYTE 16 =
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F

sPlaintext STATIC BYTE 16
sCiphertext STATIC BYTE 16

LOADA sSEEDKey
LOADA sPlaintext
LOADA sCiphertext
PRIM prmSEEDECBDecipher
```
**SEED ECB Encipher**

This primitive performs SEED ECB Encipher on a sixteen byte block of memory using a sixteen byte key.

**Availability**

MULTOS 4   MULTOS 4.2   MULTOS 4.3.1   MULTOS 4.3.2   MULTOS 4.4   MULTOS 4.5.x

![Available](image)

**Syntax**

PRIM 0xCE

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>KeyAddr</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each input parameter is 2 bytes in size. The value *KeyAddr* is the location of the key used in the SEED algorithm, *InputAddr* is the location of the data block that serves as input to the encipher operation and *OutputAddr* is the location where the output should be written.

**Remarks**

SEED is a 128-bit symmetric key block cipher that had been developed by KISA, the Korea Information Security Agency.

This primitive generates a SEED ECB ciphertext output from an 16-byte plaintext input.

The result may overwrite the input; i.e., *OutputAddr* and *InputAddr* may point to the same location.

**Condition Code**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unchanged</td>
<td>Unchanged</td>
<td>Unchanged</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

**Primitive Set and Number**

Set Zero, number 0xCE
The following example declares 48 bytes of static memory to hold the SEED Key, the plaintext block and the ciphertext block. The address for each of these is loaded onto the stack and the SEED ECB Encipher primitive called.

```plaintext
prmSEEDECBEncipher  EQU  0xCE

sSEEDKey  STATIC BYTE 16 =
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
sPlaintext STATIC BYTE 16
sCiphertext STATIC BYTE 16

LOADA  sSEEDKey
LOADA  sCiphertext
LOADA  sPlaintext
PRIM   prmSEEDECBEncipher
```
**Set AFI**

This primitive sets the value of the Application Family Indicator for the current application.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

*  ✓  ✓  ✓  ✓  ✓  ✓

*This primitive is mandatory if the device supports ISO/IEC 14443 Type B contactless communication.

**Syntax**

```plaintext
// Stack holds AFI value
PRIM 0x12
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th></th>
<th>Stack Out</th>
<th>{empty}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AFIvalue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The input parameter `AFIvalue` is 1-byte in size and holds the value to which the AFI will be set.

**Remarks**

The AFI may be specified by a contactless ISO/IEC 14443 Type B terminal during anti-collision processing. If a device contains an application that has the same AFI, then the device will respond otherwise the device will not respond.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

**Primitive Set and Number**

Set Zero, Number 0x12
**Set ATR File Record**

This primitive writes a record into the ATR File record.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

**Syntax**

```c
// Stack holds ATRAddr parameter
PRIM 0x07
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATRAddr</td>
<td>Length</td>
</tr>
</tbody>
</table>

The 2-byte ATRAddr is the address of the record that will be written to the ATR File. The output parameters Length is 1-byte in size and indicates the actual number of bytes written.

**Remarks**

The ATR file is an elementary file held in the root directory of the MULTOS device. Inside the file is one record per application loaded onto the MULTOS device. An application may write data to its own record using this primitive, but cannot effect the records of other applications. Note that the ATR File does not get returned by the MULTOS device as part of the Answer To Reset.

The ATR file record pointed to by ATRAddr must be formatted with the first byte giving the number of bytes in the record followed by the record itself. For example.

```
0x05,0x01,0x02,0x03,0x04,0x05
```

The ATR file data is copied from the byte after the segment address specified by the application.

The ATR record should not occupy the top byte of the stack.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

C  Set if the amount of data written is less than requested  
V  Unchanged  
N  Unchanged  
Z  Set if no data is written, cleared otherwise

**Primitive Set and Number**

Set Zero, Number 0x07

**Example**

The following example sets the ATR File record corresponding to the application to ten bytes that are held in the variable sATR.

```assembly
prmSetATRFileRecord EQU 0x07

sATR STATIC BYTE 7 = 0x06, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06
LOADA  sATR
PRIM   prmSetATRFileRecord
```
Set ATR Historical Characters
This primitive writes data to the historical characters an ATR.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x
✅  ✅  ✅  ✅  ✅  ✅

Syntax
// Stack holds HistAddr parameter
PRIM 0x08

Arguments
None.

Stack Usage

Stack In  HistAddr
Stack Out  Length

The 2-byte input parameter HistAddr is the address from which to copy those bytes that will be written to the historical characters of the ATR. The 1-byte output parameter Length indicates the total number of bytes actually written.

Remarks
The ATR Historical Characters at HistAddr must be formatted with the first byte giving the number of bytes in the record followed by the record itself. For example, if the value to be written was 0x01, 0x02, 0x03, 0x04, 0x05, then the value should be

0x05,0x01,0x02,0x03,0x04,0x05

A write of zero is acceptable and will erase any historical characters present in the ATR. The maximum write size is 15 bytes.

As of MULTOS 4 there is a primary and secondary ATR values. The first is returned on a cold reset and the second on a warm reset. An application may request permission to write to the historical characters of one of those ATR. An application can not request control of both ATR historical characters nor can multiple applications control them. The request for control is contained in the application load certificate used.

If an application attempts to write to the historical characters of an ATR controlled by another application, the write request will not be honoured and the CCR Z flag will be set.
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- **C**: Set if the amount of data written is less than requested
- **V**: Unchanged
- **N**: Unchanged
- **Z**: Set if no data is written, cleared otherwise

**Primitive Set and Number**

Set Zero, Number 0x08

**Example**

The following example sets the ATR Historical Characters to the application to eight bytes that are held in the variable sATR.

```assembly
prmSetATRHistoricalCharacters EQU 0x08

sHistATR STATIC BYTE 9 = 0x08, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
LOADA sHistATR
PRIM prmSetATRHistoricalCharacters
```
Set ATS Historical Characters

This primitive writes data to the Historical Characters of the MULTOS device’s ATS for ISO/IEC 14443 Type A contactless operation.

Availability

<table>
<thead>
<tr>
<th>Version</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTOS 4</td>
<td>☒</td>
</tr>
<tr>
<td>MULTOS 4.2</td>
<td>✅*</td>
</tr>
<tr>
<td>MULTOS 4.3.1</td>
<td>☒</td>
</tr>
<tr>
<td>MULTOS 4.3.2</td>
<td>✅*</td>
</tr>
<tr>
<td>MULTOS 4.4</td>
<td>✅*</td>
</tr>
<tr>
<td>MULTOS 4.5.x</td>
<td>✅*</td>
</tr>
</tbody>
</table>

*This primitive is mandatory if the device supports ISO/IEC 14443 Type A contactless communication.

Syntax

```
// Stack holds HistAddr parameter
PRIM 0x0E
```

Arguments

None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>HistAddr</td>
<td>Length</td>
</tr>
</tbody>
</table>

The 2-byte input parameter `HistAddr` is the address from which to copy those bytes that will be written to the historical characters of the ATR. The 1-byte output parameter `Length` indicates the total number of bytes actually written.

Remarks

The ATS Historical Characters at `HistAddr` must be formatted with the first byte giving the number of bytes in the record followed by the record itself. For example, if the value to be written was 0x01, 0x02, 0x03, 0x04, 0x05, then the value should be

```
0x05,0x01,0x02,0x03,0x04,0x05
```

A write of zero is acceptable and will erase any historical characters present in the ATS. The maximum write size is implementation specific. The ATS historical characters should not occupy the top byte of the stack.

Permission to update the ATS historical characters is requested in the application load certificate. Only one application can control them. If an application attempts to write to the historical characters of an ATS controlled by another application, the write request will not be honoured and the CCR Z flag will be set.

If the an application on a device that is configured to work over a contact interface only, then the CCR C flag is cleared and the CCR Z flag is set.
Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set if the amount of data written is less than requested
V  Unchanged
N  Unchanged
Z  Set if no data is written, cleared otherwise

Primitive Set and Number

Set Zero, Number 0x0E

Example

The following example sets the ATS Historical Characters for the application to the ten bytes that are held in the variable sHistATS.

```assembly
prmSetATSHistoricalCharacters EQU 0x0E
sHistATS STATIC BYTE 11 = 0x0A, 1, 2, 3, 4, 5, 6, 7, 8, 9, A

LOADA sHistATS
PRIM prmSetATSHistoricalCharacters
// Check CCR for result
// CCR.C set if bytes copied < bytes requested
// CCR.Z set if no bytes copied.
JLE err_incompletecopy
// error handling code would be found after the label
```

An alternative method of checking how much data was written is to compare the length byte left on the stack with the expected length. The following example shows this.

```assembly
sHistATS STATIC BYTE 11 = 0x0A, 1, 2, 3, 4, 5, 6, 7, 8, 9, A

LOADA sHistATS
PRIM prmSetATSHistoricalCharacters
// primitive leaves one byte length value on stack
// to check: load static length byte to stack
// and compare to length byte on stack
LOAD sHistATS, 1
CMPN , 1
JNE err_incompletecopy
// error handling code would be found after the label
```

Set FCI File Record

This primitive writes to the File Control Information (FCI) associated with the calling application.
Availability

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax

// Stack holds address of record
PRIM 0x11

Arguments

None.

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCIAddr</td>
<td>Length</td>
</tr>
</tbody>
</table>

The 2-byte input parameter FCIAddr is the location of the data to be written to the FCI. The data at FCIAddr must be formatted with the first byte indicating the length of the record in bytes, followed by the record itself. The 1-byte output parameter Length indicates the actual number of bytes written.

Remarks

This primitive allows an application to change the file control information available in a response to a SELECT FILE command when the application is selected.

The FCI record should not occupy the top byte of the stack.

No more than the specified number of bytes is written. The actual number written is returned on the stack. Note that the length of the FCI record is limited to the length given by the ALC.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Set if the amount of data written is less than requested
V  Unchanged
N  Unchanged
Z  Set if no data is written, cleared otherwise

Primitive Set and Number

Set zero, number 0x11
**Example**

The following example sets the FCI File record corresponding to the application to ten bytes that are held in the variable sFCI.

```assembly
prmSetFCIFileRecord EQU 0x11
sFCI STATIC BYTE 11 = 0x0A, 1,2,3,4,5,6,7,8,9,A

LOADA sFCI
PRIM prmSetFCIFileRecord
```
Set PIN Data
Sets data relating to the PIN which is either the local application PIN or the Global PIN depending on the access_list bit settings in the ALC. See Initialise PIN for details.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0x85, ElementId

Arguments

ElementId can take the following values:

- 0x00: Set the PIN Try Counter
- 0x01: Set the PIN Try Limit
- 0x03: PIN Verification Status (new in MULTOS 4.5.2)

Stack Usage

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
</tr>
</tbody>
</table>

Value is the one byte value to set. PIN verification Status must be given the values

- 0x5A = PIN is unverified
- 0xA5 = PIN is verified

Remarks

This primitive will abend if the PIN has not yet been initialised.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C: Unchanged
V: Unchanged
N: Unchanged
Z: Unchanged

Primitive Set and Number

Set one, number 0x85
**Set Silent Mode**
This primitive switches the MULTOS device into or out of silent mode. Silent mode ensures that no device unique information is returned by MULTOS’ card edge API. The “suspend” option switches off Silent mode temporarily until the next reset, when it will be reinstated.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xE3, Mode

**Arguments**
The 1 byte argument Mode is used to specify whether Silent Mode should be turned on or off or suspended as follows.

- Mode = 0: Turn off silent mode completely.
- Mode = 1: Turn silent mode on permanently on all interfaces.
- Mode = 2: Turn permanent silent mode on for contact and off for contactless
- Mode = 3: Turn permanent silent mode on for contactless and off for contact
- Mode = 4: Temporarily turn silent mode off
- Mode = 5: Turn silent mode back on after temporary disablement

**Stack Usage**

Stack In     {empty}
Stack Out    {empty}

**Remarks**
Silent mode only affects information returned directly from MULTOS via the card edge API in GET CONFIGURATION DATA, GET MANUFACTURER DATA, GET MULTOS DATA and OPEN MEL commands. It does not affect the information returned by MULTOS to applications using primitives.

**Condition Code**

```
C  V  N  Z
-  -  -  -
```
Unchanged.

Unchanged

Unchanged

Primitive set and number

Set one, number 0xE3
**Set Transaction Protection**

This primitive permits a series of writes to be treated as a single entity, which is then written or discarded in its entirety.

**Availability**

MULTOS 4    MULTOS 4.2    MULTOS 4.3.1    MULTOS 4.3.2    MULTOS 4.4    MULTOS 4.5.x

✓ ✓ ✓ ✓ ✓ ✓ ✓

**Syntax**

PRIM 0x80, Options

**Arguments**

The 1-byte argument *Options* is used to turn transaction protection on and off as well as indicating if the writes should be committed or discarded. See the Remarks section for the bit flag settings.

**Stack Usage**

Stack In {empty}

Stack Out {empty}

**Remarks**

The value of the *Options* argument is a bitmap as follows.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Any other values are undefined</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>Transaction protection off</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Transaction protection on</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>Discard changes</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>Commit changes</td>
</tr>
</tbody>
</table>

The bit 0 flag is only interrogated if transaction protection has been switched on in a previous call to the primitive. The bit 1 flag is always interrogated and sets transaction protection on or off.

Transaction protection is a mechanism that allows an application to commit several writes to non-volatile memory in an atomic fashion. When transaction protection is off, the default setting, each write is applied as the instruction is executed. However, when transaction protection is on writes to non-volatile memory are not applied immediately as is normally the case. They are only applied when the application explicitly commits the writes. If the application exits, delegates, or abnormally ends, then all uncommitted writes are discarded.

Uncommitted writes are not visible to the application.
Transaction protection applies to writes to Static and writes performed to system memory by any relevant primitive. It does not affect writes to Public and Dynamic, nor does it affect any writes that MULTOS may need to perform in order to support the cryptographic primitives or the Get Random Number primitive.

There may also be a limitation on the number of transactions which may be held pending at any one point in time. Again, this is dependent upon the memory availability within the platform.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged  

**Primitive Set and Number**

Set one, number 0x80

**Examples**

The following example shows two similar code snippets. The first does not make use of transaction protection, the second does.

```assembly
prmTransactionProtection EQU 0x80
TPOn EQU 2
TPOffandCommit EQU 1
TPOffandDiscard EQU 0

// Transaction protection off
SETB SB[0], 3
ADDB SB[0], 1
ADDB SB[0], 1
// result at SB[0] now 5

SETB SB[0], 3
PRIM prmTransactionProtection, TPOn
ADDB SB[0], 1
ADDB SB[0], 1
PRIM prmTransactionProtection, TPOffandCommit
// result at SB[0] now 4 as uncommitted writes
// are not available to an application
```
**SetContactlessSelectSW**

This primitive sets the value of the status word returned by MULTOS in the future when the application is selected on the contactless interface.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x06, SW1, SW2

**Arguments**

There are two 1-byte sized arguments. The value SW1 is the most significant byte of the status word and SW2 is the least significant byte.

**Stack Usage**

Stack In    {empty}
Stack Out   {empty}

There are no input or output parameters.

**Remarks**

When an application is successfully selected MULTOS returns a status word of 90 00. This primitive allows an application to set a different status word value to return on the contactless interface. It can only be reset by another call to this Primitive.

The existing SetSelectSW functionality remains unchanged and sets the SW1SW2 for both the contact and contactless response. If the application then wishes to distinguish between the two interfaces then it must call the new primitive to update the contactless select SW1SW2.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged  
V Unchanged  
N Unchanged  
Z Unchanged
**Primitive Set and Number**

Set Two, Number 0x06

**SetSelectSW**

This primitive sets the value of the status word returned by MULTOS when the application is selected in the future.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![checkmarks]

**Syntax**

PRIM 0x04, SW1, SW2

**Arguments**

There are two 1-byte sized arguments. The value SW1 is the most significant byte of the status word and SW2 is the least significant byte.

**Stack Usage**

Stack In  {empty}
Stack Out  {empty}

There are no input or output parameters.

**Remarks**

When an application is successfully selected MULTOS returns a status word of 90 00. This primitive allows an application to set a different status word value to return. Note that MULTOS will still route commands to the selected application, regardless of the SW set using this primitive.

The application’s Select SW will be retained after the MULTOS device is powered-off and can only be reset by another call to this Primitive.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged
Primitive Set and Number

Set Two, Number 0x04
**SHA-1**

This primitive calculates the SHA-1 hash digest of a message of arbitrary length.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

✓  ✓  ✓  ✓  ✓  ✓  ✓

**Syntax**

PRIM 0xCA

**Arguments**

None.

**Stack Usage**

Stack In: | lenMsg | addrHash | addrMsg |
Stack Out: | {empty} |

Each of the input parameters is 2-bytes in size. The value lenMsg is the size of the input to the SHA-1 algorithm. The second parameter addrHash is the location where the 20-byte hash digest will be written. The parameter addrMsg is the location of the input of size lenMsg.

**Remarks**

The primitive functions properly even if lenMsg is zero.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive set and number**

Set zero, number 0xCA
**Shift Left**

This primitive performs a bitwise shift left on a block of bytes.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Syntax**

```
// Stack holds BytesIn parameter
PRIM 0x02, Length, ShiftBits
```

**Arguments**

Both arguments are 1-byte in size. `Length` gives the size of the data block to be shifted and `ShiftBits` indicates the number of bits to shift.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>BytesIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>BytesOut</td>
</tr>
</tbody>
</table>

The input parameter `BytesIn` is of size `Length` and is the byte block to be shifted. The output parameter `BytesOut` is the byte block of size `Length` that holds the result of `ShiftBits` shift operations on `BytesIn`.

**Remarks**

This primitive bit-shifts data leftwards, filling the least significant bits with zeroes.

The effect of the primitive is undefined if any of the following is true:

- `ShiftBits` is zero
- `Length` is zero
- `ShiftBits >= 8 * Length`

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- C: Set if the last bit shifted out is a one, cleared otherwise
- V: Unchanged
- N: Unchanged
Z Set if the result is zero, cleared otherwise

**Primitive Set and Number**

Set two, number 0x02

**Example**

The following example pushes a word onto the stack and shows how the word is affected by successive calls to the Shift Left primitive.

```
prmShiftLeft EQU 0x02

PUSHW 0x0001
PRIM prmShiftLeft,2,4 //Stack=0x0010
PRIM prmShiftLeft,2,4 //Stack=0x0100
PRIM prmShiftLeft,2,4 //Stack=0x1000
```
**Shift Right**

This primitive performs a bitwise shift right on a block of bytes.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmark](image)

**Syntax**

```plaintext
// Stack holds ByteIn parameter
PRIM 0x03, Length, ShiftBits
```

**Arguments**

Both arguments are 1-byte in size. *Length* gives the size of the data block to be shifted and *ShiftBits* indicates the number of bits to shift.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>BytesIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>BytesOut</td>
</tr>
</tbody>
</table>

The input parameter *BytesIn* is of size *Length* and is the byte block to be shifted. The output parameter *BytesOut* is the byte block of size *Length* that holds the result of *ShiftBits* shift operations on *BytesIn*.

**Remarks**

This primitive bit-shifts data rightwards, filling the most-significant bits with zeroes.

The effect of the primitive is undefined if any of the following is true:

- *ShiftBits* is zero
- *Length* is zero
- *ShiftBits* >= 8 * *Length*

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

- **C** Set if the last bit shifted out is a one, cleared otherwise
- **V** Unchanged
- **N** Unchanged
Z Set if the result is zero, cleared otherwise

**Primitive Set and Number**

Set two, number 0x03

**Example**

The following example pushes a word onto the stack and shows how the word is affected by successive calls to the Shift Right primitive.

```
prmShiftRight EQU 0x03

PUSHW 0x1000 //Stack=0x1000
PRIM prmShiftRight,2,4 //Stack=0x0100
PRIM prmShiftRight,2,4 //Stack=0x0010
PRIM prmShiftRight,2,4 //Stack=0x0001
```
**Shift Rotate**

This primitive provides an efficient way of shifting and rotating a block of data by a variable number of bits.

**Availability**

MULTOS 4   MULTOS 4.2   MULTOS 4.3.1   MULTOS 4.3.2   MULTOS 4.4   MULTOS 4.5.x

[ ] [ ] [ ] [ ] [ ] [ ]

**Syntax**

PRIM 0x07, Mode, Direction

**Arguments**

Both arguments are 1-byte in size. Mode defines the function (0x01 = Shift, 0x02 = Rotate) and Direction defines the sense of the function (0x01 = Left, 0x02 = Right).

**Stack Usage**

Stack In  
NumBits  DataLen  DataAddr

Stack Out  
{empty}

All parameters are 2 bytes long. NumBits is the number of bits to shift / rotate by. DataLen is the length of the data (in bytes) of the data pointed to by DataAddr that is to be shifted / rotated.

**Remarks**

When shifting, vacated bits are filled with zero.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N  Unchanged
Z  Unchanged

**Primitive Set and Number**

Set two, number 0x07
**Store CCR**
The byte at the top of the stack is moved to the Condition Code Register.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmarks](Image)

**Syntax**

PRIM 0x06

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>setCCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
</tr>
</tbody>
</table>

The 1-byte input parameter setCCR is the value that will be stored in the condition control register byte.

**Remarks**

This primitive moves one byte from the stack to the CCR.

**Condition Code**

The bit flag values will be those specified by the value on the stack.

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
**Primitive Set and Number**

Set zero, number 0x06

**Example**

The following example sets the Condition Code Register to 00001001b by pushing 0x09 to the stack and then calling the primitive to move that value. This will now set the CCR C and the CCR Z bit flags. The branch instruction BLE will fire resulting in the code pointer moving to the address of LessThan label.

```assembly
prmStoreCCR EQU 0x06
PUSHB 0x09
PRIM prmStoreCCR
BLE LessThan

// This line will not be executed

LessThan
// Example code jumps here.
```
**Subtract BCDN**

This primitive subtracts two stack resident unsigned byte blocks of the same size, where the blocks hold binary coded decimal (BCD) values. The result is placed on the stack.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

- ☒
- ☑
- ☑
- ☑
- ☑
- ☑

**Syntax**

```c
//Stack holds Operand1, Operand2 bytes
PRIM 0x12, length
```

**Arguments**

The argument *length* gives the size of the byte blocks to be added.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>Output</td>
<td></td>
</tr>
</tbody>
</table>

The parameters *Operand1* and *Operand2* are both of size *length* and these are the values that will be added. The parameter *Output* is of size *length* and holds the result of the addition.

**Remarks**

The *length* value is specified using a single byte. Therefore, the maximum length of a block is 255 bytes.

The value designated by an operand should be in BCD format. If not in BCD format, the processing in MULTOS device will abnormally end the application.

The CCR C flag is set if the result of the operation is greater than that which can be held in *length* bytes. The Z flag is set if the result is zero.

The operation performed is

\[ \text{Output} = \text{Operand1} - \text{Operand2} \]
**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

C  Set if a carry occurs, cleared otherwise.
V  Unchanged
N  Unchanged
Z  Set if the result is zero, cleared otherwise.

**Primitive Set and Number**

Set one, number 0x12

**Examples**

The following examples illustrate how to use the primitive as well as the CCR settings.

```
prmSubtractBCDN  EQU  0x12

PUSHB           0
PUSHB           1
PRIM  prmSubtractBCDN, 1
// result on stack is 99 CCR C set and CCR Z cleared
PUSHW 0x0150
PUSHW 0x0100
PRIM  prmSubtractBCDN, 2
// result on stack 0x0100 CCR C and CCR Z both cleared
```
**Triple DES Decipher**

This primitive performs a Triple DES Decipher on an eight byte block of memory in accordance with [FIPS46-3].

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

```
PRIM 0xD8
```

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>{empty}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 2 byte parameter KeyAddr is the starting address of the Triple DES keys to be used. The 1 byte parameter KeyLen is the length in bytes of the Triple DES keys at address KeyAddr. The 2 byte parameter OutputAddr is the starting address of the resultant 8-bytes of plaintext. The 2 byte parameter InputAddr is the starting address of the 8-bytes of ciphertext.

**Remarks**

This primitive performs the Triple DES decipher operation on an 8-byte block of memory. The Triple DES keys $K_1$, $K_2$ and optional $K_3$ are held in an 16 or 24 byte block. If KeyLen is 16 then $K_3$ shall equal $K_1$.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input.

This primitive is only available to an application if “Strong Cryptography” is set on in the application’s access_list when loaded.

**Condition Code**

```
C  V  N  Z
-  -  -  -
```

C Unchanged
Primitive Set and Number

Set zero, number 0xD8

Example

The following example declares 16 bytes of static memory to hold the two Keys (128-bits), the ciphertext is held as session data, while the resulting plaintext will be written to public. The address for each of these is loaded onto the stack and the Triple DES Decipher primitive is called.

```plaintext
prm3DESDecipher  EQU  0xD8
KEYLEN  EQU  16

sKey  STATIC BYTE 16 =
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D
,0x0E,0x0F,0x10
dCiphertext  DYNAMIC BYTE 16
pPlaintext PUBLIC BYTE 16

LOADA   sKey
PUSHB   KEYLEN
LOADA   pPlaintext
LOADA   dCiphertext
PRIM    prm3DESDecipher
```
**Triple DES Encipher**

This primitive performs Triple DES Encipher on an eight byte block of memory in accordance with [FIPS46-3].

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xD9

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>KeyAddr</th>
<th>KeyLen</th>
<th>OutputAddr</th>
<th>InputAddr</th>
<th>{empty}</th>
</tr>
</thead>
</table>

The 2 byte parameter KeyAddr is the starting address of the Triple DES keys to be used. The 1 byte parameter KeyLen is the length in bytes of the Triple DES keys at address KeyAddr. The 2 byte parameter OutputAddr is the starting address of the resultant 8-bytes of ciphertext. The 2 byte parameter InputAddr is the starting address of the 8-bytes of plaintext.

**Remarks**

This primitive performs the Triple DES encipher operation on an 8-byte block of memory. The Triple DES keys $K_1$, $K_2$ and optional $K_3$ are held in a 16 or 24 byte block. If KeyLen is 16 then $K_3$ shall equal $K_1$.

The output is written at the specified segment address and this may be the same as the address of the input; i.e., the output overwrites the input.

This primitive is only available to an application if “Strong Cryptography” is set on in the application’s access_list when loaded.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C Unchanged
Primitive Set and Number

Set zero, number 0xD9

Example

The following example declares 24 bytes of static memory to hold the three Keys (192-bits), the plaintext is held as session data, while the resulting ciphertext will be written to public. The address for each of these is loaded onto the stack and the Triple DES Encipher primitive is called.

```
prm3DESEncipher EQU 0xD9
KEYLEN EQU 24

sKey STATIC BYTE 24 =
0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,
0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18

dPlaintext DYNAMIC BYTE 16

pCiphertext PUBLIC BYTE 16

LOADA sKey
PUSHB KEYLEN
LOADA pPlaintext
LOADA dCiphertext
PRIM prm3DESEncipher
```
Unpad
This primitive identifies an un-padded message from a padded message.

Availability
MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

Syntax
PRIM 0x16, PadScheme

Arguments
The 1-byte parameter PadScheme specifies the unpadding scheme, as follows.
- 0x01 and 0x02: Zero or more bytes of 0x00 are searched from the end of the message until an 0x80 is encountered or there are no more bytes to search. The length of the resultant unpadded message is then returned.

Stack Usage
Stack In
LenMsg  AddrMsg  
Stack Out
LenMsgUnpadded

- The 2-byte parameter LenMsg specifies the length in bytes of the padded message.
- The 2-byte parameter AddrMsg specifies the segment address of the padded message.

- The 2-byte result LenMsgUnpadded is the length in bytes of the unpadded message.

Remarks
The padded block is not modified in any way. The result lenMsgUnpadded contains the length of the unpadded message within the padded message and the calling application is responsible for manipulating the unpadded part of the message as required.

If no 0x80 byte is encountered within the padded message, then lenMsgUnpadded is zero.

The primitive abends, if an invalid PadScheme value is supplied.

Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged
V  Unchanged
N Unchanged
Z Unchanged

**Primitive Set and Number**

Set zero, number 0x16
Update Process Events
This primitive enables or disables individual events for an application according to the mask provided.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.x</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.1 / 2 / 3</th>
<th>MULTOS 4.5.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Syntax

PRIM 0x08, mask

Arguments

mask specifies a two byte bitmap as follows.

- 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

Stack Usage

Stack In: {empty}
Stack Out: {empty}

Remarks

Condition Code

- - - - C V N Z

C Unchanged.
V Unchanged
N Unchanged
Z Unchanged.

Primitive set and number
Set two, number 0x08
**Update Session Size**

This primitive temporarily updates the total size of the application’s session memory.

**Availability**

MULTOS 4   MULTOS 4.2   MULTOS 4.3.x   MULTOS 4.4   MULTOS 4.5.1   MULTOS 4.5.2

\[ \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \]

**Syntax**

PRIM 0x04

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>Stack Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionSize</td>
<td>Result</td>
</tr>
</tbody>
</table>

SessionSize specifies the total size of session data in bytes. SessionSize is a 2-byte value.

**Result** holds the result of the operation as follows:

- 0 = update failed as either SessionSize is either more than the session size held in the application’s ALC or there is insufficient free RAM to accommodate the increase in the size of the application’s session.
- 1 = update succeeded.

**Remarks**

1. As of MULTOS 4.5.3 the check against the ALC size is optional.
2. After calling this primitive, function call returns are not possible.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- C  Unchanged
- V  Unchanged
- N  Unchanged
- Z  Unchanged

**Primitive Set and Number**

Set zero, number 0x04

© 2019 MAOSCO Limited. MAO-DOC-TEC-006 v1.56
MULTOS is a registered trademark of MULTOS Limited.
**Update Static Size**

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.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0x04, Options

**Arguments**

The 1 byte argument *Options* is used to specify the size of the stack parameter *StaticSize*.

- Options = 0: 32-bit (4-byte) *StaticSize*.
- Options = 1: 64-bit (8-byte) *StaticSize*.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>StaticSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Out</td>
<td>Result</td>
</tr>
</tbody>
</table>

*StaticSize* specifies the total size of Static in bytes. *StaticSize* can either be a 32-bit (4-byte) or a 64-bit (8-byte) value depending upon the value of *Options*.

*Result* holds the result of the operation as follows:

- 0 = update failed as either *StaticSize* is either more than the Static size specified in the application’s ALC or there is insufficient free NVM to accommodate the increase in the size of the application’s Static.
- 1 = update succeeded

**Condition Code**

```
 C  V  N  Z
 -  -  -  -  -  -  -  -
```

- C Unchanged
- V Unchanged
- N Unchanged
- Z Unchanged

**Primitive Set and Number**

Set one, number 0x04
**Verify Asymmetric and Retrieve General**

This primitive verifies an asymmetric signature of a message of arbitrary length.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Syntax**

PRIM 0xE2, Mode

**Arguments**

The argument *Mode* indicates whether a protected or unprotected variant of RSA is to be used. Defined values for *Mode* are 0x01 for standard mode with a public exponent of 3 and 0x81 for standard mode with a public exponent of 3 using an “unprotected” variant. The effect of any other value is undefined.

**Stack Usage**

Stack In

<table>
<thead>
<tr>
<th>MsgLen</th>
<th>ModLen</th>
<th>ModAddr</th>
<th>InAddr</th>
<th>CertType</th>
<th>HashLen</th>
<th>HashAddr</th>
</tr>
</thead>
</table>

Stack Out {empty}

All the parameters except *CertType* are 2 bytes in size. The value *MsgLen* is either the size in bytes of the data that has been signed and the signature or the length of data to recover, while *ModLen* is the length in bytes of the public key modulus value to use in order to verify the signature. *ModAddr* and *InAddr* are the locations of the public key modulus and message respectively. The 1-byte parameter *CertType* indicates whether the signature to verify is in MULTOS 3 or MULTOS 4 format. The parameter *HashLen* indicates the size in bytes of the modulus value used in calculating the asymmetric hash digest value. Finally *HashAddr* is the location of the hash modulus of size *HashLen*.

**Remarks**

The *CertType* can take a value of 0x03 indicating a MULTOS 3 certificate format or 0x04 indicating a MULTOS 4 certificate format. Any other value is undefined. A MULTOS 4 signature block consists of:

- 16-byte asymmetric hash digest
- *n*-byte data
- 8-byte random padding
- 8-byte fixed padding

The value *n* is found by subtracting 32 from the modulus length.
A MULTOS 3 signature block consists of:

- 16-byte asymmetric hash digest
- $n$-byte data

Here the value $n$ is found by subtracting 16 from the modulus length.

When $MsgLen$ is less than or equal to $ModLen$, $MsgLen$ is interpreted to be the length of data to recover from the signature component. The signature component is found at $InAddr$ and is considered to be of size $ModLen$. The data recovered of size $MsgLen$ is returned starting at the least significant end of decrypted signature block.

When $MsgLen$ is greater than $ModLen$, the value at $InAddr$ is considered to consist of a plaintext header and signature, where the signature is of size $ModLen$. The data recovered will include the plaintext followed by the recovered data. Note that the data will not include the asymmetric hash digest value.

RSA is the only supported signature verification algorithm. The public exponent is always considered to have a value of 3.

The recovered message will overwrite the input message.

The unprotected variant has some restrictions. They are:

- The public key modulus must be in static memory. If it is not, the results can not be guaranteed and may result in an abnormal end to application execution.
- The message must be either in static memory or in public memory. If it is not, the results can not be guaranteed and may result in an abnormal end to application execution.
- If the message is in public memory, it must start at the base of public

### Condition Code

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**C** Unchanged

**V** Unchanged

**N** Unchanged

**Z** Set if the signature is correct, cleared otherwise

### Primitive Set and Number

Set One, Number 0xE2
**Verify PIN**

Verifies the PIN which is either the local application PIN or the Global PIN depending on the access_list bit settings in the ALC. See Initialise PIN for details.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5.x

![Checkmark or Cross]

**Syntax**

PRIM 0xE7

**Arguments**

None.

**Stack Usage**

<table>
<thead>
<tr>
<th>Stack In</th>
<th>PINLen</th>
<th>PINAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack Out</td>
<td>Status</td>
<td></td>
</tr>
</tbody>
</table>

PINAddr is a 2 byte address of a buffer containing the PIN to be verified. PinLength (1 byte) is the length of the PIN pointed to by PINAddr.

Status is a 2 byte value, 0x5AA5 for PIN verified and 0xA55A for PIN NOT verified. A value of 0xAAAA indicates that verification has been disabled.

**Remarks**

This primitive will abend if the PIN has not yet been initialised.

This primitive does NOT maintain the value of the PIN Try Counter. The application must do this.

**Condition Code**

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C  Unchanged  
V  Unchanged  
N  Unchanged  
Z  Unchanged

**Primitive Set and Number**

set zero, number 0xE7
APDU Commands

This section provides an alphabetical listing of APDU commands defined for MULTOS. The APDU commands defined for MULTOS step/one are available under licence in a separate document [OFFCARD]. The areas covered here are:

Enablement

Enablement is the initialisation (or pre-personalisation) of a MULTOS device’s configuration, ready for the loading and deleting of applications. The APDU command used is SET MSM CONTROLS.

Application Loading

An application is loaded to a MULTOS device using APDU commands, CREATE MEL APPLICATION, LOAD APPLICATION SIGNATURE, LOAD CODE, LOAD DATA, LOAD DATA (Extended), LOAD DIR FILE RECORD, LOAD FCI RECORD, LOAD KTU CIPHERTEXT and OPEN MEL APPLICATION.

Application Deletion

An application is deleted from a MULTOS device using the APDU command DELETE MEL APPLICATION.

ISO Commands

MULTOS devices also support ISO defined commands, GET RESPONSE, READ BINARY, READ RECORD and SELECT FILE.

Device Information

Details about a particular MULTOS device can be obtained by APDU commands, CHECK DATA, GET CONFIGURATION DATA, GET DATA, GET MANUFACTURER DATA and GET MULTOS DATA.

Other

Other APDU commands supported are CARD UNBLOCK and GET PURSE TYPE.

Usage Notes

The subsections that follow provide a list of status word values that can be returned in response to the command. There are two cases that have not been included due to their ubiquity. In every case successful completion is indicated by a status word value of 90 00. In those cases where data is returned it is also possible to receive a status word of 61 xx if the La value is greater than the Lc value.

The APDU command table values are all hexadecimal despite not having the ‘0x’ prefix.
**CARD UNBLOCK**

The Card Unblock command is used by an IFD to unblock a MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 3</th>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>08</td>
<td>00</td>
<td>00</td>
<td>08</td>
<td>CUBMAC</td>
<td>-</td>
</tr>
</tbody>
</table>

The CUBMAC is an 8-byte value device unique value supplied by the KMA.

**APDU Response**

Status word values that can be returned are:

- 9D 60  MAC verification failed
- 9D 61  Maximum number of unblocks reached
- 9D 62  This is not a blocked device

No data is returned in response to this command.

**Remarks**

Card Unblock command unblocks a device that has been previously blocked by an application using the Card Block primitive.

This command requires a Card Unblock MAC, CUBMAC, to be sent as command data. The CUBMAC is specific to each MULTOS device and can be obtained by the MULTOS Issuer from the MULTOS KMA.

The Card Unblock command can only be used once during the lifetime of a MULTOS device.

The Card Unblock command has been removed from MULTOS 4.3 as the mechanism for blocking and unblocking a card has been revised.

**Standards**

MULTOS
CHECK DATA

The Check Data command checks a specified area of a MULTOS device’s memory in order to prove its authenticity. The challenge value is a random number agreed upon by a Personalisation Bureaux and the MULTOS OS Implementer. Both parties then send the same command and parameters to their devices, if the responses match the MULTOS devices are genuine.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Conditional Usage and Security

This command is only available before enablement.

APDU Command

```
+----+----+----+----+---+----------------+-----+
<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>06</td>
<td>00</td>
<td>00</td>
<td>0C</td>
<td>challenge value +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
<td>or</td>
<td>or</td>
<td>check data start address +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>01</td>
<td>0E</td>
<td>check data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
```

See the Remarks section for more information on the permitted P1, P2 and Lc combinations.

The challenge value is an 8-byte data block. The start address and data length are variable in length as explained in the Remarks section.

APDU Response

Status word values that can be returned are:

9D 31  Check data parameter is incorrect (invalid length)
9D 32  Check data parameter is incorrect (illegal memory check area)
9D 63  Crypto function not supported
6A 83  Function not supported

A 16-byte check data digest is returned upon successful execution of the command.

Remarks

The CHECK DATA command takes as a parameter the physical address of the device memory being checked. In all cases the address offset specified is a logical offset from the beginning of the memory area to be checked.
The response “Crypto function not supported” is returned if the contactless interface is in use. If the command is issued after the device has been enabled, the response “Function not supported” will be returned.

As of MULTOS 4.2 the functionality has also been extended so that it can handle implementations which have more than 64K of ROM and EEPROM.

For devices where the combined ROM and EEPROM size is less than 64K:

- P1 must be set to 0x00
- P2 must be set to 0x00
- The command data field ‘check_data_start_address’ is a 16 bit value
- The command data field ‘check_data_length’ is a 16 bit value
- The Lc value must be 0x0C
- If P1 is set to 0x01, the response will be 9D63 ‘function not supported’
- Only APDU command possible is: BE 06 00 00 0C [data field] 10

For devices where the combined ROM and EEPROM size is greater than or equal to 64K:

- P1 must be set to 0x01
- P2 may be either 0x00 or 0x01, where 0x00 indicates that the check data operation should be done on the ROM area and where 0x01 indicates that the check data operation should be done on the EEPROM area
- The command data field ‘check_data_start_address’ is a 24 bit value
- The command data field ‘check_data_length’ is a 24 bit value
- The Lc value must be 0x0E
- If P1 is set to 0x00, the response will be 9D63 ‘function not supported’
- Possible APDU commands are:
  - BE 06 01 00 0E [data field] 10
  - BE 06 01 01 0E [data field] 10

**Standards**

MULTOS
**CREATE MEL APPLICATION**

This command is the last sent in the application loading process. The Application Load Certificate is sent as data, which allows MULTOS to retrieve and authenticate the application and associated data.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>4.3.1</td>
<td>4.3.2</td>
<td>4.4</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>16</td>
<td>00</td>
<td>00</td>
<td>var.</td>
<td>Application Load Certificate</td>
<td>-</td>
</tr>
</tbody>
</table>

The data sent is the Application Load Certificate that corresponds to the application being loaded.

**APDU Response**

Status word values that can be returned are:

- 6A 81 Retry counter expired
- 9D 50 Invalid MCD Issuer Product ID
- 9D 51 Invalid MCD Issuer ID
- 9D 52 Invalid Set MSM Controls Data Date
- 9D 53 Invalid MCD number
- 9D 54 Reserved Field Error
- 9D 55 Reserved Field Error
- 9D 56 Reserved Field Error
- 9D 57 Reserved Field Error
- 9D 05 Incorrect certificate type
- 9D 15 Application not open
- 9D 19 Invalid certificate
- 9D 1A Invalid signature
- 9D 1B Invalid key transformation unit
- 9D 1E Application signature does not exist
- 9D 1F KTU does not exist
- 9D 63 Crypto function not supported

No data is returned in response to this command.
Remarks

Create MEL Application is an MSM command and is always available. However, in contactless mode, this command can only be executed when the cryptographic coprocessor is available. When it is not, the device will return “Crypto function not supported”.

From MULTOS 4 the Application Load Certificate may be transmitted over several CREATE MEL APPLICATION commands. The device will build the certificate using the data components in the order in which they arrive.

If the CREATE MEL APPLICATION command fails then a retry counter is reduced by one. When the counter reaches 0 the device will not allow any further loads. Note that this counter value can not be incremented. See the MULTOS Implementation Report [MIR] to find out the retry counter value.

From MULTOS 4.2 there is an optional retry counter to limit the number of successful Application load operations a single device will perform. This prevents an attacker driving an unlimited number of operations using the device’s secret key by repeatedly loading and deleting an application which could be a benefit for DPA attacks. To find out the retry counter value see the MULTOS Implementation Report [MIR].

When the function is not completed in a low power environment the device will abnormally end execution.

Standards

MULTOS
**DELETE MEL APPLICATION**

This command is used to delete an application from a MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>18</td>
<td>00</td>
<td>00</td>
<td>var.</td>
<td>Application Delete Certificate</td>
<td>-</td>
</tr>
</tbody>
</table>

The data sent is the Application Delete Certificate that corresponds to the application to be removed from the MULTOS device.

**APDU Response**

Status word values that can be returned are:

- 6A 81 Retry counter has expired
- 9D 50 Invalid MCD Issuer Product ID
- 9D 51 Invalid MCD Issuer ID
- 9D 52 Invalid Set MSM Controls Data Date
- 9D 53 Invalid MCD number
- 9D 05 Incorrect certificate type
- 9D 15 Application not open
- 9D 19 Invalid certificate
- 9D 20 Application not loaded
- 9D 63 Crypto function not supported

No data is returned in response to this command.
**Remarks**

DELETE MEL APPLICATION is an MSM command and is always available. However, in contactless mode, this command can only be executed when the cryptographic coprocessor is available. When it is not, the device will return “Crypto function not supported”.

From MULTOS 4 the Application Delete Certificate may be transmitted over several CREATE MEL APPLICATION commands. The device will build the certificate using the data components in the in which they arrive.

If the DELETE MEL APPLICATION command fails then a retry counter is reduced by one. When the counter reaches 0 the device will not allow any further deletions. Note that this counter value can not be incremented. See the MULTOS Implementation Report [MIR] to find out the retry counter value.

When the function is not completed in a low power environment the device will abnormally end execution.

**Standards**

MULTOS
**FREEZE**

This command can be used with MULTOS devices that support step/one application loading and deleting.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xBE</td>
<td>0x1A</td>
<td>-</td>
<td>-</td>
<td>0x91 or 0xD8</td>
<td>freeze_certificate</td>
<td></td>
</tr>
</tbody>
</table>

The freeze certificate and signature are generated by the step/one certificate generation tools appropriate to the product vendor.

**APDU Response**

Status word values that can be returned are:

- 0x9000 Success
- 0x6A81 Function not supported
- 0x9D05 Incorrect certificate type
- 0x9D19 Invalid certificate
- 0x9D1D Enablement data not set
- 0x9D64 No applications loaded

**Standards**

MULTOS
step/one
**GET CONFIGURATION DATA**

This command allows a terminal to retrieve extended information about the MULTOS device. The command assists device management by providing the immediate determination of a device’s configuration and capabilities without reference to an alternate data source. Where appropriate, the data is available before and after a device is enabled.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>10</td>
<td>Token MSB</td>
<td>Token LSB</td>
<td>-</td>
<td>-</td>
<td>00</td>
</tr>
</tbody>
</table>

The values in P1 and P2 represent the most significant byte and least significant byte of the request token. Acceptable token values are given in the table in the Remarks section.

**APDU Response**

Status word values that can be returned are:

- 61 xx   xx byte of data to retrieve using Get Response
- 6B 00   Wrong Parameters P1,P2

Data is returned in response to this command. See the table in the Remarks section for further details.
Remarks

The command returns data based on the P1 and P2 parameters. The table below has more details.

<table>
<thead>
<tr>
<th>Token</th>
<th>Request</th>
<th>Data Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 00</td>
<td>Platform Identification</td>
<td>os_type + os_version + supported_functions + product_name</td>
</tr>
<tr>
<td>0x01 00</td>
<td>Largest ALU Possible</td>
<td>max_alu_size</td>
</tr>
<tr>
<td>0x02 00</td>
<td>Communication Transfer Parameters</td>
<td>comms_tx_parameters</td>
</tr>
<tr>
<td>0x03 00</td>
<td>ATR Control</td>
<td>cold_reset_application_id + warm_reset_application_id</td>
</tr>
<tr>
<td>0x04 00</td>
<td>AMD Version Information</td>
<td>amd_version_data</td>
</tr>
<tr>
<td>0x05 00</td>
<td>Codelets available</td>
<td>codelet_list</td>
</tr>
<tr>
<td>0x06 00</td>
<td>Applications loaded*</td>
<td>{application_id + application_memory_allocated}</td>
</tr>
<tr>
<td>0x07 00</td>
<td>MKD_PKC*</td>
<td>MULTOS_pk_certificate</td>
</tr>
<tr>
<td>0x08 00</td>
<td>Codelet checksums*</td>
<td>The 4 byte MULTOS checksum for each codelet listed in the same orders as the codelets in token 0x0500</td>
</tr>
<tr>
<td>0x09 00</td>
<td>ATS Control*</td>
<td>application_ATS_selected.application_id or MULTOS_aid</td>
</tr>
<tr>
<td>0x0A 00</td>
<td>Build Number**</td>
<td>The build number of the implementation. Encoding defined by the implementer.</td>
</tr>
<tr>
<td>0x0B 00</td>
<td>Primitives Supported**</td>
<td>A list of bits, one bit per possible primitive (4 sets of 256 primitives, with a &quot;1&quot; indicating that the primitive is supported. The first byte contains the bits for set 0 primitives 0-7 (held in bits 0-7), the second byte for set 0 primitives 8-17 and so on.</td>
</tr>
<tr>
<td>0x0C 00</td>
<td>Chip Identity Data**</td>
<td>Silicon manufacturer specific chip identity data.</td>
</tr>
</tbody>
</table>

Starred token values are not implemented on all masks but are mandatory from MULTOS 4.5* and MULTOS 4.5.1** onwards.

The Platform Identification request returns product specific values.

The Largest ALU Possible request returns the maximum size of an ALU that can be loaded.
The Communications Transfer Parameters request returns the device’s x and y parameters along with an indication of what contactless protocol, if any, is supported.

The ATR Control request returns two 17-byte AIDs. Both are formatted such that the first byte is the length of the following ATR value, the next bytes are the ATR and the field is padded with 0xFF up to the 17-byte size limit. If no application controls an ATR the length byte will be set to 0x00 followed by 17 bytes of 0xFF.

The AMD Version Information request returns a 2-byte AMD ID and a 2-byte AMD version value.

The Codelets available request returns a list of 2-byte Codelet IDs.

The Applications loaded request returns a list of applications loaded and their memory allocations. Each entry in the list consists of a 17-byte application ID and 3-byte application_memory_allocated data field.

The MKD_PKC request returns the MCD’s Certified Public Key.

Standards

MULTOS
**GET DATA**

This command allows an IFD to retrieve the Data Objects (DO) from any generic MAOS device. Specifically for MULTOS, this command returns data objects to identify the MULTOS platform type and other objects in a structure as agreed with Global Platform.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

*Conditionally Usage and Security*

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CA</td>
<td>00</td>
<td>66</td>
<td>-</td>
<td>-</td>
<td>00</td>
</tr>
</tbody>
</table>

**APDU Response**

This command will always execute successfully. The card data DOs are returned in response to this command.

**Remarks**

The DO returned is of the form:

```
   card_data_Dos
   66 Len  card_data_discretionary_DOs

   P3 Len  tag_issuing_authority  card_data_DO_items

   06 Len  tag_issuing_authority_OID
   TBA - MAOSCO / GP

   platform_identification  logical_id_qualifier

   platform_id_tag  platform_id_value
   logical_id_tag  logical_id_value

   TBA

   06 Len  platform_id_OID
   e.g. (multos_OID, 0, 5, 01)

   TBA - MAOSCO
```

The usage of P1 and P2 is as specified in ISO/IEC 7816-4 Section 6.9.3.
Standards

ISO/IEC 7816-4, Section 6.9.
GlobalPlatform
MULTOS
**GET MANUFACTURER DATA**

This command allows a terminal to retrieve information about the hardware of the MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>02</td>
<td>00</td>
<td>00</td>
<td>-</td>
<td>-</td>
<td>16</td>
</tr>
</tbody>
</table>

**APDU response**

This command will always execute successfully. Data is returned by this command.

**Remarks**

The data returned by this command is of the form:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size(bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC Manufacturer ID</td>
<td>1</td>
</tr>
<tr>
<td>IC Type</td>
<td>1</td>
</tr>
<tr>
<td>ROM IC Details</td>
<td>2</td>
</tr>
<tr>
<td>MCD ID</td>
<td>6</td>
</tr>
<tr>
<td>Initialisation Date</td>
<td>7</td>
</tr>
<tr>
<td>Processor Page Size</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Transmit TPDU Size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Receipt TPDU Size</td>
<td>2</td>
</tr>
</tbody>
</table>

**Standards**

MULTOS
**GET MULTOS DATA**
This command allows an IFD to retrieve information about the MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>-</td>
<td>-</td>
<td>7F</td>
</tr>
</tbody>
</table>

**APDU response**

This command will always execute successfully. Data is returned by this command.
### Remarks

The data returned is of the form:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTOS Version Number</td>
<td>2</td>
</tr>
<tr>
<td>IC Manufacturer ID</td>
<td>1</td>
</tr>
<tr>
<td>Implementer ID</td>
<td>1</td>
</tr>
<tr>
<td>MCD ID</td>
<td>6</td>
</tr>
<tr>
<td>Product ID</td>
<td>1</td>
</tr>
<tr>
<td>Issuer ID</td>
<td>4</td>
</tr>
<tr>
<td>MSM Controls Data Date</td>
<td>1</td>
</tr>
<tr>
<td>MCD Number</td>
<td>8</td>
</tr>
<tr>
<td>RFU</td>
<td>80</td>
</tr>
<tr>
<td>Maximum Dynamic Size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Public Size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum DIR File Record Size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum FCI Record Size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum ATR Historical Byte Record Size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum ATR File Record Size</td>
<td>2</td>
</tr>
<tr>
<td>MULTOS Public Key Certificate Length</td>
<td>2</td>
</tr>
<tr>
<td>Security Level</td>
<td>1</td>
</tr>
<tr>
<td>Certification Method ID</td>
<td>2</td>
</tr>
<tr>
<td>Application Signature Method ID</td>
<td>2</td>
</tr>
<tr>
<td>Encipherment Descriptor</td>
<td>2</td>
</tr>
<tr>
<td>Hash Method ID</td>
<td>2</td>
</tr>
</tbody>
</table>

The fields Product ID, Issuer ID, MSM Controls Data Date, MCD Number and RFU can collectively be referred to MSM Issuer Permissions.

### Standards

MULTOS
**GET PURSE TYPE**

This command identifies which Mondex Purse Type a MULTOS device can support: Originator or Non-originator.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

The command is a Master File command and, therefore, is only available when the Master File has been selected.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>04</td>
<td>00</td>
<td>00</td>
<td>-</td>
<td>-</td>
<td>01</td>
</tr>
</tbody>
</table>

**APDU Response**

This command will always execute successfully. One byte of data is returned by this command.

**Remarks**

A return value of 0x4F indicates that the device can support an originator purse, while a value of 0xB0 indicates a non-originator. Any other values are undefined.

**Standards**

MULTOS
GET RESPONSE
This command is issued by an IFD in response to a previous status word of 61 xx, where the xx indicates the number of bytes to retrieve.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Conditional Usage and Security

This command is always available.

APDU Command

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>C0</td>
<td>00</td>
<td>00</td>
<td>-</td>
<td>-</td>
<td>var.</td>
</tr>
</tbody>
</table>

The Le to use is dependent on the value given least significant byte of a 61 xx status word issued in response to the command immediately preceding.

APDU Response

Status word values that can be returned are:

- 62 81 Part of returned data may be corrupted
- 67 00 Wrong length, Le field incorrect
- 6A 86 Incorrect parameters P1,P2
- 6C xx Wrong length, xx indicates actual length

The data returned is of variable length.
Remarks

The class byte may also be set to the same value as the last command; i.e., the command that generated the response data.

In general, under the T = 0 transport protocol a status word of 61 xx is returned when the length of data to be returned is greater than the expected length; i.e., La > Le. When that status word is issued an IFD may send the command GET RESPONSE.

If MULTOS receives an unexpected GET RESPONSE command it will be routed to the currently selected application, or an error returned if no application is selected.

Standards

MULTOS
ISO 7816, Part 4
**LOAD APPLICATION SIGNATURE**

This command is used to load the Application Signature of an ALU. The Application Signature may be divided into component blocks, each of which is individually transmitted using this command.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is available after enablement and must follow the OPEN MEL APPLICATION command.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>28</td>
<td>MSB Starting Offset</td>
<td>LSB Starting Offset</td>
<td>var.</td>
<td>Application signature component</td>
<td>-</td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

**APDU Response**

Status word values that can be returned are:

9D 15 Application not open
9D 17 Invalid offset

**Remarks**

The Application Signature components can be sent in any order. The parameter bytes P1 and P2 are used to indicate the address within the reserved application signature memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

**Standards**

MULTOS
**LOAD CODE**

This command is used to load the Code of an ALU. The Code may be divided into component blocks, each of which is individually transmitted using this command.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
<th>MULTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>4.3.1</td>
<td>4.3.2</td>
<td>4.4</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is available after enablement and must follow the OPEN MEL APPLICATION command.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>24</td>
<td>MSB Starting Offset</td>
<td>LSB Starting Offset</td>
<td>var.</td>
<td>Code component</td>
<td>-</td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

**APDU Response**

Status word values that can be returned are:

9D 15  Application not open
9D 17  Invalid offset

**Remarks**

The Code components can be sent in any order. The parameter bytes P1 and P2 are used to indicate the address within the reserved Code memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

**Standards**

MULTOS
**LOAD DATA**

This command is used to load the Data of an ALU. The Data component may be divided into component blocks of data, each of which is individually transmitted using this command.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is available after enablement and must follow the OPEN MEL APPLICATION command.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>26</td>
<td>MSB</td>
<td>LSB</td>
<td>var.</td>
<td>Data component</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Starting Offset</td>
<td>Starting Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

**APDU Response**

Status word values that can be returned are:

- 9D 15 Application not open
- 9D 17 Invalid offset

**Remarks**

The Data components can be sent in any order. The parameter bytes P1 and P2 are used to indicate the address within the reserved Data memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

**Standards**

MULTOS
LOAD DATA (Extended)

Description

This command is used to load the Data of an Application or an Extended Data Application containing static data greater than 64k. The Data component may be divided into component blocks of data, each of which is individually transmitted using this command.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Conditional Usage and Security

This command is available after enablement and must follow the OPEN MEL APPLICATION command.

APDU Command

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>26</td>
<td>MSB</td>
<td>LSB</td>
<td>var.</td>
<td>Data component</td>
<td>-</td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

APDU Response

Status word values that can be returned are:

9D 15 Application not open
9D 17 Invalid offset

Remarks

The Data components can be sent in any order.

For a normal Application, the parameter bytes P1 and P2 are used to indicate the address within the reserved Data memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

For an Extended Data Application, the parameter bytes P1 and P2 are used to indicate the block number of the 255-byte blocks within the reserved Data memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the block number. Note that the block number uses zero based counting.
Standards

MULTOS
LOAD DIR FILE RECORD

This command is used to load the Directory File Record of an ALU. The Directory File Record may be divided into component blocks of data, each of which is individually transmitted using this command.

Availability

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Conditional Usage and Security

This command is available after enablement and must follow the command OPEN MEL APPLICATION command.

APDU Command

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>20</td>
<td>MSB Starting Offset</td>
<td>LSB Starting Offset</td>
<td>var.</td>
<td>DIR File Record component</td>
<td>-</td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

APDU Response

Status word values that can be returned are:

9D 15 Application not open
9D 17 Invalid offset

Remarks

The Directory File Record components can be sent in any order. The parameter bytes P1 and P2 are used to indicate the address within the reserved Directory File Record memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

Standards

MULTOS
**LOAD FCI RECORD**

**Description**

This command is used to load the File Control Information Record of an ALU. The FCI Record may be divided into component blocks of data, each of which is individually transmitted using this command.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is available after enablement and must follow the OPEN MEL APPLICATION command.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>22</td>
<td>MSB</td>
<td>LSB</td>
<td>var.</td>
<td>FCI Record component</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Starting Offset</td>
<td>Starting Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

**APDU Response**

Status word values that can be returned are:

- 9D 15 Application not open
- 9D 17 Invalid offset

**Remarks**

The FCI Record components can be sent in any order. The parameter bytes P1 and P2 are used to indicate the address within the reserved FCI Record memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

**Standards**

MULTOS
**LOAD KTU CIPHERTEXT**

This command is used to load the Key Transformation Unit of an ALU. The KTU may be divided into component blocks of data, each of which is individually transmitted using this command.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is available after enablement and must follow the OPEN MEL APPLICATION command.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>2A</td>
<td>MSB Starting Offset</td>
<td>LSB Starting Offset</td>
<td>var.</td>
<td>KTU component</td>
<td>-</td>
</tr>
</tbody>
</table>

No data is returned in response to this command.

**APDU Response**

Status word values that can be returned are:

9D 15 Application not open
9D 17 Invalid offset

**Remarks**

The KTU components can be sent in any order. The parameter bytes P1 and P2 are used to indicate the address within the reserved KTU memory area to load the component of size Lc. P1 is the most significant byte, while P2 is the least significant byte of the address. Note that the addressing uses zero based counting.

**Standards**

MULTOS
**OPEN MEL APPLICATION**

This command reserves and initialises memory in order to load an application into the MULTOS device, checking that there is sufficient memory of each type for a successful load. If the load can proceed, the MULTOS device will return its certified public key.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data component</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>12</td>
<td>00</td>
<td>00</td>
<td>var.</td>
<td>Open command data component</td>
<td>00</td>
</tr>
</tbody>
</table>

The device’s unique certified public key is returned in response to this command.

**APDU Response**

Status word values that can be returned are:

- 9D 07 Incorrect session data size
- 9D 08 Incorrect DIR file record size
- 9D 09 Incorrect FCI record size
- 9D 10 Insufficient memory to load application
- 9D 11 Invalid Application ID
- 9D 12 Duplicate Application ID
- 9D 13 Application previously loaded
- 9D 14 Application history full
- 9D 1D MSM controls not set
- 9D 21 Invalid open command data length
- 9D 50 Invalid MCD Issuer Product Id
- 9D 51 Invalid MCD Issuer Id
- 9D 52 Invalid Set MSM Controls Data Date
- 9D 53 Invalid MCD number
Remarks

When the command is transmitted, any other currently open or partially loaded application is abandoned.

The \( Lc \) value is given as variable; however, there are only two permissible values. They are: \( 0x89 \), when no application code hash is present and \( 0x9D \), when an application code hash (calculated using the SHA-1 algorithm) is present.

The Open command data component consists of:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCD Issuer Product IDs</td>
<td>32</td>
</tr>
<tr>
<td>MCD Issuer ID</td>
<td>4</td>
</tr>
<tr>
<td>Set MSM Controls Dates</td>
<td>32</td>
</tr>
<tr>
<td>MCD Number</td>
<td>8</td>
</tr>
<tr>
<td>RFU</td>
<td>18</td>
</tr>
<tr>
<td>Application ID</td>
<td>17</td>
</tr>
<tr>
<td>Random Seed</td>
<td>8</td>
</tr>
<tr>
<td>File Mode Type</td>
<td>1</td>
</tr>
<tr>
<td>Code Size</td>
<td>2</td>
</tr>
<tr>
<td>Data Size</td>
<td>2</td>
</tr>
<tr>
<td>Session Data Size</td>
<td>2</td>
</tr>
<tr>
<td>Application Signature</td>
<td>2</td>
</tr>
<tr>
<td>Application Code Hash</td>
<td>1</td>
</tr>
<tr>
<td>KTU Length</td>
<td>2</td>
</tr>
<tr>
<td>DIR File Record Size</td>
<td>2</td>
</tr>
<tr>
<td>FCI Record Size</td>
<td>2</td>
</tr>
<tr>
<td>Access List</td>
<td>2</td>
</tr>
</tbody>
</table>

The fields Random Seed and Access List were introduced as of MULTOS 4. Neither field is present for a MULTOS 3 OPEN MEL APPLICATION command.
In order to open an application successfully:

- the MULTOS device must have been enabled
- there must have enough free memory space of each type to load the ALU
- no application with the same Application ID (AID) as an already loaded application may be loaded.
- If the random seed is non-zero, then the combination of Application ID and Random Seed must not have previously been loaded to this MULTOS device.

**Standards**

MULTOS


**READ BINARY**

**Description**

This command reads a block of bytes from a transparent MULTOS elementary file or from an area of application Static memory previously specified by an application.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅</td>
<td>✅</td>
<td>✅*</td>
<td>✅*</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

* reading from an area of application Static memory is an optional feature in MULTOS 4.3

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>00/0C</td>
<td>B0/B1</td>
<td>see below</td>
<td>see below</td>
<td>see below</td>
<td>see below</td>
<td>var.</td>
</tr>
</tbody>
</table>

If a MULTOS elementary file is being read then CLA must equal 0x00 and INS must equal 0xB0. In this case P1 and P2 contain the offset, the most significant bit of the offset must equal zero and no command data is supplied.

If an area of application Static memory is being read then CLA must equal 0x00 if secure messaging is not being used or 0x0C if secure messaging is being used. INS must equal 0xB0 if an offset of less than 32768 bytes is specified and in this case P1 and P2 contain the offset. If an offset of 32768 bytes or more is specified then INS must equal 0xB1 and the command data contains the offset in TLV format. If secure messaging is being used then the command data must contain the appropriate ciphertext and MAC.

**APDU Response**

If a MULTOS elementary file is being read then the response contains the file data.

Status word values that can be returned are:

62 81  Part of returned data may be corrupted
If an area of application Static memory is being read then the response contains the area of Static being read, possibly in TLV format. If secure messaging is being used then the response data is encrypted and a MAC is also included.

Status word values that can be returned depend upon the functionality of the application that enabled the accelerated READ BINARY command.

**Remarks**

The operating system provides one transparent elementary file: the ATR File. This command must be used to read the data in that file.

**Standards**

ISO7816 Part 4, 6.1 Read Binary Command
**READ RECORD(S)**

This command reads a record from the currently selected fixed length elementary file.

**Availability**

MULTOS 4  MULTOS 4.2  MULTOS 4.3.1  MULTOS 4.3.2  MULTOS 4.4  MULTOS 4.5

- ✔  ✔  ✔  ✔  ✔  ✔

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B2</td>
<td>Record Number</td>
<td>0x04</td>
<td>-</td>
<td>-</td>
<td>var.</td>
</tr>
</tbody>
</table>

Upon successful execution of this command a record is returned.

**APDU Response**

Status word values that can be returned are:

- 62 81 Part of returned data may be corrupted
- 62 82 End of file reached
- 67 00 Wrong length, empty Le field
- 69 81 Command incompatible with file structure
- 69 82 Security status not satisfied
- 6A 81 Function not supported
- 6A 82 File not found
- 6A 83 Record not found
- 6C xx Wrong length, xx indicates actual length

**Remarks**

The response message gives the contents of the specified record(s) of an EF with record structure.

The operating system provides one fixed length elementary file: the DIR File. This command must be used to read the data in that file and the P2 value must be 0x04.

**Standards**

MULTOS Specification
ISO7816 Part 4 Section 6.5
**SELECT FILE**

This command is used to select the Master File (MF), the Directory File (DIR), the ATR File or an application loaded into the MULTOS device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>A4</td>
<td>var.</td>
<td>var.</td>
<td>var.</td>
<td>var.</td>
<td>var.</td>
</tr>
</tbody>
</table>

See the remarks section for valid P1 P2 values.

Data may be returned in response to this command.

**APDU Response**

Status word values that can be returned are:

- 62 83  Selected File invalidated
- 62 84  FCI not formatted according to ISO 7816
- 6B 00  Wrong Parameters
- 6A 81  Function not supported
- 6A 82  File not found
- 6A 86  Incorrect parameters P1,P2
- 6A 87  Lc inconsistent with P1,P2
Remarks

The following table shows all acceptable P1, P2 and Lc values that can be for the command Select File. MULTOS will not handle a Select File command with any other combinations. However, an application may use any combination.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>LC</th>
<th>CMD DATA</th>
<th>SELECTS</th>
<th>IF FILE EXISTS, RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>none</td>
<td>none</td>
<td>Master File</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>02</td>
<td>3F 00</td>
<td>Master File</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>02</td>
<td>2F 00</td>
<td>Directory File*</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>02</td>
<td>2F 01</td>
<td>ATR File</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>00</td>
<td>0C</td>
<td>02</td>
<td>2F 00</td>
<td>Directory File*</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>00</td>
<td>0C</td>
<td>02</td>
<td>2F 01</td>
<td>ATR File</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>04</td>
<td>00</td>
<td>01 - 10</td>
<td>AID or partial AID</td>
<td>Application (DF)</td>
<td>Success and FCI</td>
</tr>
<tr>
<td>04</td>
<td>02</td>
<td>01 - 10</td>
<td>AID or partial AID</td>
<td>Application (DF)</td>
<td>Success and FCI</td>
</tr>
<tr>
<td>04</td>
<td>0C</td>
<td>01 - 10</td>
<td>AID or partial AID</td>
<td>Application (DF)</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>08</td>
<td>00</td>
<td>02</td>
<td>3F 00</td>
<td>Master File</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>08</td>
<td>00</td>
<td>02</td>
<td>2F 00</td>
<td>Directory File**</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>08</td>
<td>0C</td>
<td>02</td>
<td>3F 00</td>
<td>Master File</td>
<td>Success: 90 00</td>
</tr>
<tr>
<td>08</td>
<td>0C</td>
<td>02</td>
<td>2F 00</td>
<td>Directory File**</td>
<td>Success: 90 00</td>
</tr>
</tbody>
</table>

The Lc listing for all the cases where P1 is 0x04 indicates that the Lc must have a value between 0x01 and 0x10.

The Le is given in APDU Command as variable. FCI data is returned only if it is present for an application and in the cases where P1 and P2 are 0x04 0x00 or 0x04 0x02.

If MULTOS cannot successfully process the command, and an application is currently selected, MULTOS passes the command to the selected application to handle or reject as appropriate.

The application selection process will operate over all of the loaded applications and not just the first application that has an AID that (partially) matches the AID in the SELECT command. The command will reply with “file not found” only if there are no loaded applications that have an AID that (partially) matches and which are permitted over the selected interface.

If the MULTOS device is blocked then the command is not available and a status response of 62 83 is returned.

If the application has the “Process Events” permission, MULTOS does not test the most significant 6 bits of P2. The processing of the least significant 2 bits of P2 remain unchanged. For more details on Process Events please see [MDG].

NOTE*: Processed by the currently selected application, if there is one.
NOTE**: Always processed by MULTOS and any currently selected application is deselected.

The MULTOS AID is 0xA0000001444D554C544F53.
Standards

MULTOS
ISO 7816 Part 4
**SET MSM CONTROLS**

This command is used to transmit MSM Controls Data, also referred to as Enablement Data, to the target device.

**Availability**

<table>
<thead>
<tr>
<th>MULTOS 4</th>
<th>MULTOS 4.2</th>
<th>MULTOS 4.3.1</th>
<th>MULTOS 4.3.2</th>
<th>MULTOS 4.4</th>
<th>MULTOS 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Conditional Usage and Security**

This command is always available.

**APDU Command**

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>var.</td>
<td>MSM Controls Data component</td>
<td>-</td>
</tr>
</tbody>
</table>

The Lc field gives the length of MSM Controls Data component transmitted in the command.

**APDU Response**

Status word values that can be returned are:

- 9D 40 Invalid MSM controls ciphertext OR
- 9D 40 Already enabled with step/one data (MULTOS products supporting step/one loading)
- 9D 41 MSM controls set

No data is returned in response to this command.

**Remarks**

The MSM Controls Data may be transmitted over several APDUs. The same CLA INS P1 and P2 values are used for each command sent. Note that the data must be transmitted sequentially so that the MSM Controls Data can be properly reconstructed on the device.

MSM Controls Data is specific to an individual MULTOS device. The same data cannot be used on different devices.

**Standards**

MULTOS
## MULTOS Status Codes

The previous section listed the commands that are handled by MULTOS along with possible status word responses. A MULTOS specific status word always has the most significant byte set to 0x9D. The following table provides a comprehensive listing of all MULTOS specific status words and provides an explanation of each.

<table>
<thead>
<tr>
<th>SW2</th>
<th>Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>Incorrect certificate type</td>
<td>The MULTOS device was given an ADC when it expected an ALC or vice-versa.</td>
</tr>
<tr>
<td>07</td>
<td>Incorrect session data size</td>
<td>The session data size given in the ALC is larger than the maximum available on the device.</td>
</tr>
<tr>
<td>08</td>
<td>Incorrect DIR file record size</td>
<td>The DIR file record size given in the ALC is larger than the maximum available on the device.</td>
</tr>
<tr>
<td>09</td>
<td>Incorrect FCI record size</td>
<td>The FCI record size given in the ALC is larger than the maximum available on the device.</td>
</tr>
<tr>
<td>10</td>
<td>Insufficient memory to load application</td>
<td>The device is unable to allocate all the memory required to perform the application load.</td>
</tr>
<tr>
<td>11</td>
<td>Invalid application id</td>
<td>The Application ID length is either 0 or greater than 16 bytes in size.</td>
</tr>
<tr>
<td>12</td>
<td>Duplicate application id</td>
<td>An application with the same AID is currently loaded. The AID value must be unique on the device.</td>
</tr>
<tr>
<td>13</td>
<td>Application previously loaded</td>
<td>The combination of AID and non-zero random seed value held in the ALC is already listed in the application history list.</td>
</tr>
<tr>
<td>14</td>
<td>Application history full</td>
<td>An attempt has been made to load or delete an application using a non-zero random seed value when the application history list is full. Loads or deletes that do not use the random seed value will be handled normally.</td>
</tr>
<tr>
<td>15</td>
<td>Application not open</td>
<td>An attempt has been made to send a LOAD … or CREATE MEL APPLICATION command prior to the successful execution of the OPEN MEL APPLICATION command.</td>
</tr>
<tr>
<td>17</td>
<td>Invalid offset</td>
<td>Given in response to a LOAD … command when the combination of the P1 P2 specified offset and Lc value result in an attempt to load data outside the area reserved for the component in the OPEN MEL APPLICATION command.</td>
</tr>
<tr>
<td>18</td>
<td>Application already loaded</td>
<td>An attempt has been made to load a shell or default application onto a MULTOS device that already has applications loaded. Or the application is in fact already on the device.</td>
</tr>
<tr>
<td>19</td>
<td>Invalid certificate</td>
<td>The ALC or ADC received by the MULTOS device was not successfully authenticated.</td>
</tr>
<tr>
<td>1A</td>
<td>Invalid signature</td>
<td>The Application Signature provided was not successfully verified.</td>
</tr>
<tr>
<td>1B</td>
<td>Invalid key transformation unit</td>
<td>The deciphered KTU contains data that does not match the corresponding values on the device. The KTU for the MULTOS device is invalid.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Error Message</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>1D</td>
<td>MSM controls not set</td>
<td>The MULTOS device has not yet been enabled.</td>
</tr>
<tr>
<td>1E</td>
<td>Application Signature does not exist</td>
<td>The MULTOS device did not receive an Application Signature when it was expecting one.</td>
</tr>
<tr>
<td>1F</td>
<td>KTU does not exist</td>
<td>The MULTOS device did not receive a KTU when it was expecting one.</td>
</tr>
<tr>
<td>20</td>
<td>Application not loaded</td>
<td>An attempt has been made to delete an application from a MULTOS device that doesn’t exist.</td>
</tr>
<tr>
<td>30</td>
<td>Check data parameter is incorrect (invalid start address)</td>
<td>The start address given is not found in the memory area to be checked.</td>
</tr>
<tr>
<td>31</td>
<td>Check data parameter is incorrect (invalid length)</td>
<td>The combination of start address and length is not found in the memory area to be checked.</td>
</tr>
<tr>
<td>32</td>
<td>Check data parameter is incorrect (illegal memory check area)</td>
<td>The address and length given represent a valid memory area, but it is not permitted to perform a check data over it.</td>
</tr>
<tr>
<td>40</td>
<td>Invalid MSM controls ciphertext</td>
<td>The format or unverified content of the MSM controls data.</td>
</tr>
<tr>
<td>41</td>
<td>MSM controls set</td>
<td>An attempt has been made to enable a device that has already been enabled.</td>
</tr>
<tr>
<td>42</td>
<td>Set MSM Controls data length less than 2 bytes</td>
<td>The very first command sent includes the total length of data to be sent. If the first command does not contain at least two bytes of data, this error is given.</td>
</tr>
<tr>
<td>43</td>
<td>Invalid msm controls data length</td>
<td>The data is less than an expected minimum, greater than an expected maximum or its size is not an integer multiple of 8.</td>
</tr>
<tr>
<td>44</td>
<td>Excess msm controls ciphertext</td>
<td>More data has been sent to the device than was expected.</td>
</tr>
<tr>
<td>45</td>
<td>Verification of msm controls data failed</td>
<td>The MSM controls data was not successfully verified.</td>
</tr>
<tr>
<td>50</td>
<td>Invalid mcd issuer product id</td>
<td>The bit mapped product IDs in the ALC or ADC does not contain the MULTOS device’s product ID.</td>
</tr>
<tr>
<td>51</td>
<td>Invalid mcd issuer id</td>
<td>The MULTOS device issuer ID does not match that given in the ALC or ADC.</td>
</tr>
<tr>
<td>52</td>
<td>Invalid set msm controls data date</td>
<td>The bit mapped MSM Controls Data Dates in the ALC or ADC does not contain the MULTOS device’s date.</td>
</tr>
<tr>
<td>53</td>
<td>Invalid mcd number</td>
<td>The MULTOS device MCD Number does not match that given in the ALC or ADC.</td>
</tr>
<tr>
<td>54</td>
<td>Reserved field error</td>
<td>The reserved field in the ALC does not match the value given in the OPEN MEL APPLICATION command.</td>
</tr>
<tr>
<td>55</td>
<td>Reserved field error</td>
<td>The reserved field in the ALC does not match the value given in the OPEN MEL APPLICATION command.</td>
</tr>
<tr>
<td>56</td>
<td>Reserved field error</td>
<td>The reserved field in the ALC does not match the value given in the OPEN MEL APPLICATION command.</td>
</tr>
<tr>
<td>57</td>
<td>Reserved field error</td>
<td>The reserved field in the ALC does not match the value given in the OPEN MEL APPLICATION command.</td>
</tr>
<tr>
<td>60</td>
<td>MAC verification failed</td>
<td>Card block or unblock MAC did not verify.</td>
</tr>
<tr>
<td>61</td>
<td>Maximum number of unblocks reached</td>
<td>The device will not process this or any further Card Unblock commands because the limit has been reached.</td>
</tr>
<tr>
<td>62</td>
<td>This is not a blocked device</td>
<td>An attempt to unblock a device that is not blocked has been made.</td>
</tr>
</tbody>
</table>
### Instruction Map

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Tag</th>
<th>Instruction</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>SYSTEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No Operation</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>SW1 SW2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>set SW1 and SW2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>La</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Set La</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>SW1 SW2 La</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Set SW1, SW2 and La</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>La</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Exit</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>SW1 SW2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>set SW1, SW2 and exit</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>La</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Set La and exit</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>SW1 SW2 La</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Set SW1, SW2, La and exit</td>
</tr>
<tr>
<td>0x01</td>
<td>0</td>
<td>BRANCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP if Equal</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP if Less Than</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP if Less/Equal</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP if Greater Than</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP if Greater/Equal</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP if Not Equal</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>RelCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch RelCP</td>
</tr>
<tr>
<td>0x02</td>
<td>0</td>
<td>JUMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP on stack</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP if Equal</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP if Less Than</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP if Less/Equal</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP if Greater Than</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP if Greater/Equal</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP if Not Equal</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump to CP</td>
</tr>
<tr>
<td>0x03</td>
<td>0</td>
<td>CALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP on stack</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP if Equal</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP if Less Than</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP if Less/Equal</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP if Greater Than</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP if Greater/Equal</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP if Not Equal</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call CP</td>
</tr>
<tr>
<td>0x04</td>
<td>0</td>
<td>STACK</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes of zeroes onto the stack.</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>push byte b onto the stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>Push word w onto the stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Illegal Instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>Pop n bytes from the stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Pop one byte from the stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Pop one word from the stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Illegal Instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x05</td>
<td>0</td>
<td>PRIMRET</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td>Call Primitive p</td>
</tr>
<tr>
<td>1</td>
<td>p a</td>
<td>Call Primitive p with arg a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>p a b</td>
<td>Call Primitive p with args a,b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>p a b c</td>
<td>Call Primitive p with args a,b,c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Return from Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>in</td>
<td>Return from Call discarding in bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>out</td>
<td>Return from Call with out bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>in out</td>
<td>Return from Call discarding in bytes and returning out bytes.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td>0</td>
<td>unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Illegal instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpCode</td>
<td>Tag</td>
<td>Instruction</td>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
<td>Notes</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-------</td>
</tr>
<tr>
<td>0x07</td>
<td>0</td>
<td>LOAD</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>Duplicates top n bytes of stack</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from SB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from ST[0ffset]</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from DB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from LB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from DT[0ffset]</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from PB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Push n bytes from PT[0ffset]</td>
</tr>
<tr>
<td>0x08</td>
<td>0</td>
<td>STORE</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to DT[-2*n]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to SB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to ST[0ffset]</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to DB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to LB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to DT[0ffset]</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to PB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes and store them to PT[0ffset]</td>
</tr>
<tr>
<td>0x09</td>
<td>0</td>
<td>LOADI</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at DT[-2] onto the stack</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at SB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at ST[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at DB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at LB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at DT[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at PB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load n bytes from the segment address at PT[0ffset] onto the stack</td>
</tr>
<tr>
<td>0x0A</td>
<td>0</td>
<td>STOREI</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in DT[-2*n]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in SB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in ST[0ffset]</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in DB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in LB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in DT[0ffset]</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in PB[0ffset]</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pop n bytes from the stack and store them to the segment address in PT[0ffset]</td>
</tr>
<tr>
<td>0x0B</td>
<td>0</td>
<td>LOADA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Illegal instruction</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of SB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of ST[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of DB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of LB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of DT[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of PB[0ffset] onto the stack</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load the segment address of PT[0ffset] onto the stack</td>
</tr>
<tr>
<td>0x0C</td>
<td>0</td>
<td>INDEX</td>
<td>b</td>
<td>offset</td>
<td></td>
<td></td>
<td>Illegal instruction</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>b offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Multiplies the byte at the top of the stack with b, adds the segment address of a record of SB[0ffset] to the product and loads the result onto the stack</td>
</tr>
<tr>
<td>OpCode</td>
<td>Tag</td>
<td>Instruction</td>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-------</td>
</tr>
<tr>
<td>0x00</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at the top of the stack to b</td>
</tr>
<tr>
<td>0x02</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at SB[offset] to b</td>
</tr>
<tr>
<td>0x03</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at ST[offset] to b</td>
</tr>
<tr>
<td>0x04</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at DB[offset] to b</td>
</tr>
<tr>
<td>0x05</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at LB[offset] to b</td>
</tr>
<tr>
<td>0x06</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at DT[offset] to b</td>
</tr>
<tr>
<td>0x07</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at PB[offset] to b</td>
</tr>
<tr>
<td>0x08</td>
<td>0</td>
<td>SETB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Sets the byte at PT[offset] to b</td>
</tr>
<tr>
<td>0x0D</td>
<td>0</td>
<td>CMPB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0E</td>
<td>0</td>
<td>CMPB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at the top of the stack with b</td>
</tr>
<tr>
<td>0x0F</td>
<td>0</td>
<td>CMPB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at SB[offset] with b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at ST[offset] with b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at DB[offset] with b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at LB[offset] with b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at DT[offset] with b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at PB[offset] with b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares byte at PT[offset] with b</td>
</tr>
<tr>
<td>0x10</td>
<td>0</td>
<td>SUBB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x11</td>
<td>0</td>
<td>SUBB</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte at the top of the stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at SB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at ST[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at DB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at LB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at DT[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at PB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts b from the byte stored at PT[offset]</td>
</tr>
<tr>
<td>0x12</td>
<td>0</td>
<td>CMPW</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the word at the top of the stack with w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at SB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at ST[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at DB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at LB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at DT[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at PB[offset]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at PT[offset]</td>
</tr>
<tr>
<td>OpCode</td>
<td>Tag</td>
<td>Instruction</td>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-------</td>
</tr>
<tr>
<td>4</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares w with the word stored at PT[offset]</td>
</tr>
<tr>
<td>0x13</td>
<td>ADDW</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at the top of the stack.</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds w to the word at PT[offset]</td>
</tr>
<tr>
<td>0x14</td>
<td>SUBW</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at the top of the stack.</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at BB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>w</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts w from the word at PT[offset]</td>
</tr>
<tr>
<td>0x15</td>
<td>CLEARN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears the top n bytes of the stack</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clears n bytes starting at PT[offset]</td>
</tr>
<tr>
<td>0x16</td>
<td>TESTN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes of the stack with 0</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at SB[offset] with 0</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at ST[offset] with 0</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at DB[offset] with 0</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at LB[offset] with 0</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at DT[offset] with 0</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at PB[offset] with 0</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares the top n bytes at PT[offset] with 0</td>
</tr>
<tr>
<td>0x17</td>
<td>INCN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at DT[-n] of length n bytes</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at SB[offset] of length n bytes.</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at ST[offset] of length n bytes.</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at DB[offset] of length n bytes.</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at LB[offset] of length n bytes.</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at DT[offset] of length n bytes.</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at PB[offset] of length n bytes.</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Increments the number at PT[offset] of length n bytes.</td>
</tr>
<tr>
<td>0x18</td>
<td>DECN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at DT[-n] of length n bytes</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at SB[offset] of length n bytes.</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at ST[offset] of length n bytes.</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at DB[offset] of length n bytes.</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at LB[offset] of length n bytes.</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at DT[offset] of length n bytes.</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at PB[offset] of length n bytes.</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decrements the number at PT[offset] of length n bytes.</td>
</tr>
<tr>
<td>0x19</td>
<td>NOTN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts the top n bytes of the stack.</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inverts n bytes stored at PT[offset]</td>
</tr>
<tr>
<td>OpCode</td>
<td>Tag</td>
<td>Instruction</td>
<td>b1 w1</td>
<td>b2 w2</td>
<td>b3 w3</td>
<td>b4</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>0x1A 0</td>
<td>CMPN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at DT[-2*n]</td>
</tr>
<tr>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compares top n bytes of stack with the bytes stored at PT[offset]</td>
</tr>
<tr>
<td>0x1B 0</td>
<td>ADDN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack with bytes stored at DT[-2*n]</td>
</tr>
<tr>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adds top n bytes of stack to bytes stored at PT[offset]</td>
</tr>
<tr>
<td>0x1C 0</td>
<td>SUBN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack with bytes stored at DT[-2*n]</td>
</tr>
<tr>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtracts top n bytes of stack from bytes stored at PT[offset]</td>
</tr>
<tr>
<td>0x1D 0</td>
<td>ANDN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at DT[-2*n]</td>
</tr>
<tr>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ANDs top n bytes of stack with bytes stored at PT[offset]</td>
</tr>
<tr>
<td>0x1E 0</td>
<td>ORN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically Ors top n bytes of stack with bytes stored at DT[-2*n]</td>
</tr>
<tr>
<td>1</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically Ors top n bytes of stack with bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically Ors top n bytes of stack with bytes stored at ST[offset]</td>
</tr>
<tr>
<td>OpCode</td>
<td>Tag</td>
<td>Instruction</td>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-------</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ORs top n bytes of stack with bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ORs top n bytes of stack with bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ORs top n bytes of stack with bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ORs top n bytes of stack with bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically ORs top n bytes of stack with bytes stored at PT[offset]</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
<td>XORN</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top 2n bytes of stack with bytes stored at DT[2n]</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at SB[offset]</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at ST[offset]</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at DB[offset]</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at LB[offset]</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at DT[offset]</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at PB[offset]</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logically XORs top n bytes of stack with bytes stored at PT[offset]</td>
</tr>
</tbody>
</table>
Primitive Set Listing

MULTOS Primitives are divided into four sets: 0, 1, 2 and 3. These are based on the number of non-stack argument bytes that are passed to a primitive. Please note that any number of values may be placed on the stack for the primitive without it impacting on the Primitive Set designation. Within each set a hex value is assigned to the primitive. This is used to uniquely identify that primitive within the set.

Set 0

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Case</td>
<td>0x01</td>
</tr>
<tr>
<td>Reset WWVT</td>
<td>0x02</td>
</tr>
<tr>
<td>Get Session Size</td>
<td>0x03</td>
</tr>
<tr>
<td>Update Session Size</td>
<td>0x04</td>
</tr>
<tr>
<td>Load CCR</td>
<td>0x05</td>
</tr>
<tr>
<td>Store CCR</td>
<td>0x06</td>
</tr>
<tr>
<td>Set ATR File Record</td>
<td>0x07</td>
</tr>
<tr>
<td>Set ATR Historical Characters</td>
<td>0x08</td>
</tr>
<tr>
<td>Get Memory Reliability</td>
<td>0x09</td>
</tr>
<tr>
<td>Lookup</td>
<td>0x0A</td>
</tr>
<tr>
<td>Memory Compare</td>
<td>0x0B</td>
</tr>
<tr>
<td>Memory Copy</td>
<td>0x0C</td>
</tr>
<tr>
<td>Query Interface Type</td>
<td>0x0D</td>
</tr>
<tr>
<td>Set ATS Historical Characters</td>
<td>0x0E</td>
</tr>
<tr>
<td>Memory Copy Non-Atomic</td>
<td>0x0F</td>
</tr>
<tr>
<td>Control Auto Reset WWVT</td>
<td>0x10</td>
</tr>
<tr>
<td>Set FCI File Record</td>
<td>0x11</td>
</tr>
<tr>
<td>Set API</td>
<td>0x12</td>
</tr>
<tr>
<td>Card Unblock</td>
<td>0x13</td>
</tr>
<tr>
<td>Lookup Word</td>
<td>0x14</td>
</tr>
<tr>
<td>Get Configuration Data</td>
<td>0x15</td>
</tr>
<tr>
<td>Get Transaction State</td>
<td>0x16</td>
</tr>
<tr>
<td>Exit to MULTOS and Restart</td>
<td>0x17</td>
</tr>
<tr>
<td>Delegate</td>
<td>0x80</td>
</tr>
<tr>
<td>Reset Session Data</td>
<td>0x81</td>
</tr>
<tr>
<td>Checksum</td>
<td>0x82</td>
</tr>
<tr>
<td>Call Codelet</td>
<td>0x83</td>
</tr>
<tr>
<td>Query Codelet</td>
<td>0x84</td>
</tr>
<tr>
<td>Exchange Data</td>
<td>0x85</td>
</tr>
<tr>
<td>Query Channel</td>
<td>0x86</td>
</tr>
<tr>
<td>Get FCI State</td>
<td>0x87</td>
</tr>
<tr>
<td>Query Algorithm</td>
<td>0x8A</td>
</tr>
<tr>
<td>Platform Optimised Checksum</td>
<td>0x89</td>
</tr>
<tr>
<td>DES ECB Encipher</td>
<td>0xC1</td>
</tr>
<tr>
<td>Modular Multiplication</td>
<td>0xC2</td>
</tr>
<tr>
<td>Modular Reduction</td>
<td>0xC3</td>
</tr>
<tr>
<td>Get Random Number</td>
<td>0xC4</td>
</tr>
<tr>
<td>DES ECB Decipher</td>
<td>0xC5</td>
</tr>
<tr>
<td>Generate DES CBC Signature</td>
<td>0xC6</td>
</tr>
<tr>
<td>Generate Triple DES CBC Signature</td>
<td>0xC7</td>
</tr>
<tr>
<td>Modular Exponentiation / RSA Sign</td>
<td>0xC8</td>
</tr>
<tr>
<td>Modular Exponentiation CRT / RSA Sign CRT</td>
<td>0xC9</td>
</tr>
<tr>
<td>SHA-T</td>
<td>0xCA</td>
</tr>
<tr>
<td>GSM Authenticate</td>
<td>0xCB</td>
</tr>
<tr>
<td>Generate Random Prime</td>
<td>0xCC</td>
</tr>
<tr>
<td>SEED ECB Decipher</td>
<td>0xCD</td>
</tr>
<tr>
<td>SEED ECB Encipher</td>
<td>0xCE</td>
</tr>
<tr>
<td>Secure Hash</td>
<td>0xCF</td>
</tr>
<tr>
<td>ECC Addition</td>
<td>0xD0</td>
</tr>
<tr>
<td>ECC Convert Representation</td>
<td>0xD1</td>
</tr>
<tr>
<td>ECC Equality Test</td>
<td>0xD2</td>
</tr>
<tr>
<td>ECC Inverse</td>
<td>0xD3</td>
</tr>
<tr>
<td>ECC Scalar Multiplication</td>
<td>0xD4</td>
</tr>
<tr>
<td>AES Decipher</td>
<td>0xD6</td>
</tr>
<tr>
<td>AES Encipher</td>
<td>0xD7</td>
</tr>
<tr>
<td>Triple DES ECB Decipher</td>
<td>0xD8</td>
</tr>
<tr>
<td>Triple DES ECB Encipher</td>
<td>0xD9</td>
</tr>
<tr>
<td>Primitive Name</td>
<td>Hex Value</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Check BCD</td>
<td>0x0A</td>
</tr>
<tr>
<td>Get Replaced Application State</td>
<td>0x0B</td>
</tr>
<tr>
<td>Modular Exponentiation CRT Protected / RSA Sign CRT Protected</td>
<td>0x0C</td>
</tr>
<tr>
<td>Get AID</td>
<td>0x0D</td>
</tr>
<tr>
<td>Secure Hash IV</td>
<td>0x0E</td>
</tr>
<tr>
<td>Initialise PIN</td>
<td>0x0F</td>
</tr>
<tr>
<td>Read PIN</td>
<td>0x10</td>
</tr>
<tr>
<td>Verify PIN</td>
<td>0x11</td>
</tr>
<tr>
<td>Get Process Event</td>
<td>0x12</td>
</tr>
<tr>
<td>Reject Process Event</td>
<td>0x13</td>
</tr>
<tr>
<td>RSA Verify</td>
<td>0x14</td>
</tr>
<tr>
<td>Flush Public</td>
<td>0x15</td>
</tr>
</tbody>
</table>

**Set 1**

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query0</td>
<td>0x00</td>
</tr>
<tr>
<td>Query1</td>
<td>0x01</td>
</tr>
<tr>
<td>Query2</td>
<td>0x02</td>
</tr>
<tr>
<td>Query3</td>
<td>0x03</td>
</tr>
<tr>
<td>Set Static Size</td>
<td>0x04</td>
</tr>
<tr>
<td>Memory Compared Enhanced</td>
<td>0x05</td>
</tr>
<tr>
<td>Memory Copy From Replaced Application</td>
<td>0x06</td>
</tr>
<tr>
<td>Manage Stack</td>
<td>0x07</td>
</tr>
<tr>
<td>DivideN</td>
<td>0x08</td>
</tr>
<tr>
<td>Get DIR File Record</td>
<td>0x09</td>
</tr>
<tr>
<td>Get File Control Information</td>
<td>0x0A</td>
</tr>
<tr>
<td>Get Manufacturer Data</td>
<td>0x0B</td>
</tr>
<tr>
<td>Get MULTOS Data</td>
<td>0x0C</td>
</tr>
<tr>
<td>Get Purse Type</td>
<td>0x0D</td>
</tr>
<tr>
<td>Memory Copy Fixed Length</td>
<td>0x0E</td>
</tr>
<tr>
<td>Memory Compare Fixed Length</td>
<td>0x0F</td>
</tr>
<tr>
<td>MultiplyN</td>
<td>0x10</td>
</tr>
<tr>
<td>Add BCDN</td>
<td>0x11</td>
</tr>
<tr>
<td>Subtract BCDN</td>
<td>0x12</td>
</tr>
<tr>
<td>Memory Copy Non-Atomic Fixed Length</td>
<td>0x13</td>
</tr>
<tr>
<td>Convert BCD</td>
<td>0x14</td>
</tr>
<tr>
<td>Pad</td>
<td>0x15</td>
</tr>
<tr>
<td>Unpad</td>
<td>0x16</td>
</tr>
<tr>
<td>Get Available Interface Types</td>
<td>0x17</td>
</tr>
<tr>
<td>Set Transaction Protection</td>
<td>0x18</td>
</tr>
<tr>
<td>Get Delegator AID</td>
<td>0x19</td>
</tr>
<tr>
<td>Set PIN Data</td>
<td>0x1A</td>
</tr>
<tr>
<td>Get PIN Data</td>
<td>0x1B</td>
</tr>
<tr>
<td>Get Data</td>
<td>0x1C</td>
</tr>
<tr>
<td>Generate Asymmetric Hash General</td>
<td>0x1D</td>
</tr>
<tr>
<td>Generate MAC</td>
<td>0x1E</td>
</tr>
<tr>
<td>Modular Inverse</td>
<td>0x1F</td>
</tr>
<tr>
<td>ECC Verify</td>
<td>0x20</td>
</tr>
<tr>
<td>Configure Read Binary</td>
<td>0x21</td>
</tr>
<tr>
<td>Memory Copy Additional Static</td>
<td>0x22</td>
</tr>
<tr>
<td>Memory Fill Additional Static</td>
<td>0x23</td>
</tr>
<tr>
<td>Get Static Size</td>
<td>0x24</td>
</tr>
<tr>
<td>Generate Asymmetric Signature General</td>
<td>0x25</td>
</tr>
<tr>
<td>Verify Asymmetric and Retrieve General</td>
<td>0x26</td>
</tr>
<tr>
<td>Set Silent Mode</td>
<td>0x27</td>
</tr>
<tr>
<td>Initialise PIN Extended</td>
<td>0x28</td>
</tr>
<tr>
<td>ECC Generate Signature</td>
<td>0x29</td>
</tr>
<tr>
<td>ECC Verify Signature</td>
<td>0x2A</td>
</tr>
<tr>
<td>ECC Generate Key Pair</td>
<td>0x2B</td>
</tr>
<tr>
<td>ECC Elliptic Curve Diffie Hellman</td>
<td>0x2C</td>
</tr>
<tr>
<td>ECC ECIES Decipher</td>
<td>0x2D</td>
</tr>
<tr>
<td>ECC ECIES Encipher</td>
<td>0x2E</td>
</tr>
</tbody>
</table>

**Set 2**

© 2019 MAOSCO Limited. MAO-DOC-TEC-006 v1.56
MULTOS is a registered trademark of MULTOS Limited.
### Set 3

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Manipulate Byte</td>
<td>0x01</td>
</tr>
<tr>
<td>Shift Left</td>
<td>0x02</td>
</tr>
<tr>
<td>Shift Right</td>
<td>0x03</td>
</tr>
<tr>
<td>SetSelectSW</td>
<td>0x04</td>
</tr>
<tr>
<td>CardBlock</td>
<td>0x05</td>
</tr>
<tr>
<td>SetContactlessSelectSW</td>
<td>0x06</td>
</tr>
<tr>
<td>Shift Rotate</td>
<td>0x07</td>
</tr>
<tr>
<td>Update Process Events</td>
<td>0x08</td>
</tr>
<tr>
<td>Return From Codelet</td>
<td>0x80</td>
</tr>
<tr>
<td>Block Decipher</td>
<td>0xDA</td>
</tr>
<tr>
<td>Block Encipher</td>
<td>0xDB</td>
</tr>
<tr>
<td>Generate RSA Key Pair</td>
<td>0xE0</td>
</tr>
</tbody>
</table>

#### Bit Manipulate Word

| Call Extension 0, 1, 2, 3, 4, 5, 6   | 0x8x, where x = extension value [0, 6] |

0x86: The compiler uses a printf library and this library uses this primitive to perform the actual printf operation. The printf library calls the primitive with the three primitive parameters set to zero and it also pushes a value of zero onto the stack before calling the primitive. The simulator executes this primitive, printing the printf message onto the output window.

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