

**Application Programmers
Reference Manual**



Published by:
MAOSCO Limited
47-53 Cannon Street, London, EC4M 5SQ
Tel: +44 171 557 5000

Version 1.0 August 1997

© Copyright Mondex International Limited 1997

This document is confidential. 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 Mondex International Ltd.

This document is made available under the terms of the Application Licence Agreement signed with MAOSCO Limited and must not be disclosed to any other person or organisation otherwise than as set out in the terms of that Agreement.

This is a reference document that provides detailed information on the **MULTOS** application programming interfaces and the **MULTOS** Executable Language (MEL). **MULTOS** is a multi-application operating system that runs on a variety of smartcard hardware platforms from different manufacturers.

This manual is intended for experienced programmers. Some of the material assumes a general understanding of smartcard programming. The ideal reader should already be familiar with the ISO 7816 standards for smartcards, in particular Parts 3 and 4, which describe the communications protocols. Readers not already familiar with these standards should at least obtain copies for reference while reading this document.

Programmers without a background in smartcard programming, but with experience in assembly language, and those who are familiar with microprocessor architecture and related concepts such as addressing modes, register sets, instruction sets and program flow control, should find themselves in relatively familiar territory. Readers whose only experience is in higher-level languages like BASIC and C may be less comfortable with the material, which is presented at the abstract machine level.

This manual consists of this preface, a glossary of special terms, five main sections, and three appendices:

- **Section 1, Application Abstract Machine (AAM)**, describes the general architecture of the **MULTOS** platform. The AAM is an abstract, hardware-independent target platform that enables developers to write applications that are portable across all **MULTOS** implementations without knowing anything about the actual underlying hardware.
- **Section 2, IFD-Application Interface**, describes how an application receives commands from an Interface Device (IFD), and how it returns responses. This section requires some familiarity with ISO 7816, Parts 3 and 4, for complete understanding.
- **Section 3, Instruction Set**, gives a detailed description of each of the 31 instructions of the **MULTOS** application programming interface.

- **Section 4, Mandatory Primitives**, describes the primitives available to applications on all **MULTOS** implementations. These primitives are library functions that extend the instruction set.
- **Section 5, Optional Primitives**, describes the implementation-dependent primitives that may be available to applications. An application can query the specific platform on which it is running to know if a particular primitive is available. Many of the optional primitives provide cryptography functionality. While the general reader can understand the mechanism for using them, knowledge of cryptography is, of course, required to know exactly how, when, and why.
- **Appendix A** is a complete **instruction map**.
- **Appendix B** is a **list of primitives**.
- **Appendix C** lists **implementation-dependent features**.

This document is one of a set of **MULTOS** Design Standards. Other documents in the set, particularly the **MULTOS** Design Standard, IFD-**MULTOS** Interface Specification, may be of interest to all readers and may be required for a given application development project.

Although this manual provides complete information on the **MULTOS** application programming interface, it is not a tutorial and does not give specific examples of MEL programming beyond what is necessary to clarify certain instructions or primitives. Every effort has been made to make the presentation of the material logical and clear. As with all reference works, most readers will benefit from several readings followed by frequent reference while using the material.

The following table identifies notational conventions, defines terms that are particular to the **MULTOS** environment, and spells out acronyms that are used throughout this document. The glossary appears at the beginning because many, if not most, of the entries are certain to be new to most readers. Acronyms are spelled out again when they first occur in the text and most of the terms are explained in more detail in the context in which they appear.

Term	Description
<i>0x</i>	<i>This prefix indicates a hexadecimal number.</i>
<i>AAM</i>	<i>Application Abstract Machine Applications run on an abstract, or virtual, platform that is implemented in software. They need not know the implementation details of the actual underlying hardware.</i>
<i>abend</i>	<i>Abnormal End An application abends if it attempts any forbidden operation, such as the execution of an undefined instruction.</i>
<i>APDU</i>	<i>Application Protocol Data Unit This is the application-level message format defined in ISO 7816, Part 4. The message format seen by a MULTOS application is logically equivalent to an APDU, although the layout in memory is different.</i>
<i>API</i>	<i>Application Programming Interface The resources supplied to an application by an operating system, in this case, MULTOS.</i>
<i>ATR</i>	<i>Answer to Reset A message transmitted by a MULTOS Carrier Device (MCD) when an electrical reset signal is applied to it. Part of the ATR, the “historical bytes” or “historical characters” can be set by an application.</i>

<i>ATR File</i>	<i>A file on an MCD in to which applications can write data; this data is made available to an IFD on request. From a technical point of view, the ATR file has nothing to do with the ATR; however, it may contain information related to that in the ATR.</i>
<i>byte</i>	<i>A storage unit of eight bits. MULTOS documentation refers to the bits within a byte from 0 (least significant) to 7 (most significant). This is different from the convention used in ISO 7816, which refers to them from b1 (least significant) to b8 (most significant).</i>
<i>code address</i>	<i>The address of an instruction in the code space expressed as the offset, in bytes, of the start of the instruction relative to the start of the code space.</i>
<i>codelet</i>	<i>A series of MEL instructions that do not form part of an application but can be executed by an application using the Call Codelet primitive. Some implementations of MULTOS support codelets.</i>
<i>commit</i>	<i>The process of applying several writes in an atomic fashion, even though these writes were requested by separate instructions.</i>
<i>Delegation</i>	<i>The process by which one application (the delegator) sends a command to another application (the delegate). The delegate processes the command and returns a response to the delegator.</i>
<i>DIR File</i>	<i>Directory File A file on an MCD that contains a record for each application loaded on the MCD. These records may be read by an IFD or by any application. No application may write to them.</i>
<i>Dynamic</i>	<i>That part of an application's address space that holds private, volatile data. Dynamic starts at the tagged address DB[0] and extends to the tagged address DT[-1]. It comprises session data and the stack.</i>

<i>File Control Information</i>	<i>Data held on the MCD and related to a particular application. This data may be read by an IFD or by any application. No application may write to it.</i>
<i>IFD</i>	<i>Interface Device</i> <i>An Interface Device is a terminal that is communicating with an MCD.</i>
<i>literal</i>	<i>A byte or word of data whose actual or literal value is provided by an instruction. In the instruction: "Push the byte 101 on to the stack", 101 is a byte of literal data. In the instruction: "Store 3 bytes to address 999", the numbers 3 and 999 are not regarded as literals.</i>
<i>MCD</i>	MULTOS <i>Carrier Device</i> <i>A MULTOS Carrier Device is the hardware platform on which MULTOS resides.</i>
<i>MEL</i>	MULTOS <i>Executable Language</i> <i>MEL is the (abstract) machine language of the AAM.</i>
MULTOS	<i>Multi-Application Operating System</i> MULTOS <i>is the operating system software that implements the application abstract machine on an MCD (hardware) platform and provides all of the application services described in this manual.</i>
<i>Public</i>	<i>That part of an application's address space that holds data that is shared with the IFD or with another application. Public starts at the tagged address PB[0] and extends to the tagged address PT[-1]. Applications may also use Public to hold private data but are themselves responsible for deleting this data.</i>
<i>segment address</i>	<i>One of two methods by which an application may refer to an address within its data space. A segment address is a word (sixteen bit) number. Any given byte in an application's data space has one unique segment address. (See also tagged address)</i>

<i>session data</i>	<i>That part of Dynamic whose size is fixed when the application is loaded. Session data starts at the tagged address DB[0] and extends to the tagged address DB[sizeSessionData-1]. Session data is volatile. Its duration is a session (the time between two successive application selects).</i>
<i>Shell</i>	<i>A special mode of operation in which MULTOS redirects all application-level input commands to a defined shell application. System-level commands that require special access privileges are not redirected.</i>
<i>stack</i>	<i>That part of Dynamic that may expand and contract as the application executes. The stack starts at the tagged address DB[sizeSessionData] and extends to the tagged address DT[-1]. In practice, register DT is the stack pointer, DT[-1] represents the top of the stack, and the stack grows upward. The stack is volatile. Its duration is one command-response pair.</i>
<i>Static</i>	<i>That part of an application's address space that holds private, non-volatile data. Static starts at the tagged address SB[0] and extends to the tagged address SB[sizeStatic-1] or, synonymously, ST[-1].</i>
<i>tagged address</i>	<p><i>One of two methods by which an application may refer to an address within its data space. A tagged address is a word (sixteen bit) offset from an address register. Since there are seven address registers, any given byte may in principle be referred to by seven different tagged addresses, although in practice the only tagged addresses which should be used are:</i></p> <ul style="list-style-type: none"><i>• for data in Static, a zero or positive offset from SB or a negative offset from ST</i><i>• for data in Dynamic, a zero or positive offset from DB, a negative offset from DT, or an offset from LB</i>

- for data in *Public*, a zero or positive offset from *PB* or a negative offset from *PT*. (See also segment address)

<i>TPDU</i>	<p><i>Transport Protocol Data Unit</i></p> <p><i>This is the low-level message format for communication between the IFD and the MCD. The protocols T=0 and T=1, as defined in ISO 7816, Part 3, each consist of a set of TPDU formats. ISO 7816, Part 4 defines mapping between TPDU and APDUs. MULTOS performs APDU / TPDU translation.</i></p>
<i>transaction protection</i>	<p><i>A mechanism by which several successive writes to non-volatile data, made by one application, are committed in an atomic fashion.</i></p>
<i>word</i>	<p><i>A unit or storage of two bytes (16 bits). The most significant byte is stored at the lowest address.</i></p>
<i>write</i>	<p><i>An update to non-volatile storage.</i></p>

Preface
 Glossary

1

1.0 Application Abstract Machine 1
 1.1 Architectural Overview 1
 1.2 Address Space 1
 1.2.1 Code 1
 1.2.2 Data 1
 1.3 Registers 1
 1.3.1 Address Registers 1
 1.3.2 Control Registers 1
 1.3.3 Initial Register Values 1
 1.4 Instructions 1
 1.4.1 Types of Instruction 1
 1.4.2 Special characteristics of block operators 1
 1.4.3 Instruction Functions 1
 1.4.4 Machine-Language Coding 1
 1.5 Abending 1
 1.6 Functions 1
 1.7 Primitives 1
 1.7.1 Arithmetic, bitwise and block operation primitives 1
 1.7.2 Cryptographic primitives 1
 1.7.3 Operating system access primitives 1

2

2.0 IFD - Application Interface 1
 2.1 Using Public as the communications buffer 1
 2.2 Application entry 1
 2.3 Application exit 1
 2.4 Delegation 1
 2.4.1 Command rerouting 1
 2.4.2 Obtaining data from another application 1

3

3.0	Instruction Set	1
3.1	Introduction	1
3.2	ADDDB (ADD literal to Byte)	1
3.3	ADDN (ADD N-byte blocks)	1
3.4	ADDW (ADD literal to Word)	1
3.5	ANDN (bitwise AND N-byte blocks)	1
3.6	BRANCH (BRANCH to relative code address)	1
3.7	CALL (CALL function)	1
3.8	CLEARN (CLEAR N-byte block to 0)	1
3.9	CMPB (CoMPare literal with Byte)	1
3.10	CMPN (CoMPare N-byte blocks)	1
3.11	CMPW (CoMPare literal with Word)	1
3.12	DECN (DECrement N-byte block)	1
3.13	INCN (INCrement N-byte block)	1
3.14	INDEX (INDEX array)	1
3.15	JUMP (JUMP to code address)	1
3.16	LOAD (LOAD data to stack)	1
3.17	LOADA (LOAD Address)	1
3.18	LOADI (LOAD Indirect)	1
3.19	NOTN (bitwise NOT N-byte block)	1
3.20	ORN (bitwise OR N-byte blocks)	1
3.21	PRIMRET (call PRIMitive or RETurn)	1
3.22	SETB (SET Byte to literal)	1
3.23	SETW (SET Word to literal)	1
3.24	STACK (push or pop STACK)	1
3.25	STORE (STORE data from stack)	1
3.26	STOREI (STORE Indirect)	1
3.27	SUBB (SUBtract literal from Byte)	1
3.28	SUBN (SUBtract N-byte blocks)	1
3.29	SUBW (SUBtract literal from Word)	1
3.30	SYSTEM (various actions)	1
3.31	TESTN (TEST N-byte block against 0)	1
3.32	XORN (bitwise eXclusive OR N-byte blocks)	1

4

4.0	Mandatory Primitives	1
4.1	Introduction	1
4.2	Bit Manipulate Byte	1
4.3	Bit Manipulate Word	1
4.4	Check Case	1
4.5	DivideN	1
4.6	Get DIR File Record	1
4.7	Get File Control Information	1
4.8	Get Manufacturer Data	1
4.9	Get MULTOS Data	1
4.10	Get Memory Reliability	1
4.11	Get Purse Type	1
4.12	Load CCR	1
4.13	Lookup	1
4.14	Memory Compare	1
4.15	Memory Compare Fixed Length	1
4.16	Memory Copy	1
4.17	Memory Copy Fixed Length	1
4.18	MultiplyN 1	
4.19	Query0, Query1, Query2, Query3.....	1
4.20	Reset WWT	1
4.21	Set ATR File Record	1
4.22	Set ATR Historical Characters	1
4.23	Shift Left	1
4.24	Shift Right	1
4.25	Store CCR	1

5

5.0	Optional Primitives	1
5.1	Introduction	1
5.2	Call Codelet	1
5.3	Checksum	1
5.4	Delegate	1
5.5	DES ECB Decipher	1
5.6	DES ECB Encipher	1
5.7	Generate Asymmetric Hash	1
5.8	Generate DES CBC Signature	1
5.9	Generate Triple DES CBC Signature	1
5.10	Get Delegator AID	1
5.11	Get Random Number	1
5.12	Modular Exponentiation	1
5.13	Modular Exponentiation CRT	1
5.14	Modular Multiplication	1
5.15	Modular Reduction	1
5.16	Query Codelet	1
5.17	Reset Session Data	1
5.18	Return From Codelet	1
5.19	Set Transaction Protection	1
5.20	SHA-1	1

Appendix

Appendix A	Instruction Map	1
Appendix B	List of Primitives	1
Appendix C	Implementation-Dependent Features	1

1. Application Abstract Machine

Operating system or platform services are available to application programs in the form of an application abstract machine (AAM). This abstract, or virtual, machine enables applications to call upon and interact with a clearly defined set of platform services without any knowledge of the underlying hardware implementation, which may vary from manufacturer to manufacturer. Equally importantly, it ensures that an application correctly written for the abstract machine, as specified in this document, will run correctly on all hardware implementations that support the AAM.

There are two exceptions to this transparent portability across implementations. First, not all implementations support signed arithmetic, which is not typically needed in smartcard applications. Second, the optional primitives described in section 5 may not be available in all implementations. Applications that use optional primitives can ensure, using the mechanism provided, that a given primitive is available before calling it, so the programmer can decide what action to take if the primitive is not implemented. Both of these limitations are discussed in more detail in the appropriate sections.

1.1 Architectural Overview

Viewed from the highest level, the application abstract machine consists of:

- two separate address spaces, one for code and one for data

The data address space is divided into three *segments*.

- seven address registers and two control registers
- two addressing modes, *segment* and *tagged*
- 31 instructions

The core architecture is extended by:

- a set of mandatory primitives (library functions)
- a set of optional primitives (library functions)

Section 1

The following table gives a high-level overview of this architecture. Each of the architectural components is described in the following sections.

Item	Description
Address space	<p>Code space of up to 64K bytes. A given MULTOS implementation may impose a maximum code size of less than 64K bytes.</p> <p>Data space of up to 64K bytes, divided into three segments: Static, Dynamic, Public</p> <p>A given MULTOS implementation may impose a maximum data size of less than 64K bytes, and the boundaries between segments may also be different in different implementations.</p>
Address registers	<p>Static Base (SB) and Static Top (ST)</p> <p>Dynamic Base (DB), Local Base (LB) and Dynamic Top (DT). DT functions as the stack pointer and LB as the local frame pointer.</p> <p>Public Base (PB) and Public Top (PT)</p> <p>Each of these is sixteen bits.</p>
Control registers	<p>Code Pointer (CP): sixteen bits</p> <p>Condition Code Register (CCR) : eight bits</p>
Instruction length	Variable from 1 to 5 bytes

Table 1-1

1.2 Address Space

The AAM address space is divided into two independently addressable areas, one for code and one for data. (Certain primitives allow the application to read and write data outside its address space, but these are not discussed here.)

1.2.1 Code

The code space contains the application's instructions. It consists of up to 64K bytes of contiguous, non-volatile store, addressed from 0. An application program that is n bytes long is addressed from 0 to $n-1$. Applications cannot read or write this store, they can only execute it. The instructions are of variable length from one to five bytes.

Notation:

Since the code space is limited to 64K bytes, an address in the code space can be represented in a word. Such an address is called a *code address*. The first instruction in the program begins at code address zero. The code address of an instruction need bear no relation to its physical address in the underlying real hardware, and this physical address is unknown to the application.

While an instruction is executing, register CP contains the code address of the next instruction to be executed.

In some **MULTOS** implementations, applications can execute code that does not lie in their address space. Such implementations support *codelets*. Codelets are blocks of MEL code accessible to all applications. An application can execute a codelet using the Call Codelet (section 5.2) and Return From Codelet (section 5.18) primitives.

1.2.2 Data

The data space contains the application's volatile and non-volatile data. It consists of up to 64K bytes addressed from 0.

Notation:

Since the data space is limited to 64K bytes, an address in the data space can be represented in a word. Such an address is called a *segment address*. The segment address of a data item need bear no relation to its physical address in the underlying real hardware, and this physical address is unknown to the application.

Section 1

The data space is divided into three segments. Each segment is internally contiguous, but there may be gaps between the segments. All three segments can be read and written by the application, but cannot be executed. These segments are:

- Static, which is non-volatile
- Dynamic, which is volatile
- Public, which is volatile.

The data space is further defined by five properties:

- The bytes in each segment are contiguous, so applications can perform pointer arithmetic. For example, if the segment addresses 1212 and 1213 are both valid, and lie within the same segment, then they address adjacent bytes. An application may manipulate a segment address as it would any other number. (Obvious perhaps, but worth stating because of the importance of pointer arithmetic for many basic operations.)
- The segment address of the first byte of Static is zero, so the segment address of a given location in Static is always equal to its offset from the beginning of Static. If Static contains n bytes, the segment address of the last byte of Static is $n-1$. Data items in Static can be pointers to (segment addresses of) other data items in Static because Static is non-volatile; such pointers can, if required, be initialised before the application is loaded.
- In contrast, the segment address of any given location in Dynamic or in Public is not equal to its offset from the beginning of the segment. Moreover, the segment address of a location in Dynamic or Public is fixed only for the duration of one command-response pair. As a result, it is not safe for Static data items to be pointers to data in Dynamic or Public. Such pointers could not be initialised before the application was loaded, and would not necessarily remain valid between command-response pairs. (Of course, Static can be used to store data which only has duration of one command-response pair, but this is likely to be inefficient.) Some implementations may always provide the same segment address for a given offset in Dynamic or Public, but this behaviour is not guaranteed and applications should be written to behave correctly in the contrary case.

- The AAM is big-endian. Multi-byte blocks are stored with the least significant byte at the highest address. The address of a multi-byte block is the address of the most significant byte. Multi-byte blocks that are to be treated as numbers (for example, they may be incremented) are also stored this way with the bytes in strict order.
- Data is usually regarded as unsigned. However, it may also be regarded as signed. Some **MULTOS** implementations set condition code register flags to indicate when a result is negative or when signed overflow occurs. The signed value of a block is considered to be its two's complement form, where the most significant bit is the sign bit. For example, the two-byte block 0x8900 represents the decimal number 35072 when regarded as unsigned, and the decimal number -30464 when regarded as signed.

Because segment addresses in Dynamic and Public are not fixed, MEL instructions cannot refer to data using segment addresses. Instead, a *tagged address* is used. This is a nineteen-bit value consisting of a three-bit tag (address register number) and a sixteen-bit offset.

Notation:

An address used by an instruction has the format of a tag and an offset. The tag is one of the address registers; the offset is a word. Together, they make a *tagged address*.

The syntax used for writing tagged addresses is t[w]. For example, the tagged address of the first byte in Static is SB[0]. The tagged address of the last byte in Static is ST[0xffff]; word arithmetic wraps around, so this is equivalent to ST[-1].

In contrast to a segment address, which is a number that can be manipulated at runtime, a tagged address exists only in the program code.

Each of the seven address registers contains a segment address. The address registers SB (Static Base) and ST (Static Top) point to the boundaries of Static, DB (Dynamic Base) and DT (Dynamic Top) to the boundaries of Dynamic, and PB (Public Base) and PT (Public Top) to the boundaries of Public. For each segment, the Top register points to the byte immediately after the last valid byte. For example, the last valid byte of Static is ST[-1]. Register LB (Local Base) functions as a stack frame pointer. It points into Dynamic to indicate the start of the local data of the currently executing function.

Section 1

Note:

It is possible for an application program to corrupt LB, possibly to an address outside Dynamic, but all other registers are guaranteed correct at all times.

The following diagram shows the address registers pointing into the data space:

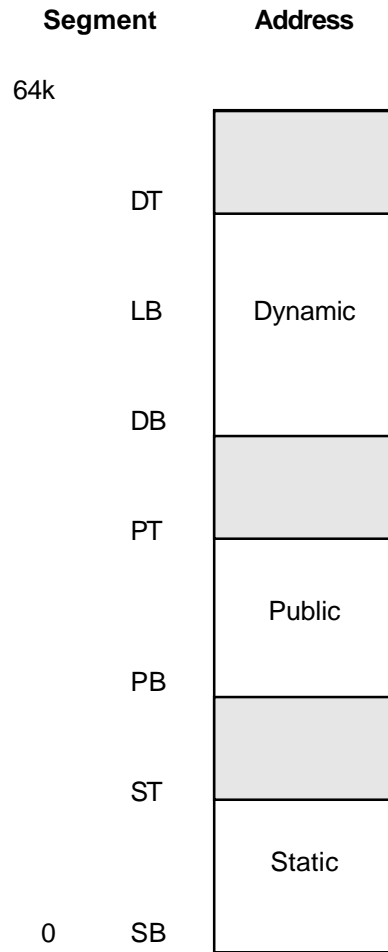


Table 1-2

As stated in the previous section, the segment address of a given location in Static is always equal to its offset from the beginning of Static. Furthermore, the segment address of a given location in Static is fixed for the life of an application, and is equal to its offset from SB.

Conversely, the segment address of any given location in Dynamic or in Public is fixed only for the duration of one command-response pair. Dynamic is addressed from DB[0] to DT[0xffff], and Public from PB[0] to

PT[0xffff], but the corresponding segment addresses only remain valid for the duration of one command-response pair.

Applications should make no assumptions about the relative locations of the segments. Although the diagram shows that DB>PT, it is equally possible that PB>DT. Although the diagram shows areas of inaccessible memory between the segments, these areas may be of zero size. In other words, it is possible that ST=PB, or PT=DB, or ST=DB.

Each segment is described in more detail in the following sections.

1.2.2.1 Static

The Static segment contains the application's non-volatile data.

The data is addressed using register SB (Static Base) and register ST (Static Top).

SB is defined to be zero. ST is equal to the size of the application's Static data; it is set when the application is loaded.

MULTOS guarantees that no other application can read or write this data.

Using current technology, Static is likely to be implemented in EEPROM (Electrically Erasable Programmable Read-Only Memory).

EEPROM has several characteristics that the programmer should keep in mind when writing software in native code:

- writing to it may be performed in *pages*, typically 32 bytes
- a power failure during a write may leave an entire page corrupted, even if the page contains data that is logically unaffected by the write .

MULTOS provides two mechanisms for removing this burden from the MEL application programmer:

- data item protection
- transaction protection.

Data item protection:

MULTOS guarantees that even if a data item spans page boundaries, it will either be completely updated or not updated at all. Moreover, other items on the same page will not be corrupted. Data item protection is always used. The application programmer need not be concerned with the page alignment of data, nor even know the page size of the underlying hardware.

Transaction protection:

MULTOS provides a mechanism for caching all writes, which the application can control. The application can turn caching on, perform several writes, and then perform a commit. If there is a power failure before the commit, no data items are updated, but once the commit has started **MULTOS** guarantees that all affected data items are updated.

Note: Uncommitted writes are not visible. Reading back from a data item that is the subject of an uncommitted write gives the value prior to the write. (See section 5.19.)

Transaction protection is controlled by an optional primitive, so it may not be available on all implementations.

1.2.2.2 Dynamic

The Dynamic segment contains the application's volatile data. Dynamic is divided into two parts, *session data* and *stack*. The size of the session data is a constant that is set when the application is loaded; for some applications, it may be zero. The stack is initially empty, but expands and contracts as the application executes.

The data is addressed from register DB (Dynamic Base) to register DT (Dynamic Top). Register LB (Local Base) serves as the local stack frame pointer for function calls. It addresses the topmost frame, that is, the currently executing function's local data. (See also section 1.6, "Functions".)

Register DT serves as the stack pointer. A one byte data item at the top of the stack is addressed as DT[-1], the next byte below is DT[-2], and so on. A push operation increments DT and a pop operation decrements DT. (See also sections 3.16, 3.24, and 3.25 for further explanations of stack operations.)

When an application is called to process an Application Protocol Data Unit (APDU), Dynamic contains the application's session data. At this

point, $DB[sizeSessionData] = LB[0] = DT[0]$; the stack is empty. (See section 2 for further explanation of APDU processing.)

DB is fixed for the duration of a command-response pair. Many instructions and primitives change DT, which may vary between $DB[sizeSessionData]$ and $DB[dynamicSize]$, where the value of `dynamicSize` is implementation-dependent. The CALL and PRIMRET instructions change LB. LB should normally remain between $DB[sizeSessionData]$ and $DT[0]$, but it is possible for an application to corrupt it.

MULTOS guarantees that no other application can read or write the data in Dynamic.

MULTOS guarantees that the application's session data is set as follows on entry:

- If this is the first command received by the application during the current session, the session data is filled with zeroes.
- Otherwise, the session data is unchanged from when the application last exited.

A new session begins whenever the Interface Device (IFD) successfully selects an application, even if the application is already selected. Delegating to an application is not the same as selecting it. (See section 2 for an explanation of delegation.) Consider two examples:

- Application A is selected, and receives commands X, Y and Z. It delegates all three commands to application B. Both applications will find that their session data has been zeroed when they receive X, but has not been zeroed when they receive Y and Z.
- Application A is selected, and receives commands X, Y and Z. It processes X itself, but delegates Y and Z to application B. Application A will find that its session data has been zeroed when it receives X, but not when it receives Y and Z. Application B will find that its session data has been zeroed when it receives Y, but not when it receives Z.

A typical use for session data is to support the use of a session PIN (Personal Identification Number). The application could reserve one byte

of session data to support the PIN-received flag. On receiving a command, the application could update the flag as follows:

```
if ( PIN command and PIN == stored PIN )
    set DB[0] to 1 ;
else if ( DB[0] is 1 )
    Process command ;
else
    Reject command ;
```

The optional primitive Reset Session Data allows a shell application to reset the session data of all other applications. (See section 5.17.)

1.2.2.3 Public

The Public segment is used for command and response data passed between an IFD and an application. During delegation, Public contains data passed between two applications, the delegator and the delegate. (See section 2 for an explanation of delegation.) Applications may also use Public as temporary working storage.

The data is addressed from register PB (Public Base) to register PT (Public Top). When an application is called to process an APDU, Public contains the APDU. (See section 2 for further explanation of APDU processing.)

PB and PT are fixed for the duration of a command-response pair. The difference between them is the size of Public.

MULTOS guarantees that data in Public remains private to the application until the application exits or delegates. Public is then made available as follows:

- If the application delegates, the whole of Public becomes available to the delegate.
- If the application exits, and is itself a delegate, the whole of Public becomes available to the delegator.
- If the application exits, and is not itself a delegate, then SW1, SW2 and La bytes of response data are made available to the IFD. (SW1, SW2, and La response parameters are defined by ISO 7816. See also section 2.)

An application may write secret data into Public, but it holds responsibility for overwriting this data before delegating or exiting. To understand the conditions of responsibility for secret data, application programmers should be aware that:

- An application does not, in general, know if it is a delegate, so it should not leave secret data in Public on exit. (An application can explicitly determine whether it is a delegate, and find out the identity of its delegator, by calling the optional primitive Get Delegator AID, if available. See section 5.10.)
- If the application has an abnormal end (*abend*), **MULTOS** destroys data in Public automatically, so no secret data is revealed.
- If the **MULTOS** Carrier Device (MCD) is reset, **MULTOS** destroys data in Public automatically, so no secret data is revealed.

Section 2 describes the use of Public as a location for passing information between an IFD and an application, or between applications.

1.3 Registers

There are seven address registers, and two control registers.

1.3.1 Address Registers

Register number	Register mnemonic	Register name
1	SB	Static base
2	ST	Static top
3	DB	Dynamic base
4	LB	Local base
5	DT	Dynamic top
6	PB	Public base
7	PT	Public top

Table 1-3

Note: The register number above, expressed as three-bit value, identifies the register when used as the tag component of a tagged address. (See also section 1.4.4)

1.3.2 Control Registers

Two registers, the Code Pointer and the Condition Code Register control program flow.

Register mnemonic	Register name
CP	Code Pointer
CCR	Condition Code Register

Table 1-4

The CCR contains eight bits: four available for use by the application programmer, and four that are set or cleared depending on the result of an instruction. Applications can control program flow based on the condition codes using the BRANCH (section 3.6), CALL (Section 3.7) and JUMP (section 3.15) instructions. Applications can access and manipulate the CCR directly using the mandatory primitives Load CCR (section 4.12) and Store CCR (section 4.25).

Not all **MULTOS** platforms support signed arithmetic. Since traditional smartcard applications do not typically need signed arithmetic, application developers should assume that most current implementations do not support signed arithmetic. This may change in the future.

Platforms without signed arithmetic neither set nor clear N and V.

Platforms with signed arithmetic set N and V as specified for each instruction or primitive. N and V are read using the Load CCR primitive and written using the Store CCR primitive. (Store CCR writes all CCR flags, including N and VC, even on platforms that do not support signed arithmetic.) Some platforms may also support additional primitives to make use of N and V.

The application provider should ensure that signed arithmetic is supported by a particular platform before performing operations that rely on signed arithmetic.

7	6	5	4	3	2	1	0
-	-	-	-	C	V	N	Z

Table 1-5

- bits 4 to 7:** These bits may be used by the application programmer.
- bit 3 - Carry (C):** This bit indicates a carry or borrow at the most significant bit. Carry occurs when an unsigned value exceeds its limits.
- bit 2 - Overflow (V):** This bit is set to 1 when arithmetic overflow occurs, and cleared to 0 at other times. Arithmetic overflow occurs when a signed value exceeds its limits. For example, adding 1 to the byte value 0x7F gives a result, 0x80, that is negative when treated as a signed byte. Since the result of adding two positive numbers should always be positive, the overflow bit is set to indicate an abnormal result.
- bit 1 - Negative (N):** This bit indicates the most significant bit (sign bit) of the result of an instruction.
- bit 0 - Zero (Z):** This bit is set to 1 to indicate a zero result and cleared to 0 to indicate a non-zero result.

Not all **MULTOS** platforms support signed arithmetic. Since traditional smartcard applications do not typically need signed arithmetic, application developers should assume that most current implementations do not support signed arithmetic. This may change in the future.

Platforms without signed arithmetic neither set nor clear N and V.

Platforms with signed arithmetic set N and V as specified for each instruction or primitive. N and V are read using the Load CCR primitive and written using the Store CCR primitive. (Store CCR writes all CCR flags, including N and V, even on platforms that do not support signed arithmetic.) Some platforms may also support additional primitives to make use of N and V.

The application provider should ensure that signed arithmetic is supported by a particular platform before performing operations that rely on signed arithmetic.

1.3.3 Initial Register Values

All address and control registers are set to defined values when an application is loaded or at the moment when the operating system passes control to the application, as follows:

- SB is always zero.
- ST is equal to the number of bytes in the application's Static, initialised when the application is loaded.
- The remaining address registers are initialised when the application is passed control by **MULTOS**. (See also section 2.2, Application Entry.)
- CP is set to zero when the application is passed control by **MULTOS**. Execution begins at code address 0.
- All eight bits of the CCR are cleared.

1.4 Instructions

This section introduces the instructions by category and explains how instructions are coded and laid out within an application's code address space. Each individual instruction is given a detailed explanation later in section 3.

1.4.1 Types of Instruction

The following table classifies all 31 instructions. The PRIMRET instruction occurs twice in the table because it acts both as a call to a primitive (classified as system control) and as a return (classified as flow control).

Function	Instructions
System control	SYSTEM, PRIMRET (when used to call a primitive)
Flow control	BRANCH, JUMP, CALL, PRIMRET (when used as return)
Stack operators	STACK, LOAD, STORE, LOADI, STOREI, LOADA, INDEX
Literal operators (act on byte or word data at a specified tagged address, using data supplied in the instruction)	SETB, CMPB, ADDB, SUBB, SETW, CMPW, ADDW, SUBW
Unary block operators (act on block data at a specified tagged address)	CLEARN, TESTN, INCN, DECN, NOTN
Binary block operators (act on block data at a specified tagged address, using data at the top of Dynamic)	CMPN, ADDN, SUBN, ANDN, ORN, XORN

Table 1-6

1.4.2 Special characteristics of block operators

Binary block operators are guaranteed safe. Even if parameters overlap, the answer is correct.

To achieve this, some **MULTOS** platforms may require temporary working memory on the stack equal to the block size. For example, when adding two eight byte blocks, there should be at least eight bytes available on the stack.

For related reasons, unary block operators may also require temporary storage on the stack, as may certain primitives.

Note: If sufficient temporary space is not available when executing a block operation, the application abends. (See also section 1.5.)

1.4.3 Instruction Functions

The following tables give brief descriptions of the instructions in each functional group.

System control

Instructions	Function
SYSTEM	Either no operation, or any combination of: <ul style="list-style-type: none"> • set La with a literal word • set SW1, SW2 with a literal word • exit to MULTOS.
PRIMRET, when used to call a primitive	Call the specified primitive function.

Table 1-7

Flow control

These instructions control program flow, conditionally or unconditionally. Conditional control is based on combinations of CCR flags C and Z being set and not set, where the combinations have been chosen to give the normal mathematical operations for unsigned integers. (These combinations are specified in the description of each flow control instruction in section 3.)

Instruction	Function
BRANCH	Branch to the given code address. The branch may be unconditional, or conditional on the CCR C and Z flags.
JUMP	Jump to the given code address or to the code address located at the given tagged address. The jump may be unconditional, or conditional on the CCR C and Z flags. Unconditional indirect jumps are also supported.
CALL	Call the function at a given code address or at the code address located at the given tagged address. The call may be unconditional, or conditional on the CCR C and Z flags. Unconditional indirect calls are also supported.
PRIMRET, when used to call a primitive	Return from the current function (remove the current stack frame, and replace the multi-byte block that was the passed parameters with the multi-byte block that is the return parameters).

Table 1-8

Stack operators

The following table lists the instructions that perform stack operations. They all affect the part of the Dynamic segment allocated to the stack and accessed using register DT.

Instruction	Function
STACK	Push onto the stack or pop from the stack one of: <ul style="list-style-type: none"> • a literal byte • a literal word • an initialised (set to 0) multi-byte block.
LOAD	Push a multi-byte block stored at a tagged address onto the stack.
STORE	Pop a multi-byte block from the stack and store it at a tagged address.
LOADI	Fetch a segment address from a given tagged address, and push a multi-byte block from that segment address onto the stack.
STOREI	Fetch a segment address from a given tagged address, and pop a multi-byte block from the stack and store it at that segment address.
LOADA	Convert a given tagged address to a segment address and push the converted segment address onto the stack.
INDEX	This instruction is typically used to index arrays. Multiply an array index stored at the top of Dynamic by the record size. Convert a given tagged address to a segment address, then add it to the product and push the result onto the stack.

Table 1-9

Literal operators

The following table lists the instructions that perform operations using literal values that are encoded as part of the instruction.

Instruction	Function
SETB/W	Set a byte/word at a tagged address to a literal byte/word.
CMPB/W	Compare a literal byte/word with a byte/word at a tagged address.
ADDB/W	Add a literal byte/word to a byte/word at a tagged address.
SUBB/W	Subtract literal byte/word from a byte/word at a tagged address.

Table 1-10

Unary block operators

The following table lists the instructions that perform operations using a single block of data.

Instruction	Function
CLEARN	Clear (set to zero) a multi-byte block either at the top of Dynamic or at a tagged address.
TESTN	Test (compare with zero) a multi-byte block either at the top of Dynamic or at a tagged address.
INCN	Increment a multi-byte block either at the top of Dynamic or at a tagged address.
DECN	Decrement a multi-byte block either at the top of Dynamic or at a tagged address.
NOTN	Bitwise NOT a multi-byte block either at the top of Dynamic or at a tagged address.

Table 1-11

Binary block operators

The following table lists the instructions that perform operations using two blocks of data.

Instruction	Function
CMPN	Compare a multi-byte block at the top of Dynamic with a multi-byte block either below it at the top of Dynamic or at a tagged address.
ADDN	Add a multi-byte block at the top of Dynamic to a multi-byte block either below it at the top of Dynamic or at a tagged address.
SUBN	Subtract a multi-byte block at the top of Dynamic from a multi-byte block either below it at the top of Dynamic or at a tagged address.
ANDN	Bitwise AND a multi-byte block at the top of Dynamic with a multi-byte block either below it at the top of Dynamic or at a tagged address.
ORN	Bitwise OR a multi-byte block at the top of Dynamic with a multi-byte block either below it at the top of Dynamic or at a tagged address.
XORN	Bitwise XOR a multi-byte block at the top of Dynamic with a multi-byte block either below it at the top of Dynamic or at a tagged address.

Table 1-12

1.4.4 Machine-Language Coding

This section describes how instructions and their arguments are encoded as they appear in an application's code address space at runtime. This is the format used by programmers working directly in MEL and the output format used by developers of higher-level tools such as assemblers or compilers.

All AAM instructions have a common format:

- the op-field: the five most significant bits of the first byte of the instruction
- the t-field: the three least significant bits of the first byte of the instruction
- up to 4 bytes of arguments (b-fields or w-fields). These may be treated either as bytes or as words. When they are used as bytes, they are notated as b1, b2, b3, b4. When they are used as words, w1 designates b1 and b2 (evaluated as $b1*256+b2$), w2 designates b2 and b3, and w3 designates b3 and b4.

The op-field contains the opcode; there are thus 32 possible instructions, of which 31 are currently defined.

The t-field (tribit) is used as:

- the size of the instruction
- an enumeration to distinguish conditions or between the separate uses of the PRIMRET and STACK opcodes
- the tag (address register number) of a tagged address.

The b-fields (byte), if present, are used as:

- a relative code address (signed)
- a literal byte
- the size of a multi-byte block.

The w-fields (word), if present, are used as:

- an absolute code address
- a literal word
- the offset of a tagged address.

1.5 Abending

In the course of executing, the application may attempt to violate system integrity. An application that does this is terminated (abnormally ends) before it does any damage. The application is said to have *abended*.

In current **MULTOS** implementations, the effect of an abend is that the MCD no longer responds to the IFD. The IFD is expected to time-out and then reset the MCD. The application that abended remains available for selection. Data that is in Public at the time of the abend is destroyed either by the abend or by the subsequent reset.

Possible reasons for an abend are:

- Invalid CP (trying to fetch an instruction from outside the application's code space). Note that the whole instruction must lie within the address space; it is not sufficient for the start of the instruction to do so.
- Invalid opcode (one opcode is reserved for future use, and must not currently be used).
- Invalid t-field (some opcodes are defined for all values of t, while others are only defined for some values).
- Invalid primitive (trying to execute a primitive that is not supported on the current platform).
- Invalid codelet (trying to execute a codelet that is not supported on the current platform).
- Invalid segment address (trying to fetch data from outside the application's data space).
- Stack underflow (trying to pop too many bytes from the stack).
- Stack overflow (trying to push too many bytes on to the stack).
- Corrupt stack frame (LB is inconsistent with a PRIMRET return instruction).

- Insufficient temporary space (trying to execute a unary or binary block operation, or certain primitives, when insufficient stack space is available to hold intermediate results. See also section 1.4.2).
- Too many writes to non-volatile storage while transaction protection is on.

In addition, the effect of specifying a multi-byte block of length zero in an instruction is undefined. It may cause an abend, it may execute but change no data, or it may execute with some change to data. There is no valid reason for a program to specify zero-length blocks.

1.6 Functions

This section describes the **MULTOS** function call mechanism, including parameter passing and return result, as well as their respective memory layout and register usage.

Execution begins when an application's entry point is called. For purposes of explanation of the function call mechanism, we refer to this as the application's main function, although it is entirely possible that it is not a function in the strict sense. That is, application exit may occur from within another function.

When an application's entry point, or main function, is called, the Dynamic address registers are initialised (the memory is shown as being allocated from left to right):

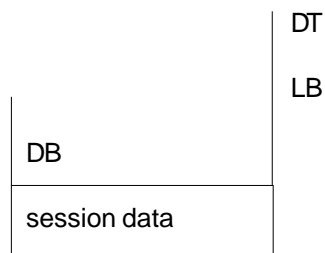


Table 1-13

The main function may allocate Dynamic using load (LOAD) and/or push (STACK) instructions.

Section 1

The Dynamic memory allocated at any time is called the main function's stack frame.

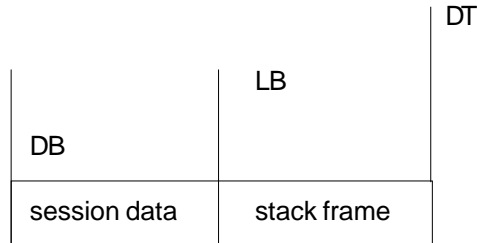


Table 1-14

All functions may call other functions by using the CALL instruction. The called function returns control to the caller by using the return version of the PRIMRET instruction. CALL pushes four bytes of link data onto the stack. Return pops these bytes off again, together with any bytes subsequently placed on the stack by the called function. These four bytes of link data contain the caller's LB and the return address. (See also the definition of the CALL instruction in section 3.7.) It is inadvisable for applications to change these four bytes, though **MULTOS** does not prevent them doing so.

To make it easier for the MEL programmer to write functions that take parameters and return results, PRIMRET also supports parameter passing. To pass parameters, the caller places the **in** parameters at the top of the stack before calling the function. The called function may then push further bytes onto the stack, and may refer to the **in** parameters by using the tagged addresses from LB[-4] downwards (LB[-5], LB[-6], and so on, for bytes). The called function places its **out** parameters at stack top, then executes the return instruction. The return instruction specifies the number of bytes in the **in** parameters and in the **out** parameters. The return instruction pops all data from the stack down to and including the **in** parameters, then pushes the **out** parameters on to the stack again.

For example, consider a function SUM3 that takes three words as **in** parameters and returns their sum in a word. Before calling SUM3, the caller loads the three words on to the stack:

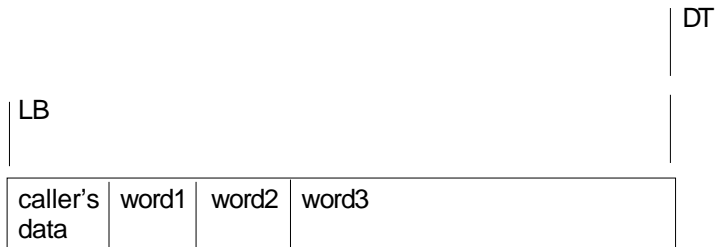


Table 1-15

The caller now calls the function SUM3, using the CALL instruction. The effect is:

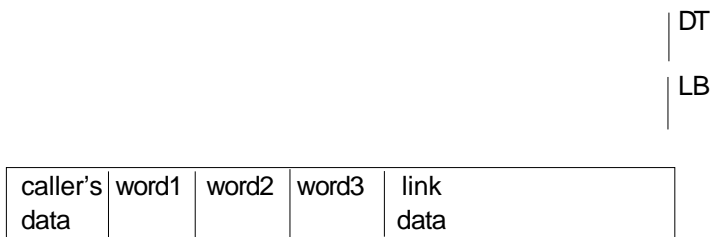


Table 1-16

SUM3 now allocates some storage, and eventually calculates the sum of the three words:

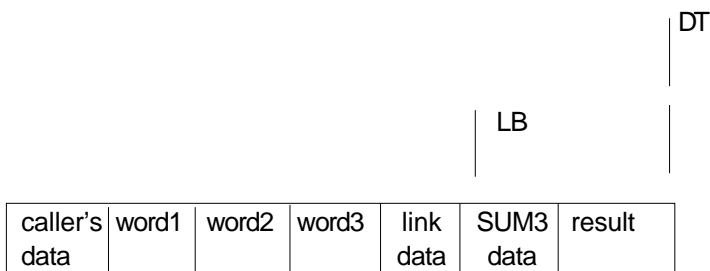


Table 1-17

1.7.1 Arithmetic, bitwise and block operation primitives

These primitives extend the functionality of the core instructions for arithmetic, bitwise, and block operations.

	Mandatory	Optional
Arithmetic	DivideN. MultiplyN.	
Bitwise	Bit Manipulate Byte. Bit Manipulate Word. Shift Left. Shift Right.	
Block	Lookup. Memory Compare. Memory Compare Fixed Length. Memory Copy. Memory Copy Fixed Length.	Checksum

Table 1-19

1.7.2 Cryptographic primitives

These primitives, all optional, provide many of the common cryptographic functions associated with smartcard applications.

	Mandatory	Optional
Asymmetric		Generate Asymmetric Hash. Modular Exponentiation. Modular Exponentiation CRT. Modular Multiplication. Modular Reduction.
Miscellaneous		Get Random Number. SHA-1.
Symmetric		DES ECB Decipher. DES ECB Encipher. Generate DES CBC Signature. Generate Triple DES CBC [Signature.

Table 1-20

1.7.3 Operating system access primitives

These primitives, both mandatory and optional, provide additional communication between application and system and access to system information and resources.

	Mandatory	Optional
Communication with the IFD and with other applications.	Check Case. Reset WWT.	Delegate. Get Delegator AID. Reset Session Data.
Memory management.	Load CCR. Query0, Query1, Query2, Query3. Store CCR.	Call Codelet Query Codelet Return From Codelet
Interpreter functions.	Get Memory Reliability.	Set Transaction Protection.
Operating system data.	Get DIR File Record. Get File Control Information. Get Manufacturer Data. Get MULTOS Data. Get Purse Type. Set ATR File Record Set ATR Historical Characters.	

Table 1-21

This overview of the AAM environment should provide a sufficient conceptual framework for understanding the remaining sections of this reference manual, which describe and explain in more detail the interaction between Interface Device (IFD) and application, the instruction set, and the sets of mandatory and optional primitives.

2. IFD – Application Interface

This section describes the communications interface between the Interface Device (IFD) and an application, including

- use of the Public data segment as a communications buffer for command-response parameters
- application entry and exit
- delegation

Complete understanding of this section requires familiarity with the mechanism and parameters of Application Protocol Data Unit (APDU) processing, as defined in ISO 7816, Part 4.

Applications interact with the IFD by receiving commands, processing them, and returning responses across the IFD - Application Interface. The precise manner in which commands are presented to applications depends on the mode of operation.

MULTOS has two modes of operation: *standard* mode (the default, normal operating system mode) and *shell* mode. Under certain circumstances, an application may be designated a *shell application*. Only one shell application can be loaded at any time. When a shell application is loaded, **MULTOS** is in shell mode, otherwise, it is in standard mode.

The difference between the two modes is that:

- in *standard* mode, a SELECT FILE command may change the currently selected file. Subsequent commands are routed to the (newly) selected file for processing.
- in *shell* mode, all application-level commands are routed to the shell application, which may in turn delegate them to other applications. In shell mode, the shell application is implicitly selected after a reset.

The only exceptions are system-level commands that require special access privileges (**MULTOS** Security Manager commands). These are always processed by **MULTOS**, regardless of the operational mode.

The commands to load and delete applications, for example, are privileged commands. Therefore, deleting the shell application returns **MULTOS** to standard mode.

Note: The details of how applications are loaded and deleted are beyond the scope of this manual. Developers who want to write shell applications should consult the **MULTOS** Design Standard, IFD-**MULTOS** Interface Specification, which contains a more complete explanation of the two modes of operation.

2.1 Using Public as the communications buffer

Although the Public data segment may be used as a general purpose volatile store, it also has a dedicated purpose for communications. Public is mapped to the communications buffer as follows:

Public	Use
PB[0]	Data field (either command or response)
PT[-17]	ProtocolFlags, where: bit 0: P3Valid bit 1: LcValid bit 2: LeValid bit 3: CmdDataRxed
PT[-16]	ProtocolType
PT[-15]	GetResponseCLA
PT[-14]	GetResponseSW1
PT[-13]	CLA
PT[-12]	INS
PT[-11]	P1
PT[-10]	P2
PT[-9]P3	
PT[-8]Lc (as a word)	
PT[-6]Le (as a word)	
PT[-4]La (as a word)	
PT[-2]SW1	
PT[-1]SW2	

Table 2-1

2.2 Application entry

When the application starts executing,

- The stack is empty: LB[0] and DT[0] refer to the same segment address in Dynamic, just beyond the session data.
- The CCR is clear.

The following data is in Public:

- CLA (class byte), INS (instruction byte), P1 and P2 (parameters) are set as in the command APDU.
- Get Response CLA is set to the CLA of the received command. (**MULTOS** also responds to a GetResponse command with a CLA of zero)
- GetResponseSW1 is set to 0x61; this is the SW1 (switch 1) returned to the IFD to elicit a GetResponse command in those instances where **MULTOS** automatically supports GetResponse without reference to the application. (These are in T=0, in case 4 and in case 2 when Le, expected length of data to be returned, is zero.)
- ProtocolType is set to 0 in a T=0 card and to 1 in a T=1 card.
- La (the actual length of data to be returned) is set to zero.
- SW1 and SW2 are set to 0x9000.

Depending on the protocol, further bytes may also be set in Public:

- In a T=0 card, P3 is set as in the command header.
- In a T=1 card, Lc (command length) and Le are set, and the command data, of length Lc, is present from PB[0].

Note: Lc, Le and La are words, not bytes.

The application can check the protocol type in order to find out whether it is running under T=0 or T=1. However, this is not recommended, and may give incorrect results if future MCDs support additional protocols.

The more robust and recommended approach is for the application to inspect CLA, INS, P1 and P2, and determine from these whether the command is:

- ISO case 1 (no command data, no response data)
- ISO case 2 (response data but no command data)
- ISO case 3 (command data but no response data)
- ISO case 4 (command data and response data)
- unrecognised

To make this possible, new applications should be designed so that it is always possible to determine the case by examining these four bytes. For pre-existing interface definitions where this is not true, the application may have to check P3 and/or Lc and/or the P3Valid and LcValid flags, in order to determine the case of the command.

If the application recognises the command and knows what case it is, it should then call the (mandatory) Check Case primitive. On a T=1 card, the primitive checks that Lc and Le are consistent with the case specified by the application. On a T=0 card, the primitive checks P3 for consistency with the specified case, and, in cases 3 and 4, request P3 bytes of command data from the IFD. See the description of the Check Case primitive in section 4.4 for further details.

Once the application has called Check Case, it may then access the command data and Lc without needing to know whether it is running under T=0 or T=1.

Note: Le is also available on T=1 cards, but is only available on T=0 cards in case 2 when P3 is non-zero. The application can inspect the LeValid flag to determine whether Le contains useful data.

If the application does not recognise the command, it should not call Check Case, but should exit as described below.

2.3 Application exit

Upon completion of processing the APDU, before it exits, the application should:

- place any response data into Public starting at PB[0].

- set SW1 and SW2 to the desired values (if not 0x9000).
- set La to the actual length of response data (if not zero).

The application should not attempt to return any response data unless it has called Check Case and has specified case 2 or case 4 in so doing. For unrecognised commands, where the application will not have called Check Case, the application should set SW1, SW2 to an appropriate response and should leave La set to zero.

2.4 Delegation

The description of communications in the previous sections assumes that the application is receiving commands directly from an IFD, and returning responses directly to the IFD. However, this need not be the case.

By a mechanism called *delegation*, an application (the *delegator*) can request another application (the *delegate*) to process a command on its behalf. This can be useful in several situations; two examples are given in the following sections.

2.4.1 Command rerouting

The delegator may receive a command that it does not recognise or does not wish to process. However, it does not wish to reject the command and respond to the IFD, but prefers to pass the command to a delegate.

In order to delegate, the delegator calls the Delegate primitive. Execution of the delegator is suspended, and the delegate is executed instead. In most cases, the delegate processes the command exactly as though the command had arrived directly from an IFD.

When the delegate has finished processing the command, and has written its response into Public, it exits as normal.

Execution of the delegator then resumes at the instruction after the instruction which calls the Delegate primitive. The delegator may simply exit in turn, thus sending the response to the IFD, or may carry out further processing before exiting.

2.4.2 Obtaining data from another application

Another situation where delegation is useful is where two applications wish to share data.

Consider the following requirement:

- Application A shall respond to a command B, which shall cause it to return a data item N.
- Application Z shall respond to a command Y, which shall cause it to respond with the same data item N.

This can be achieved as follows:

- Application A stores N in Static and returns it whenever it receives command B.
- Application Z responds to command Y as follows:
 - It constructs a command B APDU in Public.
 - It delegates to A.
 - When control is returned from A, Z performs any necessary reformatting of the response APDU and then exits.

A possible complication is that the Check Case primitive may be called twice, once by the delegator and once by the delegate. This is not an error, but it may cause confusion if the two applications consider the command to have different ISO cases. Since the delegate does not generally know that it is acting as such, it is the responsibility of the delegator to ensure that no confusion arises.

There are a number of situations in which careful design of the delegating application is necessary to avoid confusion. One example is given here. Suppose that command Y, executed by the delegator Z, is ISO case 2, but command B, executed by the delegate A, is ISO case 4. There is the danger of the following confusion:

- Application Z receives command header Y.
- Application Z calls Check Case, specifying case 2.
- Application Z builds a command B, including command data, in Public.
- Application Z delegates to application A.
- Application A calls Check Case, specifying case 4.
- **MULTOS** responds to the Check Case call by sending a T=0 procedure byte in order to elicit data from the IFD.
- The IFD knows nothing of command B; it has issued command Y, which is case 2, so it waits for response data.
- **MULTOS** waits for command data. Both sides of the interface are now waiting for events that never occur.
- Eventually, the IFD times out the MCD.

This confusion can be prevented if application Z sets the CmdDataRxd (command data received) flag in Public before it delegates. This tells **MULTOS** that the command data has already been received (in reality it has been constructed by application Z) and so the second call to Check Case works correctly.

**Section
2**

3. Instruction Set

This section presents all 31 core AAM instructions in alphabetic order and gives a full description of each one.

3.1 Introduction

Each instruction is explained under the headings *Operation*, *Condition Code*, *Description*, and *Instruction Formats*. The following sections describe the notational conventions used and what information is found under each heading.

Operation

Operation gives a formal description of the instruction's effect on the AAM CPU.

Notation:

SB, ST, DB, LB, DT, PB, and PT are the address register mnemonics as defined in section 1.3.1.

CP is the code pointer, expressed as a word offset from the beginning of the code. When an instruction is executing, CP has already been incremented to point to the first byte of the next instruction

When an instruction changes LB, DT or CP, the notation LB', DT', CP' is used to indicate the final value of these registers. The value of CP' is shown only for instructions which change the flow of control; all instructions automatically increment CP is described in the previous bullet point.

t is the tribit field of an instruction, as explained in section 1.4.4.

b1, b2, b3, b4 are byte fields of an instruction, as explained in section 1.4.4.

w1, w2, w3 are word fields of an instruction, as explained in section 1.4.4.

t[w] is the segment address of the byte which is displaced w bytes from the location addressed by register number t.

{t[w], 1} is an assignable reference to that byte.

{t[w], b} is an assignable reference to the b bytes with addresses t[w] through t[w+b-1]. For example, the segment address of the top word in Dynamic is DT[-2]. Its contents are {DT[-2], 2}. The byte it addresses indirectly is {{DT[-2], 2}, 1}

cCLA, cSW1, etc. are constants defined relative to PT to give the IFD-Application interface described in section 2.1. Therefore, PT[cCLA] is

Section 3

the segment address of the CLA byte, and {PT[cSW1], 2} is an assignable reference to the SW1 and SW2 bytes.

For purposes of clarification, comparison with C language notation may be of help. In C, if p were defined as a pointer and contained the segment address of the top word in Dynamic, then p would be equivalent to DT[-2], *p would be equivalent to {DT[-2], 2}, and **p would be equivalent to {DT[-2], 2}, 1}.

Condition Code

Condition Code describes how the CCR flags are affected by the instruction.

Note: the effect of each instruction on the N and V flags is given, although current implementations of MULTOS may not support these flags.

Description

Description gives textual description of the operation of the instruction.

Instruction Formats

Instruction Formats gives a tabular presentation of the possible formats for the instruction. The table also shows the opcode as a number in the range 0 to 0x1f.

3.2 ADDB (ADD literal to Byte)

Operation

```
t=0
    {DT[-1], 1} += b1;
t=tag
    {t[w2], 1} += b1;
```

Condition Code

	C	V	N	Z
	-	-	-	-
	↕	↕	↕	↕

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the byte at DT[-1], or
- the byte at t[w2]

by adding the literal byte b1.

Instruction Formats

Mnemonic	op	t	b1	w2
ADDB	0x0f	0	literal byte	
ADDB	0x0f	tag	literal byte	address offset

3.3 ADDN (ADD N-byte blocks)

Operation

```

t=0
    {DT[-2*b1], b1} += {DT[-b1], b1};
t=tag
    {t[w2], b1} += {DT[-b1], b1};
    
```

Condition Code

				C	V	N	Z
–	–	–	–	↕	↕	↕	↕

Section 3

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- _ the block of length b1 that starts at DT[-2*b1], or
- _ the block of length b1 that starts at t[w2]

by adding the block of length b1 that starts at DT[-b1]. Each block is treated as an unsigned integer of length b1 bytes.

Instruction Formats

Mnemonic	op	t	b1	w2
ADDN	0x1b	0	block length	
ADDN	0x1b	tag	block length	address offset

3.4 ADDW (ADD literal to Word)

Operation

```

t=0
{DT[-2], 2} += w1;
t=tag
{t[w3], 2} += w1;

```

Condition Code

				C	V	N	Z
-	-	-	-	⇕	⇕	⇕	⇕

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- _ the word that starts at DT[-2], or
- _ the word that starts at t[w3]

by adding the literal word w1.

Instruction Formats

Mnemonic	op	t	w1	w3
ADDW	0x13	0	literal word	
ADDW	0x13	tag	literal word	address offset

3.5 ANDN (bitwise AND N-byte blocks)

Operation

```

t=0
    {DT[-2*b1], b1} &= {DT[-b1], b1};
t=tag
    {t[w2], b1} &= {DT[-b1], b1};
    
```

Condition Code

					C	V	N	Z
-	-	-	-	-	0	↕	↕	↕

- C Previous value remains unchanged
- V Cleared
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the block of length b1 that starts at DT[-2*b1], or
- the block of length b1 that starts at t[w2]

by performing a bitwise AND operation between it and the block of length b1 that starts at DT[-b1].

Instruction Formats

Mnemonic	op	t	b1	w2
ANDN	0x1d	0	block length	
ANDN	0x1d	tag	block length	address offset

3.6 BRANCH (BRANCH to relative code address)

Operation

```

switch (t)
{
case EQ : temp = Z;          break;
case LT : temp = C;          break;
case LE : temp = (C || Z);  break;
case GT : temp = !(C || Z); break;
case GE : temp = !C;         break;
case NE : temp = !Z;         break;
case A  : temp = TRUE; break;
}
if ( temp )
{
    CP' += (signed)b1;
}

```

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction conditionally transfers control to the code address formed by adding b1 as a signed byte to CP. For the avoidance of doubt, note that CP is the code address of the instruction after the BRANCH. This implies that:

- if b1=0, the BRANCH instruction has no effect.
- if b1=0xfe (which evaluates to -2 as a signed byte), then, if the condition is true, the program will loop endlessly.

The condition is specified in t:

t	Mnemonic	Meaning
1	EQ	Equal
2	LT	Less Than
3	LE	Less than or Equal
4	GT	Greater Than
5	GE	Greater than or Equal
6	NE	Not Equal
7	A	Always

Note: the conditions based on the state of the Carry flag, LT, LE, GT, and GE, apply to unsigned comparisons. **MULTOS** does not directly support conditional program flow based on signed comparisons. If needed, the application developer could implement it based on the condition of the N and V flags on platforms that support signed arithmetic. (The *Load CCR* primitive, described in section 4.12 allows access to the N and V flags.)

Instruction Formats

Mnemonic	op	t	b1
BRANCH	0x01	1 - 7	relative code pointer

3.7 CALL (CALL function)

Operation

```

if ( t > 0 )
{
    switch (t)
    {
        case EQ : temp = Z;          break;
        case LT : temp = C;          break;
        case LE : temp = (C || Z);  break;
        case GT : temp = !(C || Z); break;
        case GE : temp = !C;        break;
        case NE : temp = !Z;        break;
        case A  : temp = TRUE;       break;
    }
    if ( temp )
    {
        DT' = DT + 4;
        LB' = DT';
        CP' = w1;
        {DT[0], 2} = LB;
        {DT[2], 2} = CP;
    }
}
else // t == 0
{
    DT' = DT + 2;
    LB' = DT';
    CP' = {DT[-2], 2};
    {DT[-2], 2} = LB;
    {DT[0], 2} = CP;
}

```

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

If t is non-zero, it specifies a condition as follows:

t	Mnemonic	Meaning
1	EQ	Equal
2	LT	Less Than
3	LE	Less than or Equal
4	GT	Greater Than
5	GE	Greater than or Equal
6	NE	Not Equal
7	A	Always

If the condition is true, four bytes of link data are pushed on to the stack, and control is directed to the code address specified as w1.

If t is zero, two bytes are popped from the stack. (Note: this means that the bytes must not lie in session data, as session data cannot be popped.) Four bytes of link data are pushed on to the stack, and control is directed to the code address specified by the two popped bytes.

See section 1.6 for further details.

Note: the conditions based on the state of the Carry flag, LT, LE, GT, and

GE, apply to unsigned comparisons. MULTOS does not directly support conditional program flow based on signed comparisons. If needed, the application developer could implement it based on the condition of the N and V flags on platforms that support signed arithmetic. (The *Load CCR* primitive, described in section 4.12 allows access to the N and V flags.)

Instruction Formats

Mnemonic	op	t	w1	
CALL	0x03	0		
CALL	0x03	1 - 7	code address	

3.8 CLEARN (CLEAR N-byte block to 0)

Operation

t=0
 {DT[-b1], b1} = 0;
 t=tag
 {t[w2], b1} = 0;

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction clears to zero either:

- the block of length b1 that starts at DT[-b1], or
- the block of length b1 that starts at t[w2].

Instruction Formats

Mnemonic	op	t	b1	w2
CLEARN	0x15	0	block length	
CLEARN	0x15	tag	block length	address offset

3.9 CMPB (CoMPare literal with Byte)

Operation

t=0
 {DT[-1], 1} - b1;
 t=tag
 {t[w2], 1} - b1;

Condition Code

				C	V	N	Z
-	-	-	-	Ô	Ô	Ô	Ô

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction subtracts the literal byte b1 from either:

- the byte at DT[-1], or
- the byte at t[w2].

No data is modified, but the CCR is set according to the result of the subtraction.

For example, suppose SB[2] contains the value 4. Then:

- “CMPB SB[2] with 4” obtains the result 0 from the subtraction. The instruction sets the Z flag and clear the C flag. The conditions EQ, LE and GE will be true, and the conditions NE, LT and GT will be false.
- “CMPB SB[2] with 3” obtains the result 1 from the subtraction. The instruction clears the Z flag and the C flag. The conditions NE, GT and GE will be true, and the conditions EQ, LE and LT will be false.
- “CMPB SB[2] with 5” obtains the result -1 (0xff) from the subtraction. The instruction clears the Z flag and sets the C flag. The conditions NE, LT and LE will be true, and the conditions EQ, GT and GE will be false.

Instruction Formats

Mnemonic	op	t	b1	w2
CMPB	0x0e	0	literal byte	
CMPB	0x0e	tag	literal byte	address offset

3.10 CMPN (CoMPare N-byte blocks)

Operation

$$t=0 \quad \{DT[-2*b1], b1\} - \{DT[-b1], b1\};$$

$$t=tag \quad \{t[w2], b1\} - \{DT[-b1], b1\};$$

Condition Code

				C	V	N	Z
-	-	-	-	0	0	0	0

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction subtracts the block of length b1 that starts at DT[-b1] from either:

- the block of length b1 that starts at DT[-2*b1], or
- the block of length b1 that starts at t[w2].

Each block is treated as an unsigned integer of length b1 bytes. No data is modified, but the CCR is set according to the result of the subtraction. See **CMPB** for examples.

Instruction Formats

Mnemonic	op	t	b1	w2
CMPN	0x1a	0	block length	
CMPN	0x1a	tag	block length	address offset

3.11 CMPW (CoMPare literal with Word)

Operation

```
t=0      {DT[-2], 2} - w1;
t=tag    {t[w3], 2} - w1;
```

Condition Code

				C	V	N	Z
-	-	-	-	0	0	0	0

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction subtracts the literal word w1 from either:

- the word that starts at DT[-2], or
- the word that starts at t[w3].

No data is modified, but the CCR is set according to the result of the subtraction. See **CMPB** for examples.

Instruction Formats

Mnemonic	op	t	w1	w3
CMPW	0x12	0	literal word	
CMPW	0x12	tag	literal word	address offset

3.12 DECN (DECrement N-byte block)

Operation

```
t=0      {DT[-b1], b1} -= 1;
t=tag    {t[w2], b1}  -= 1;
```

Condition Code

	C	V	N	Z
-	-	-	-	↕

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the result is zero, cleared otherwise

Description

This instruction decrements by unity either:

- the block of length b1 that starts at DT[-b1], or
- the block of length b1 that starts at t[w2].

The block is treated as an unsigned integer of length b1. This instruction does not modify the C flag of the CCR. When decrementing a byte or word, two methods are possible:

- SUBB or SUBW, which modify the C and Z flags
- DECN, which modifies the Z flag but leaves the C flag unchanged.

Instruction Formats

Mnemonic	op	t	b1	w2
DECN	0x18	0	block length	
DECN	0x18	tag	block length	address offset

3.13 INCN (INCRement N-byte block)

Operation

```

t=0
    {DT[-b1], b1} += 1;
t=tag
    {t[w2], b1} += 1;
    
```

Condition Code

	C	V	N	Z
-	-	-	-	↕

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the result is zero, cleared otherwise

Description

This instruction increments by unity either:

- the block of length b1 that starts at DT[-b1], or
- the block of length b1 that starts at t[w2].

The block is treated as an unsigned integer of length b1. This instruction does not modify the C flag of the CCR. When incrementing a byte or word, two methods are possible:

- ADDB or ADDW, which modify the C and Z flags
- INCN, which modifies the Z flag but leaves the C flag unchanged.

Instruction Formats

Mnemonic	op	t	b1	w2
INCN	0x17	0	block length	
INCN	0x17	tag	block length	address offset

3.14 INDEX (INDEX array)

Operation

```
t=tag
DT' += 2;
{DT[0], 2} = b1 * {DT[-1], 1} +
t[w2];
```

Condition Code

				C	V	N	Z
-	-	-	-	Ô	Ô	Ô	Ô

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction performs a multiplication followed by an addition:

- It multiplies the byte at DT[-1] by the literal byte b1, producing a word result. The multiply is unsigned.

- It adds the word result to the segment address corresponding to $t[w2]$, and pushes the sum on to the stack. The CCR is set according to the result of this addition.

Note: this instruction succeeds regardless of the value of $w2$. There is no requirement that $t[w2]$ be a valid segment address.

The primary purpose of this instruction is to index arrays. If the start of the array is at $t[w2]$, the record length is $b1$ bytes, and the desired record number is at $DT[-1]$, then INDEX pushes the segment address of the desired record on to the stack.

The segment address of $SB[0]$ is zero. Therefore, by specifying a tag of SB and $w2=0$, this instruction may be used to perform multiplication.

Instruction Formats

Mnemonic	op	t	b1	w2
INDEX	0x0c	tag	literal byte	address offset

3.15 JUMP (JUMP to code address)

Operation

```

if ( t > 0 )
{
    switch (t)
    {
        case EQ : temp = Z;      break;
        case LT : temp = C;      break;
        case LE : temp = (C || Z); break;
        case GT : temp = !(C || Z); break;
        case GE : temp = !C;     break;
        case NE : temp = !Z;     break;
        case A  : temp = TRUE;   break;
    }
    if ( temp )
    {
        CP' = w1;
    }
}
else // t == 0
{
    DT' = DT - 2;
    CP' = {DT[-2], 2};
}

```

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

If t is non-zero, it specifies a condition as follows:

t	Mnemonic	Meaning
1	EQ	Equal
2	LT	Less Than
3	LE	Less than or Equal
4	GT	Greater Than
5	GE	Greater than or Equal
6	NE	Not Equal
7	A	Always

If the condition is true, control is directed to the code address specified as w1.

If t is zero, two bytes are popped from the stack. Control is directed to the code address specified by the two popped bytes.

Note: the conditions based on the state of the Carry flag, LT, LE, GT, and GE, apply to unsigned comparisons. **MULTOS** does not directly support conditional program flow based on signed comparisons. If needed, the application developer could implement it based on the condition of the N and V flags on platforms that support signed arithmetic. (The *Load CCR* primitive, described in section 4.12 allows access to the N and V flags.)

Instruction Formats

Mnemonic	op	t	w1	
JUMP	0x02	0		
JUMP	0x02	1 - 7	code address	

3.16 LOAD (LOAD data to stack)

Operation

```

t=0
    DT' = DT + b1;
    {DT[0], b1} = {DT[-b1], b1};
t>tag
    DT' = DT + b1;
    {DT[0], b1} = {t[w2], b1};
    
```

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction pushes to stack top either:

- the block of length b1 that starts at DT[-b1] (in other words, it duplicates the top b1 bytes of Dynamic), or
- the block at address t[w2].

Instruction Formats

Mnemonic	op	t	b1	w2
LOAD	0x07	0	block length	
LOAD	0x07	tag	block length	address offset

3.17 LOADA (LOAD Address)

Operation

t=tag
 $DT' = DT + 2;$
 $\{DT[0], 2\} = t[w1];$

Condition Code

			C	V	N	Z
-	-	-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction pushes the segment address equivalent to tagged address t[w1] onto the stack.

Note: this instruction succeeds regardless of the value of w1. There is no requirement that t[w1] be a valid segment address.

The segment address of SB[w1] is w1. Therefore, by specifying t=SB, this instruction can be used to load the literal word w1 to the stack. However, the use of STACK is preferred for this purpose.

Instruction Formats

Mnemonic	op	t	w1
LOADA	0x0b	tag	address offset

3.18 LOADI (LOAD Indirect)

Operation

```

t=0
    DT' = DT + b1;
    {DT[0], b1} = {{DT[-2], 2}, b1};
t=tag
    DT' = DT + b1;
    {DT[0], b1} = {{t[w2], 2}, b1};

```

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction pushes a block on to the stack, where the segment address of the block is given by either:

- the word that starts at DT[-2], or
- the word that starts at t[w2].

Instruction Formats

Mnemonic	op	t	b1	w2
LOADI	0x09	0	block length	
LOADI	0x09	tag	block length	address offset

3.19 NOTN (bitwise NOT N-byte block)

Operation

t=0 {DT[-b1], b1} = ~{DT[-b1], b1};
t=tag {t[w2], b1} = ~{t[w2], b1};

Condition Code

	C	V	N	Z
-	-	-	-	0
			↕	↕

- C Previous value remains unchanged
- V Cleared
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction bitwise-inverts either:

- the block of length b1 that starts at DT[-b1], or
- the block of length b1 that starts at t[w2].

Instruction Formats

Mnemonic	op	t	b1	w2
NOTN	0x19	0	block length	
NOTN	0x19	tag	block length	address offset

3.20 ORN (bitwise OR N-byte blocks)

Operation

$t=0$ $\{DT[-2*b1], b1\} \mid= \{DT[-b1], b1\};$
 $t=tag$ $\{t[w2], b1\} \mid= \{DT[-b1], b1\};$

Condition Code

					C	V	N	Z
-	-	-	-	-	0	⇕	⇕	⇕

- C Previous value remains unchanged
- V Cleared
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the block of length b1 that starts at DT[-2*b1], or
- the block of length b1 that starts at t[w2]

by performing a bitwise OR operation between it and the block of length b1 which starts at DT[-b1].

Instruction Formats

Mnemonic	op	t	b1	w2
ORN	0x1e	0	block length	
ORN	0x1e	tag	block length	address offset

3.21 PRIMRET (call PRIMitive or RETurn)

Operation

```

t=0
    call primitive b1 in set zero
t=1
    call primitive b1 in set one,
        with argument b2
t=2
    call primitive b1 in set two,
        with arguments b2 and b3
t=3
    call primitive b1 in set three,
        with arguments b2, b3 and b4
t=4
    CP' = {LB[-2], 2};
    LB' = {LB[-4], 2};
    DT' = LB - 4;
t=5
    CP' = {LB[-2], 2};
    LB' = {LB[-4], 2};
    DT' = LB - b1 - 4;
t=6
    CP' = {LB[-2], 2};
    LB' = {LB[-4], 2};
    DT' = LB + b1 - 4;
    {DT'[-b1], b1} = {DT[-b1], b1};
t=7
    CP' = {LB[-2], 2};
    LB' = {LB[-4], 2};
    DT' = LB - b1 + b2 - 4;
    {DT'[-b2], b2} = {DT[-b2], b2};

```

Section 3

Condition Code

When t=0, 1, 2 or 3, the CCR may be altered by the primitive.

When t=4, 5, 6 or 7:

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

Depending on t, this instruction does one of the following:

- t=0 : calls a primitive with no arguments
- t=1 : calls a primitive with one byte of arguments
- t=2 : calls a primitive with two bytes of arguments
- t=3 : calls a primitive with three bytes of arguments
- t=4 : returns from a CALLED routine
- t=5 : returns from a CALLED routine, discarding b1 bytes of **in** parameters
- t=6 : returns from a CALLED routine, returning b1 bytes of **out** parameters
- t=7 : returns from a CALLED routine, discarding b1 bytes of **in** parameters and returning b2 bytes of **out** parameters

In the cases t=4 to t=7, it restores CP and LB from the link data saved on the stack. In these cases, an abend results if the stack is not in a valid state from which to return. In particular:

- there must be at least four bytes (the link data) plus **in** parameter bytes below LB on the stack

- there must be at least **out** parameter bytes between LB and DT.

See section 1.6 for further details of the call/return mechanism.

Instruction Formats

Mnemonic	op	t	b1	b2	b3	b4
PRIMRET	0x05	0	primitive	argument		
PRIMRET	0x05	1	primitive			
PRIMRET	0x05	2	primitive	arguments		
PRIMRET	0x05	3	primitive	arguments		
PRIMRET	0x05	4				
PRIMRET	0x05	5	in bytes			
PRIMRET	0x05	6	out bytes			
PRIMRET	0x05	7	in bytes	out bytes		

3.22 SETB (SET Byte to literal)

Operation

```

t=0      {DT[-1], 1} = b1;
t=tag    {t[w2], 1} = b1;
    
```

Condition Code

C V N Z

-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---

Section 3

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction modifies either:

- _ the byte at DT[-1], or
- _ the byte at t[w2]

by overwriting it with the literal byte b1.

Instruction Formats

Mnemonic	op	t	b1	w2
SETB	0x0d	0	literal byte	
SETB	0x0d	tag	literal byte	address offset

3.23 SETW (SET Word to literal)

Operation

t=0
 $\{DT[-2], 2\} = w1;$
 t=tag
 $\{t[w3], 2\} = w1;$

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction modifies either:

- _ the word that starts at DT[-2], or
- _ the word that starts at t[w3]

by overwriting it with the literal word w1.

Instruction Formats

Mnemonic	op	t	w1	w3
SETW	0x11	0	literal word	
SETW	0x11	tag	literal word	address offset

3.24 STACK (push or pop STACK)

Operation

- t=0
 - $DT' = DT + b1;$
 - $\{DT[0], b1\} = 0;$
- t=1
 - $DT' = DT + 1;$
 - $\{DT[0], 1\} = b1;$
- t=2
 - $DT' = DT + 2;$
 - $\{DT[0], 2\} = w1;$
- t=4
 - $DT' = DT - b1;$
- t=5
 - $DT' = DT - 1;$
- t=6
 - $DT' = DT - 2;$

Section 3

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

Depending on t, this instruction does one of the following:

- t=0 : pushes an initialised (to zero) block of length b1 onto the stack.
- t=1 : pushes the literal byte b1 onto the stack.
- t=2 : pushes the literal word w1 onto the stack.
- t=4 : pops b1 bytes from the stack.
- t=5 : pops a byte from the stack.
- t=6 : pops a word from the stack.

Instruction Formats

Mnemonic	op	t	b1	
			w1	
STACK	0x04	0	block length	
STACK	0x04	1	literal byte	
STACK	0x04	2	literal word	
STACK	0x04	4	block length	
STACK	0x04	5		
STACK	0x04	6		

3.25 STORE (STORE data from stack)

Operation

```

t=0
    DT' = DT - b1;
    {DT[-2*b1], b1} = {DT[-b1], b1};
t=tag
    DT' = DT - b1;
    {t[w2], b1} = {DT[-b1], b1};

```

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction pops the block of length b1 that starts at DT[-b1] and stores it to either:

- the block of length b1 that starts at DT[-2*b1], or
- the block at address t[w2].

Instruction Formats

Mnemonic	op	t	b1	w2
STORE	0x08	0	block length	
STORE	0x08	tag	block length	address offset

3.26 STOREI (STORE Indirect)

Operation

```

t=0
    DT' = DT - b1;
    {{DT[-2-b1], 2}, b1} = {DT[-b1], b1};
t=tag
    DT' = DT - b1;
    {{t[w2], 2}, b1} = {DT[-b1], b1};

```

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction pops a block from the stack, and stores it to the location whose segment address is given by either:

- _ the word that starts at DT[-2-b1], or
- _ the word that starts at t[w2].

Instruction Formats

Mnemonic	op	t	b1	w2
STOREI	0x0a	0	block length	
STOREI	0x0a	tag	block length	address offset

3.27 SUBB (SUBtract literal from Byte)

Operation

```
t=0
    {DT[-1], 1} -= b1;
t=tag
    {t[w2], 1} -= b1;
```

Condition Code

				C	V	N	Z
-	-	-	-	Ô	Ô	Ô	Ô

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the byte at DT[-1], or
- the byte at t[w2]

by subtracting the literal byte b1.

Instruction Formats

Mnemonic	op	t	b1	w2
SUBB	0x10	0	block length	
SUBB	0x10	tag	block length	address offset

3.28 SUBN (SUBtract N-byte blocks)

Operation

```

t=0
    {DT[-2*b1], b1} -= {DT[-b1], b1};
t=tag
    {t[w2], b1} -= {DT[-b1], b1};

```

Condition Code

				C	V	N	Z
-	-	-	-	0	0	0	0

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the block of length b1 that starts at DT[-2*b1], or
- the block of length b1 that starts at t[w2]

by subtracting the block of length b1 which starts at DT[-b1].
Each block is treated as an unsigned integer of length b1 bytes.

Instruction Formats

Mnemonic	op	t	b1	w2
SUBN	0x1c	0	block length	
SUBN	0x1c	tag	block length	address offset

3.29 SUBW (SUBtract literal from Word)

Operation

```
t=0      {DT[-2], 2} -= w1;
t=tag    {t[w3], 2} -= w1;
```

Condition Code

				C	V	N	Z
-	-	-	-	0	0	0	0

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the word that starts at DT[-2], or
- the word that starts at t[w3]

by subtracting the literal word w1.

Instruction Formats

Mnemonic	op	t	w1	w3
SUBW	0x14	0	literal word	
SUBW	0x14	tag	literal word	address offset

3.30 SYSTEM (various actions)

Operation

```

t=0          (none)
t=1          {PT[cSW1], 2} = w1;
t=2          {PT[cLa], 2} = w1;
t=3          {PT[cSW1], 2} = w1;
             {PT[cLa], 2} = w3;
t=4          exit;
t=5          {PT[cSW1], 2} = w1;
             exit;
t=6          {PT[cLa], 2} = w1;
             exit;
t=7          {PT[cSW1], 2} = w1;
             {PT[cLa], 2} = w3;
             exit;

```

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Description

This instruction performs different functions depending on t.

- t function
- 0 No operation
- 1 The communications interface status bytes are set to w1.
- 2 The communications interface actual response length is set to w1.
- 3 The communications interface status bytes are set to w1; the actual response length is set to w3.
- 4 Control is returned to **MULTOS**.
- 5 The communications interface status bytes are set to w1; control is returned to **MULTOS**.
- 6 The communications interface actual response length is set to w1; control is returned to the **MULTOS**.
- 7 The communications interface status bytes are set to w1; the actual response length is set to w3; control is returned to the MULTOS.

Instruction Formats

Mnemonic	op	t	w1	w3
SYSTEM	0x00	0		
SYSTEM	0x00	1	SW1, SW2	
SYSTEM	0x00	2	La	
SYSTEM	0x00	3	SW1, SW2	La
SYSTEM	0x00	4		
SYSTEM	0x00	5	SW1, SW2	
SYSTEM	0x00	6	La	
SYSTEM	0x00	7	SW1, SW2	La

3.31 TESTN (TEST N-byte block against 0)

Operation

```
t=0      {DT[-b1], b1};
t=tag    {t[w2], b1};
```

Condition Code

					C	V	N	Z
-	-	-	-	-	0	⇕	⇕	⇕

- C Previous value remains unchanged
- V Cleared
- N Set if the multi-byte block is negative, cleared otherwise
- Z Set if the multi-byte block is zero, cleared otherwise

Description

This instruction compares with zero either:

- the block of length b1 that starts at DT[-b1], or
- the block of length b1 that starts at t[w2].

No data is modified, but the CCR is set.

Instruction Formats

Mnemonic	op	t	b1	w2
TESTN	0x16	0	block length	
TESTN	0x16	tag	block length	address offset

3.32 XORN (bitwise eXclusive OR N-byte blocks)

Operation

```
t=0      {DT[-2*b1], b1} ^= {DT[-b1], b1};
t=tag    {t[w2], b1}   ^= {DT[-b1], b1};
```

Condition Code

						C	V	N	Z
-	-	-	-	-	0	↕	↕	↕	↕

- C Previous value remains unchanged
- V Cleared
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Description

This instruction modifies either:

- the block of length b1 that starts at DT[-2*b1], or
- the block of length b1 that starts at t[w2]

by performing a bitwise XOR operation between it and the block of length b1 which starts at DT[-b1].

Instruction Formats

Mnemonic	op	t	b1	w2
XORN	0x1f	0	block length	
XORN	0x1f	tag	block length	address offset

**Section
3**

4. Mandatory Primitives

Primitives extend the functionality of the instruction set. This section describes the primitive functions available to applications on all **MULTOS** implementations. The primitives are listed in alphabetic order.

4.1 Introduction

Each primitive is described under the headings *Primitive set and number*, *Arguments*, *In Parameters*, *Out Parameters*, *Effect on DT*, *Condition Code*, *Side effects*, and *Description*. The following sections describe what information is found under each heading.

Primitive set and number

There are four sets of primitives, classified by number of arguments, and referred to as set zero through to set three. A primitive of set **n** must be called with the *t* field of the PRIMRET instruction set to **n**, and takes **n** bytes of arguments.

Within each set, primitives are numbered from 0 through to 0xff. Primitives with numbers less than 0x80 are provided by all MULTOS implementations. Other primitives may or may not be supported. The Query primitive, which is mandatory, allows an application to test for the presence of a primitive. An attempt to call an unsupported primitive causes the application to abend.

Arguments

A description of the arguments (if any) in bytes *b2*, *b3* and *b4* of the instruction. The *b1* argument of the instruction always contains the primitive number. As in section 3, the notation *w2* denotes $b2 \times 256 + b3$ and *w3* denotes $b3 \times 256 + b4$.

In Parameters

A description of data to be placed in Dynamic before calling the primitive.

Out Parameters

A description of data left in Dynamic by the primitive upon completion. The addresses are shown relative to the value of DT after the primitive has executed. Where the primitive changes DT, the notation DT' is used to emphasize that the value is the new value of DT after any bytes are pushed or popped as described in *Effect on DT*.

Effect on DT

A statement of how DT, the stack pointer, is increased or decreased by the primitive.

Condition Code

A statement of how the condition codes of CCR are affected by the primitive.

Side effects

A description of other data (in Static, Dynamic or Public) changed by the primitive.

Description

A description of the operation of the primitive.

4.2 Bit Manipulate Byte

Primitive set and number

Set two, number 0x01

Arguments

b2: a bitmap as follows:

- bit 7 (most significant): '1' indicates that the byte at DT[-1] should be modified, '0' that it should not.
- bits 6, 5, 4, 3 and 2: always '0' (Otherwise, the effect of the primitive is undefined.)

- bits 1 and 0: '11' for AND, '10' for OR, '01' for EQV, '00' for XOR

b3: literal byte

In Parameters

{DT[-1], 1} = byte to be tested or modified

Out Parameters

{DT[-1], 1} = modified or unmodified byte

Effect on DT

None

Condition Code

					C	V	N	Z
-	-	-	-	-	0	Ù	Ù	Ù

- C Previous value remains unchanged
- V Cleared
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive allows an application to perform a bitwise AND, OR, EQV or XOR between a byte at DT[-1] and a literal byte. The byte at DT[-1] may either be modified or may be left unchanged, according to the state of bit 7 of argument b2; in either case, the CCR reflects the result of the operation.

The bitwise operations are defined as follows:

0 AND 0 is 0

0 AND 1 is 0

1 AND 0 is 0

1 AND 1 is 1

0 OR 0 is 0

0 OR 1 is 1

1 OR 0 is 1

1 OR 1 is 1

0 EQV 0 is 1

0 EQV 1 is 0

1 EQV 0 is 0

1 EQV 1 is 1

0 XOR 0 is 0

0 XOR 1 is 1

1 XOR 0 is 1

1 XOR 1 is 0

4.3 Bit Manipulate Word

Primitive set and number

Set three, number 0x01

Arguments

b2: a bitmap as follows:

- bit 7 (most significant): '1' indicates that the byte at DT[-1] should be modified, '0' that it should not.
- bits 6, 5, 4, 3 and 2: always '0' (Otherwise, the effect of the primitive is undefined.)
- bits 1 and 0: '11' for AND, '10' for OR, '01' for EQV, '00' for XOR

w3: literal word

In Parameters

{DT[-2], 2} = word to be tested or modified

Out Parameters

{DT[-2], 2} = modified or unmodified word

Effect on DT

None

Condition Code

					C	V	N	Z
-	-	-	-	-	0	Ù	Ù	Ù

C Previous value remains unchanged

V Cleared

N Set if the result is negative, cleared otherwise

Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive allows an application to perform a bitwise AND, OR, EQV or XOR between a word at DT[-2] and a literal word. The word at DT[-2] may either be modified or may be left unchanged, according to the state of bit 7 of argument b2; in either case, the CCR reflects the result of the operation.

4.4 Check Case

Primitive set and number

Set zero, number 0x01

Arguments

None

In Parameters

{DT[-1], 1} = 1, 2, 3 or 4, to indicate the expected ISO case

Out Parameters

None

Effect on DT

1 byte is popped.

Condition Code

					C	V	N	Z
-	-	-	-	-	-	-	-	Ù

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the specified case is consistent with the data in Public, clear if it is not

Side effects

Public may change:

- Command data may appear at PB[0] onwards.
- Lc, Le and the Protocol flags in Public may change.

Description

All applications should use this primitive on recognizing a command header, and should process the command only if the Z flag is set when the primitive returns. See section 2.2 for a complete description of case checking.

Note: if the byte at the top of the stack is not 1, 2, 3 or 4 (the defined cases), the primitive clears the Z flag, indicating an inconsistent case.

4.5 DivideN

Primitive set and number

Set one, number 0x08

Arguments

b2: number of bytes in each operand

In Parameters

{DT[-2*b2], b2} = numerator

{DT[-b2], b2} = denominator

Out Parameters

{DT[-2*b2], b2} = quotient

{DT[-b2], b2} = remainder

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	Ù

- C Set if the denominator is zero, cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the quotient is zero, cleared if the quotient is non-zero

Side effects

None

Description

This primitive performs unsigned division of two numbers. The numerator, denominator, quotient and remainder, are each of length $b2$ bytes.

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 (in other words, it is set to indicate whether the quotient is zero).
- The quotient and remainder replace the numerator and denominator at the top of Dynamic.

Like the unary and binary instructions, this primitive may use temporary space. The amount of temporary space used never exceeds $2*b2$ bytes.

4.6 Get DIR File Record

Primitive set and number

Set one, number 0x09

Arguments

b2: maximum length of data to be retrieved

In Parameters

{DT[-3], 2} = segment address to which the DIR file record should be copied

{DT[-1], 1} = record number of the desired record. Note: DIR file records are numbered from one, not from zero.

Out Parameters

{DT'[-4], 2} = segment address to which the DIR file record has been copied, unchanged

{DT'[-2], 1} = record number of the desired record, unchanged

{DT'[-1], 1} = length of data retrieved

Effect on DT

1 byte is pushed

Condition Code

	C	V	N	Z
-	-	-	-	Ù
-	-	-	Ù	-
-	-	Ù	-	-

- C Set if data retrieved was less than requested (including zero), cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the specified record does not exist, cleared otherwise

Side effects

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. The number of bytes copied is returned on the stack.

Description

This primitive allows an application to read Directory File (DIR) file records.

The effect of the primitive is undefined if b2 is zero. Applications should not set b2 to zero when testing for the presence of a particular record; they should set b2 to one to guarantee correct operation of the primitive.

4.7 Get File Control Information

Primitive set and number

Set one, number 0x0a

Arguments

b2: maximum length of data to be retrieved

In Parameters

{DT[-3], 2} = segment address to which the file control information should be copied

{DT[-1], 1} = record number of the desired record. Note that files are numbered from one, not from zero.

Out Parameters

{DT'[-4], 2} = segment address to which the file control information has been copied, unchanged

{DT'[-2], 1} = record number of the desired record, unchanged

{DT'[-1], 1} = length of data retrieved

Effect on DT

1 byte is pushed

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	Ù

- C Set if data retrieved was less than requested (including zero), cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the specified record does not exist, cleared otherwise

Side effects

The file control information 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 information. The number of bytes copied is returned on the stack.

Description

This primitive allows an application to read file control information corresponding to an application.

The effect of the primitive is undefined if b2 is zero. Applications should not set b2 to zero when testing for the presence of a particular record; they should set b2 to one to guarantee correct operation of the primitive.

4.8 Get Manufacturer Data

Primitive set and number

Set one, number 0x0b

Arguments

b2: maximum length of data to be retrieved

In Parameters

{DT[-2], 2} = segment address to which the manufacturer data should be copied

Out Parameters

{DT'[-1], 1} = length of data retrieved

Effect on DT

1 byte is popped

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	-

- C Set if data retrieved was less than requested, cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

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 manufacturer data.

The number of bytes copied is returned on the stack. This number may be placed on the stack either before or after the manufacturer data is copied to the specified segment address, so the exact effect of the primitive is undefined if the destination area includes the top of the stack.

Description

This primitive allows an application to obtain the manufacturer data of the MCD. Manufacturer data is specified in the **MULTOS** Design Standard, IFD-**MULTOS** Interface Specification.

The effect of the primitive is undefined if b2 is zero.

4.9 Get MULTOS Data**Primitive set and number**

Set one, number 0x0c

Arguments

b2: maximum length of data to be retrieved

In Parameters

{DT[-2], 2} = segment address to which the MULTOS data should be copied

Out Parameters

{DT'[-1], 1} = length of data retrieved

Effect on DT

1 byte is popped

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	-

- C Set if data retrieved was less than requested, cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

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 **MULTOS** data.

The number of bytes copied is returned on the stack. This number may be placed on the stack either before or after the **MULTOS** data is copied to the specified segment address, so the exact effect of the primitive is undefined if the destination area includes the top of the stack.

Description

This primitive allows an application to obtain the **MULTOS** data of the MCD. **MULTOS** data is specified in the **MULTOS** Design Standard, IFD-**MULTOS** Interface Specification.

The effect of the primitive is undefined if b2 is zero.

4.10 Get Memory Reliability

Primitive set and number

Set zero, number 0x09

Arguments

None

In Parameters

None

Out Parameters

None

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	Ù

- C Set if memory is unreliable, cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if memory is marginally reliable, cleared otherwise

Side effects

None

Description

This primitive allows an application to enquire whether the memory hardware is reliable. There are three possibilities:

- Memory is reliable: C and Z are both cleared. Applications are recommended to proceed normally.
- Memory is marginally reliable: C is cleared and Z is set. Applications should issue a warning that the life of the

card may be nearing its end, and/or to refuse to start new business transactions.

- Memory is unreliable: C is set and Z is cleared. Applications should issue error messages or disable themselves entirely.

All **MULTOS** implementations support this primitive. However, not all implementations monitor memory reliability. Implementations that do not monitor reliability always return the response, “memory is reliable.”

The non-volatile memory technology used in smart cards, EEPROM (Electrically Erasable Programmable Read-Only Memory), has finite life, and reliability may vary slightly from manufacturer to manufacturer. The Get Memory Reliability primitive adds an additional layer of safety for end-of-card handling on cards that monitor memory reliability. Detecting that memory is only marginally reliable or unreliable allows applications to handle such situations elegantly. The actual mechanism used to monitor memory reliability varies from implementation to implementation.

4.11 Get Purse Type

Primitive set and number

Set one, number 0x0d

Arguments

b2: maximum length of data to be retrieved

In Parameters

{DT[-2], 2} = segment address to which the Mondex Purse Type should be copied

Out Parameters

{DT'[-1], 1} = length of data retrieved

Effect on DT

1 byte is popped

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	-

- C Set if data retrieved was less than requested, cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The Mondex Purse Type 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 Mondex Purse Type.

The number of bytes copied is returned on the stack. This number may be placed on the stack either before or after the Purse Type is copied to the specified segment address, so the exact effect of the primitive is undefined if the destination area includes the top of the stack.

Description

This primitive allows an application to obtain the Mondex Purse Type of the MCD. This information is intended only for Mondex applications; for other applications it is not meaningful. Mondex Purse Type is specified in the MULTOS Design Standard, IFD-MULTOS Interface Specification.

Mondex Purse Type is currently one byte. However, the primitive allows the flexibility to request b2 bytes of data, should Mondex Purse Type ever be redefined.

The effect of the primitive is undefined if b2 is zero.

4.12 Load CCR

Primitive set and number

Set zero, number 0x05

Arguments

None

In Parameters

None

Out Parameters

{DT'[-1], 1} = CCR

Effect on DT

1 byte is pushed.

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

None

Description

This primitive pushes one byte to the stack, containing the same bit settings as the CCR.

4.13 Lookup

Primitive set and number

Set zero, number 0x0a

Arguments

None

In Parameters

{DT[-3], 2} = byte value to find in the lookup array

{DT[-1], 1} = segment address of lookup array

Out Parameters

{DT'[-1], 1} = offset of the byte in the lookup array

Effect on DT

2 bytes are popped.

Condition Code

					C	V	N	Z
-	-	-	-	-	-	-	-	Ù

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the byte is found in the array, cleared otherwise

Side effects

None

Description

This primitive searches a lookup array for a specified byte value. The lookup array must consist of:

- _ one byte giving the number of entries in the remainder of the array
- _ the bytes to be compared, in ascending numerical order.

For example, if the byte values to be compared are 0x24, 0x89 and 0x9f, then the lookup array must be composed of the four bytes 0x03, 0x24, 0x89, 0x9f. {DT[-3], 2} must be the segment address of the value 0x03. The primitive then returns the value 0x00 if {DT[-1], 1} is 0x24, 0x01 if it is 0x89 and 0x02 if it is 0x9f. In any of these cases, the primitive sets the Z flag. If {DT[-1], 1} does not match any value in the lookup array, the primitive clears the Z flag.

4.14 Memory Compare

Primitive set and number

Set zero, number 0x0b

Arguments

None

In Parameters

{DT[-6], 2} = number of bytes to compare

{DT[-4], 2} = segment address of first operand

{DT[-2], 2} = segment address of second operand

Out Parameters

None

Effect on DT

6 bytes are popped.

Condition Code

				C	V	N	Z
-	-	-	-	Ù	Ù	Ù	Ù

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive is used to compare data between two locations. The second operand (whose address is at stack top) is subtracted from the first. No data is modified, but the CCR is set according to the result of the subtraction.

Where the number of bytes to be compared is fixed, and is no more than 255, then the primitive Memory Compare Fixed Length is likely to be more efficient.

4.15 Memory Compare Fixed Length

Primitive set and number

Set one, number 0x0f

Arguments

b2: number of bytes of data to be compared

In Parameters

{DT[-4], 2} = segment address of first operand

{DT[-2], 2} = segment address of second operand

Out Parameters

None

Effect on DT

4 bytes are popped.

Condition Code

				C	V	N	Z
-	-	-	-	Ù	Ù	Ù	Ù

- C Set if a carry occurs, cleared otherwise
- V Set if a signed overflow occurs, cleared otherwise
- N Set if the result is negative, cleared otherwise
- Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive is used to compare data between two locations. The second operand (whose address is at stack top) is subtracted from the first. No data is modified, but the CCR is set according to the result of the subtraction.

The primitive works correctly even if the blocks overlap. The same effect may be achieved by LOAD followed by CMPN, but this primitive has the advantage of not requiring stack space to hold the data.

4.16 Memory Copy

Primitive set and number

Set zero, number 0x0c

Arguments

None

In Parameters

{DT[-6], 2} = number of bytes to copy

{DT[-4], 2} = segment address of destination

{DT[-2], 2} = segment address of source

Out Parameters

None

Effect on DT

6 bytes are popped.

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The memory addressed by the segment address at {DT[-4, 2]} is updated.

Description

This primitive is used to copy data from one location to another. It works correctly even if the blocks overlap. Where the number of bytes to be copied is fixed, and is no more than 255, then the primitive Memory Copy Fixed Length is likely to be more efficient.

4.17 Memory Copy Fixed Length

Primitive set and number

Set one, number 0x0e

Arguments

b2: number of bytes of data to be copied

In Parameters

{DT[-4], 2} = segment address of destination

{DT[-2], 2} = segment address of source

Out Parameters

None

Effect on DT

4 bytes are popped.

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The memory addressed by the segment address at $\{DT[-4, 2]\}$ is updated.

Description

This primitive is used to copy data from one location to another. It works correctly even if the blocks overlap. The same effect may be achieved by LOAD followed by STORE, but this primitive has the advantage of not requiring stack space to hold the data.

4.18 MultiplyN**Primitive set and number**

Set one, number 0x10

Arguments

b2: number of bytes in each operand

In Parameters

$\{DT[-2*b2], b2\}$ = operand1

$\{DT[-b2], b2\}$ = operand2

Out Parameters

$\{DT[-2*b2], 2*b2\}$ = product

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	-	-	-	Ù

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive performs unsigned multiplication of two numbers. Both operands are of length b2 bytes; their product is of length 2*b2 bytes. The result replaces the two operands at the top of Dynamic.

Like the unary and binary instructions, this primitive may use temporary space. The amount of temporary space used never exceeds more than 2*b2 bytes.

4.19 Query0, Query1, Query2, Query3

Primitive set and number

Set one, number 0x00 to number 0x03

Arguments

b2: primitive number whose existence is being queried.

In Parameters

None

Out Parameters

None

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	-	-	-	Ù

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the desired primitive exists, cleared otherwise

Side effects

None

Description

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

4.20 Reset WWT

Primitive set and number

Set zero, number 0x02

Arguments

None

In Parameters

None

Out Parameters

None

Effect on DT

None

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

None

Description

This primitive causes a message to be sent to the IFD, to inform it that more time is required for processing to complete. The nature of the message is protocol-dependent; in T=0, for example, a NULL byte is sent. The “WWT” in the primitive’s name refers to the T=0 term “Work Waiting Time”.

Applications should call this primitive from time to time during long computations.

Note: computationally-intensive primitives automatically request more time from the IFD.

4.21 Set ATR File Record

Primitive set and number

Set zero, number 0x07

Arguments

None

In Parameters

{DT[-2], 2} = segment address of the length of the Answer To Reset (ATR) file record. The length is expressed in one byte, and must be followed immediately by the data which is to form the record. The length may be zero; this will erase any existing ATR file record.

Out Parameters

{DT'[-1], 1} = length of data written

Effect on DT

1 byte is popped.

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	Ù

- C Set if the amount of data written is less than requested
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if no data is written, cleared otherwise

Side effects

The ATR file data is copied from the byte after the segment address specified by the application.

No more than the specified number of bytes are written; the actual number written is returned on the stack. This number may be placed on the stack either before or after the ATR file data is copied from the specified segment address, so the exact effect of the primitive is undefined if the source area includes the top of the stack.

Description

This primitive allows an application to write data to the ATR file.

4.22 Set ATR Historical Characters

Primitive set and number

Set zero, number 0x08

Arguments

None

In Parameters

{DT[-2], 2} = segment address of the length of the ATR historical characters. The length is expressed in one byte, and must be followed immediately by the data which is to form the historical characters. The length may be zero; this erases any existing ATR historical characters.

Out Parameters

{DT'[-1], 1} = length of data written

Effect on DT

1 byte is popped.

Condition Code

				C	V	N	Z
-	-	-	-	Ù	-	-	Ù

- C Set if the amount of data written is less than requested
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if no data is written, cleared otherwise

Side effects

The ATR historical characters data is copied from the byte after the segment address specified by the application.

No more than the specified number of bytes are written; the actual number written will be returned on the stack. This number may be placed on the stack either before or after the ATR data is copied from the specified segment address, so the exact effect of the primitive is undefined if the source area includes the top of the stack.

Description

This primitive allows an application to specify data that may be added to the ATR of the MCD. This request is accepted even if the right to add data has already been taken by another application. Therefore, in practice, the Z flag is only set in the event that the application specifies that it wishes to write zero bytes.

4.23 Shift Left**Primitive set and number**

Set two, number 0x02

Arguments

b2: number of bytes of data to be shifted

b3: number of bits by which data is to be shifted

Section 4

In Parameters

{DT[-b2], b2} = b2 bytes to be shifted

Out Parameters

{DT[-b2], b2} = b2 bytes which have been shifted

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	Ù	0	Ù	Ù

C Set if the last bit shifted out is a one, cleared otherwise

V Cleared

N Set if the result is negative, cleared otherwise

Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive bit-shifts data leftwards, filling the least significant bits with zeroes (that is, it multiplies by powers of two).

The effect of the primitive is undefined if any of the following is true:

- b2 is zero
- b3 is zero
- $b3 \geq 8 * b2$

4.24 Shift Right

Primitive set and number

Set two, number 0x03

Arguments

b2: number of bytes of data to be shifted

b3: number of bits by which data is to be shifted

In Parameters

{DT[-b2], b2} = b2 bytes to be shifted

Out Parameters

{DT[-b2], b2} = b2 bytes which have been shifted

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	Ù	0	Ù	Ù

- C Set if the last bit shifted out is a one, cleared otherwise
- V Cleared
- N Set if the result is negative, cleared otherwise (in practice, always cleared)
- Z Set if the result is zero, cleared otherwise

Side effects

None

Description

This primitive bit-shifts data rightwards, filling the most-significant bits with zeroes (that is, it divides by powers of two).

To be consistent with Shift Left, this primitive sets the N flag on **MULTOS** implementations that support this flag. In practice, however, the result will always have a zero in its most significant bit, so the N flag will always be cleared.

The effect of the primitive is undefined if any of the following is true:

- b2 is zero
- b3 is zero
- $b3 \geq 8 * b2$

4.25 Store CCR

Primitive set and number

Set zero, number 0x06

Arguments

None

In Parameters

{DT[-1], 1} = byte to be placed in CCR.

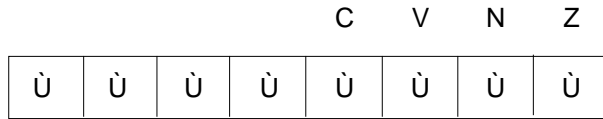
Out Parameters

None

Effect on DT

1 byte is popped.

Condition Code



All CCR bits are set from the specified value, bit Z being the least significant bit.

Side effects

None

Description

This primitive pops one byte from the stack, copying its bit settings to the CCR.

**Section
4**

5. Optional Primitives

5.1 Introduction

This section describes the primitive functions available to MEL applications, but which may not be supported by all MULTOS implementations. The primitives are listed in alphabetic order. Each primitive is described under the same headings used in section 4.

5.2 Call Codelet

Primitive set and number

Set zero, number 0x83

Arguments

None

In Parameters

{DT[-4], 2} = codeletID of codelet to call

{DT[-2], 2} = code address within codelet

Out Parameters

{DT'[-6], 2} = codeletID of caller

{DT'[-4], 2} = saved LB of caller

{DT'[-2], 2} = saved CP of caller

Effect on DT

2 bytes are pushed. LB is set to the new DT.

Condition Code

C V N Z

-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

None

Description

This primitive is used to call a codelet, as identified by the specified codeletID.

The effect is:

- If the codeletID is zero, the current application is called at the specified code address.
- If the codeletID is non-zero, and corresponds to a codelet present on the MCD, control passes to the codelet at the specified address.
- If the codeletID is non-zero, and does not correspond to a codelet present on the MCD, an abend occurs.

This primitive can therefore be used:.

- To pass control from an application to a codelet.
- To pass control from one codelet to another.
- To pass control from an application to the same application, in which case the primitive has the same effect as CALL(t=0) except for the different treatment of the stack.
- To pass control from a codelet to the same codelet; again, this is similar to CALL(t=0).
- To pass control from a codelet to the current application.

Note: Because a codeletID of zero is used to denote the current application, and there is no means to denote any other application, this primitive cannot be used, directly or via one or more codelets, to pass control from one application to another.

The *Out Parameters* and *Effect on DT* are described from the point of view of the specified codelet, not that of the caller. The caller only sees these effects directly if it uses this primitive to call itself. In all other cases, the perceived effect on the stack is determined when Return From Codelet is executed (see section 5.18).

Using the notation for describing instructions, with the additional identifier CI for the codeletID, the effect of this primitive may be written:

$$DT' = DT + 2$$

$$LB' = DT';$$

$$CP' = \{DT[-2], 2\};$$

$$CI' = \{DT[-4], 2\};$$

$$\{DT' [-6], 2\} = CI;$$

$$\{DT' [-4], 2\} = LB;$$

$$\{DT' [-2], 2\} = CP;$$

5.3 Checksum

Primitive set and number

Set zero, number 0x82

Arguments

None

In Parameters

$\{DT[-4], 2\}$ = length of block to checksum, in bytes

$\{DT[-2], 2\}$ = segment address of block to checksum

Out Parameters

$\{DT[-4], 4\}$ = checksum

Effect on DT

None

Condition Code

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

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

None

Description

This primitive generates a 4-byte checksum over the block {{DT[-2]. 2}, {DT[-4], 2}}. 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.

Consider the following example:

```

set transaction protection on ; (1)
update an area of Static ; (2)
calculate the checksum ; (3)
copy the area of Static into Public ; (4)
calculate the checksum of the copy in Public ; (5)

```

The checksum calculated in line (3) will be over the updated Static data. However, line (4) copies the non-updated data into Public, because the update has not been committed. Therefore, the checksum calculated in line (5) will be over the non-updated data, and will be different from that calculated in line (3).

The checksum is calculated using the following algorithm:

```

unsigned byte checksum[4];
checksum[0] = 0x5a;
checksum[1] = 0xa5;
checksum[2] = 0x5a;
checksum[3] = 0xa5;
for ( j = 0; j < {DT[-4], 2}; ++ j )
{
    // add data byte into most significant byte of checksum
    checksum[0] += {{DT[-2], 2 + j, 1};
    // add each byte of checksum into the next byte
    checksum[1] += checksum[0];
    checksum[2] += checksum[1];
    checksum[3] += checksum[1];
}

```

The carries are dropped from each addition. For example, the following code fragment calculates the checksum of the word 0x9988:

```

STACK: Push the word 0x9988
STACK: Push the word 2 (length to check)
LOADA: Load the address of DT[-6]
PRIMRET: Checksum
        // checksum the two bytes 0x9988

```

The result is 0x7b13059c, calculated as follows:

```

checksum[0] = 0x5a + 0x99 = 0xf3
checksum[1] = 0xa5 + 0xf3 = 0x98
checksum[2] = 0x5a + 0x98 = 0xf2
checksum[3] = 0xa5 + 0xf2 = 0x97
checksum[0] = 0xf3 + 0x88 = 0x7b
checksum[1] = 0x98 + 0x7b = 0x13
checksum[2] = 0xf2 + 0x13 = 0x05
checksum[3] = 0x97 + 0x05 = 0x9c

```

The checksum is returned in Dynamic, where it overwrites the length and segment address of the checksummed area.

Note: The exact effect of the primitive is undefined if the checksummed area includes the primitive's parameters, as in the following example:

```

STACK: Push the word 0x9988
STACK: Push the word 6 (length to check)
LOADA: Load the address of DT[-6]
PRIMRET: Checksum
        // checksum the six bytes:
        //     0x99880006<address of 0x9988>
        // - but may not have the desired effect

```

It is valid to calculate the checksum of a block of length zero; the result is 0x5aa55aa5.

5.4 Delegate

Primitive set and number

Set zero, number 0x80

Arguments

None

In Parameters

{DT[-2], 2} = segment address of the identifier of the delegate application

Out Parameters

None

Effect on DT

2 bytes are popped.

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

Public may be changed by the delegate. If delegation fails for any of the reasons listed below, Public is unchanged except that SW1, SW2 contain the status code 0x6a83.

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.

Description

This primitive allows one application to delegate to another, as described in section 2.4. The delegate application must be specified by its identifier, which is defined as a one-byte AID length followed by the AID (Application Identifier). The segment address to be supplied on the stack is the segment address of the AID length. It is permissible for this to be a partial AID, in which case the first matching application becomes the delegate.

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 specified delegate is already active (an attempt is made to delegate recursively).
- the number of suspended applications reaches an implementation-defined limit.

5.5 DES ECB Decipher**Primitive set and number**

Set zero, number 0xc5

Arguments

None

In Parameters

{DT[-0x6], 2} = segment address of key, of length 8 bytes

Section 5

{DT[-0x4], 2} = segment address of plaintext, of length 8 bytes

{DT[-0x2], 2} = segment address of ciphertext, of length 8 bytes

Out Parameters

None

Effect on DT

6 bytes are popped.

Condition Code

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

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

The plaintext is written at the specified segment address.

Description

This primitive recovers an 8-byte plaintext from a DES ECB ciphertext.

The parity bits of the key are ignored.

5.6 DES ECB Encipher

Primitive set and number

Set zero, number 0xc1

Arguments

None

In Parameters

{DT[-0x6], 2} = segment address of key, of length 8 bytes

{DT[-0x4], 2} = segment address of ciphertext, of length 8 bytes

{DT[-0x2], 2} = segment address of plaintext, of length 8 bytes

Out Parameters

None

Effect on DT

6 bytes are popped.

Condition Code

	C	V	N	Z
-	-	-	-	-

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

The ciphertext is written at the specified segment address.

Description

This primitive generates a DES ECB ciphertext from an 8-byte message.

The parity bits of the key are ignored.

5.7 Generate Asymmetric Hash

Primitive set and number

Set one, number 0xc4

Arguments

b2: mode, coded as follows:

- 0 - use standard Initial Value (IV) (in which every byte is 0x55)
- 1 - use specified Initial Value

The effect of any other value in b2 is undefined.

In Parameters

{DT[-0x8], 2} = segment address of Initial Value, which is 16 bytes in length. Initial Value is specified only in mode 1; this parameter is ignored in mode 0.

{DT[-0x6], 2} = plaintext_length

{DT[-0x4], 2} = segment address of hash digest, which is 16 bytes in length

{DT[-0x2], 2} = segment address of plaintext, which is of length plaintext_length

Out Parameters

None

Effect on DT

In mode 0, 6 bytes are popped.

In mode 1, 8 bytes are popped.

Condition Code

	C	V	N	Z
-	-	-	-	-
C	Previous value remains unchanged			
V	Previous value remains unchanged			
N	Previous value remains unchanged			
Z	Previous value remains unchanged			

Side effects

The hash digest is written at the specified segment address.

Description

This primitive generates an asymmetric hash digest over a message of arbitrary length.

5.8 Generate DES CBC Signature

Primitive set and number

Set zero, number 0xc6

Arguments

None

In Parameters

{DT[-0xa], 2} = plaintext_length

{DT[-0x8], 2} = segment address of IV, of length 8 bytes

{DT[-0x6], 2} = segment address of key, of length 8 bytes

{DT[-0x4], 2} = segment address of signature, of length 8 bytes

{DT[-0x2], 2} = segment address of plaintext, of length
plaintext_length

Out Parameters

None

Effect on DT

0xa bytes are popped.

Condition Code

	C	V	N	Z
-	-	-	-	-

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

The signature is written at the specified segment address.

Description

This function generates a signature from the plaintext by treating it as a series of 8-byte blocks, and applying the DES encipherment function to each block in turn. The blocks are chained by cipher block chaining:

```
signature = iv;
for each 8-byte block in plaintext
{
    signature = signature ^ block;
    signature = E[key](signature);
}
```

The primitive operates only on complete 8-byte blocks in the plaintext. If plaintext_length is not an exact multiple of 8, trailing bytes are ignored.

The parity bits of the key are ignored.

5.9 Generate Triple DES CBC Signature

Primitive set and number

Set zero, number 0xc7

Arguments

None

In Parameters

{DT[-0xa], 2} = plaintext_length

{DT[-0x8], 2} = segment address of IV, of length 8 bytes

{DT[-0x6], 2} = segment address of key, of length 16 bytes, which is the concatenation of:

- K1, used for the E operations, see below
- K2, used for the D operation, see below

{DT[-0x4], 2} = segment address of signature, of length 8 bytes

{DT[-0x2], 2} = segment address of message, of length plaintext_length

Out Parameters

None

Effect on DT

0xa bytes are popped.

Condition Code

C V N Z

-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The signature is written at the specified segment address.

Description

This function generates a signature from the plaintext by treating it as a series of 8-byte blocks, and applying the function $E[K1](D[K2](E[K1](\text{block})))$ to each block in turn, where E is DES encipherment, D is DES decipherment and K1, K2 are 8-byte keys. The blocks are chained by cipher block chaining:

```
signature = iv;
for each 8-byte block in plaintext
{
    signature = signature ^ block;
    signature = E[K1](signature);
    signature = D[K2](signature);
    signature = E[K1](signature);
}
```

The primitive operates only on complete 8-byte blocks in the plaintext. If plaintext_length is not an exact multiple of 8, trailing bytes are ignored.

The parity bits of the keys are ignored.

5.10 Get Delegator AID

Primitive set and number

Set one, number 0x81

Arguments

b2 = maximum length of data to be retrieved

In Parameters

{DT[-2], 2} = segment address to which the AID should be copied

Out Parameters

None

Effect on DT

2 bytes are popped

Condition Code

	C	V	N	Z
-	-	-	-	-
	Ù	-	-	Ù

- C Set if data retrieved was less than requested, cleared otherwise
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if there is no Delegator, cleared otherwise

Side effects

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 and
- the actual length of the AID, plus one for the length byte.

Description

This primitive allows an application to determine whether it is a delegate and, if so, the AID of its delegator.

5.11 Get Random Number

Primitive set and number

Set zero, number 0xc4

Arguments

None

In Parameters

None

Out Parameters

{DT'[-8], 8} = random number

Effect on DT

8 bytes are pushed.

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

None

Description

This primitive returns an eight byte random number.

The number may be generated by a true random number generator implemented in hardware. Alternatively, it may be generated as a pseudo-random number from a seed value. In either case, the number is provided in a way that guarantees its secrecy. It is not possible for any coresident application to determine what number was provided or will be provided subsequently.

5.12 Modular Exponentiation

Primitive set and number

Set zero, number 0xc8

Arguments

None

In Parameters

{DT[-0xc], 2} = exponent_length

{DT[-0xa], 2} = modulus_length

{DT[-0x8], 2} = segment address of exponent, of length exponent_length

{DT[-0x6], 2} = segment address of modulus, of length modulus_length

{DT[-0x4], 2} = segment address of base, of length modulus_length

{DT[-0x2], 2} = segment address of result, of length modulus_length

Out Parameters

None

Effect on DT

0xc bytes are popped.

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The result (the base to the power of the exponent, modulo the modulus) is written at the specified segment address.

Description

This primitive performs a modular exponentiation operation, that is, it calculates an exponent modulo a modulus.

5.13 Modular Exponentiation CRT

Primitive set and number

Set zero, number 0xc9

Arguments

None

In Parameters

Where:

- modulus_length is even
- the factors of the modulus are known to be p and q, where p < q and both are prime
- q can be expressed in modulus_length / 2 bytes

the in parameters are:

$\{DT[-0xa], 2\} = \text{modulus_length}$

$\{DT[-0x8], 2\} =$ segment address of a block of length modulus_length , containing the concatenation of the following:

- exponent modulo $p - 1$, of length $\text{modulus_length} / 2$
(referred to as dp in some documentation)
- exponent modulo $q - 1$, of length $\text{modulus_length} / 2$
(referred to as dq in some documentation)

$\{DT[-0x6], 2\} =$ segment address of a block of length $\text{modulus_length} * 3 / 2$, containing the concatenation of the following:

- p , of length $\text{modulus_length} / 2$
- q , of length $\text{modulus_length} / 2$
- reciprocal of q modulo p , of length $\text{modulus_length} / 2$
(referred to as u in some documentation)

$\{DT[-0x4], 2\} =$ segment address of base, of length modulus_length

$\{DT[-0x2], 2\} =$ segment address of result, of length modulus_length

Out Parameters

None

Effect on DT

0xa bytes are popped.

Condition Code

C	V	N	Z
-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The result (the base to the power of the exponent, modulo the modulus) is written at the specified segment address.

Description

This primitive calculates an exponent modulo a modulus, using a Chinese Remainder Theorem (CRT) representation of the modulus.

This primitive should execute faster than regular modular exponentiation for the same size modulus. In addition, since the modulus is expressed as the product of two smaller operands, it allows the use of a larger modulus than can be handled directly.

5.14 Modular Multiplication

Primitive set and number

Set zero, number 0xc2

Arguments

None

In Parameters

{DT[-0x8], 2} = modulus_length

{DT[-0x6], 2} = segment address of left operand, of length modulus_length

{DT[-0x4], 2} = segment address of right operand, of length modulus_length

{DT[-0x2], 2} = segment address of modulus, of length
modulus_length

Out Parameters

None

Effect on DT

8 bytes are popped.

Condition Code

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

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The result (left operand multiplied by right operand, modulo the modulus) overwrites the left operand.

Description

This primitive calculates a product modulo a modulus.

5.15 Modular Reduction

Primitive set and number

Set zero, number 0xc3

Arguments

None

Section 5

In Parameters

{DT[-0x8], 2} = operand_length

{DT[-0x6], 2} = modulus_length

{DT[-0x4], 2} = segment address of operand, of length operand_length

{DT[-0x2], 2} = segment address of modulus, of length modulus_length

Out Parameters

None

Effect on DT

8 bytes are popped.

Condition Code

					C	V	N	Z
					-	-	-	-

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

The result (operand modulo the modulus) overwrites the operand.

Description

This primitive reduces an operand modulo a modulus.

5.16 Query Codelet

Primitive set and number

Set zero, number 0x84

Arguments

None

In Parameters

{DT[-2], 2} = codeletID of codelet to query

Out Parameters

{DT[-2], 2} = codeletID of codelet to query, unchanged

Effect on DT

None

Condition Code

				C	V	N	Z
-	-	-	-	-	-	-	Ù

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Set if the specified codelet exists, or if zero is specified, cleared otherwise

Side effects

None

Description

This primitive is used to query the presence of a codelet, as identified by the specified codeletID. The codeletID may be zero, in which case the result is always to set the Z flag. The codeletID is not popped; it is convenient to leave it in place for use by a subsequent call to Call Codelet.

5.17 Reset Session Data

Primitive set and number

Set zero, number 0x81

Arguments

None

In Parameters

None

Out Parameters

None

Effect on DT

None

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

None on the memory of the calling application

Description

If the calling application is executing as a shell, this primitive zeroes the session data of all other applications. The caller's session data is unaffected.

If the calling application is not executing as a shell, this primitive has no effect.

5.18 Return From Codelet**Primitive set and number**

Set two, number 0x80

Arguments

b2 = number of bytes of stack to discard

b3 = number of bytes of stack to preserve

In Parameters

{LB[-6], 2} = codeletID to which to return

{LB[-4], 2} = saved LB

{LB[-2], 2} = saved CP

{LB[-6-b2], b2} = input parameters of the codelet call, to be discarded

{DT[-b3], b3} = output parameters of the codelet call, to be preserved

Out Parameters

{DT'[-b3], b3} = preserved output parameters of the codelet call

Section 5

Effect on DT

DT' is LB-6-b2+b3. LB' is saved from {LB[-4], 2}.

Condition Code

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

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

None

Description

This primitive is used to return from a codelet, as identified by the specified codeletID. See also the description of Call Codelet (section 5.2).

In most cases, the *In Parameters* below LB are exactly as set up by Call Codelet; the application does not set them up explicitly. However, as with CALL and PRIMRET (return), it is possible for the routine called by Call Codelet to manipulate these parameters to return to a code address other than the instruction after Call Codelet.

The *Out Parameters* and *Effect on DT* are described from the point of view of the codelet (or the current application) to which control is returned, not that of the codelet that calls Return From Codelet. The caller only sees these effects if it uses this primitive to return to itself; in all other cases, its execution thread is lost.

Using the notation for describing instructions, with the additional identifier CI for the codeletID, the effect of this primitive may be written:

```

CP' = {LB[-2], 2};
LB' = {LB[-4], 2};
CI' = {LB[-6], 2};
DT' = LB-b2+b3-6;
{DT' [-b3], b3} = {DT [-b3], b3};

```

5.19 Set Transaction Protection

Primitive set and number

Set one, number 0x80

Arguments

b2: a bitmap as follows:

- bits 7 (most significant), 6, 5, 4, 3 and 2: '0' (Otherwise, the effect of the primitive is undefined.)
- bit 1: set to '1' if transaction protection is to be on when the primitive returns, '0' otherwise
- bit 0: set to '1' if any uncommitted writes are to be committed, '0' otherwise.

In Parameters

None

Out Parameters

None

Effect on DT

None

Condition Code

	C	V	N	Z
-	-	-	-	-

- C Previous value remains unchanged
- V Previous value remains unchanged
- N Previous value remains unchanged
- Z Previous value remains unchanged

Side effects

The primitive itself has no direct effects. Transaction protection affects the behaviour of all writes to Static, to the ATR file, and to ATR historical characters.

Description

The primitive operates as follows:

```
if ( transaction protection is on )
{
    if ( b2 & 1 ) // if bit 0 is set
        commit any uncommitted writes;
    else
        discard any uncommitted writes;
}
if ( b2 & 2 ) // if bit 1 is set
    set transaction protection on;
else
    set transaction protection 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), each write is applied as the instruction is executed. If the next instruction reads the data back, the new values are obtained. For example, the following code sets SB[0] to 5:

```
; transaction protection is off
SETB SB[0] to 3
ADDB 1 to SB[0]
ADDB 1 to SB[0]
```

When transaction protection is on, writes to non-volatile memory are not applied immediately. They are applied only if and when the application explicitly commits the writes. If the application exits, delegates, or abends, uncommitted writes are discarded.

The application can also explicitly rollback uncommitted writes. Uncommitted writes are not visible to the application if it reads them back. Therefore, the following code:

```
; transaction protection is off
SETB SB[0] to 3
Set Transaction Protection On
ADDB 1 to SB[0]
ADDB 1 to SB[0]
Commit Transaction Protection
```

updates SB[0] to 4, not to 5. Both ADDB instructions increment the value of SB[0] that was set by SETB; the effect of the first ADDB is lost.

Transaction protection applies to writes to Static and writes performed to system memory by Set ATR File Record and Set ATR Historical Characters. 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. See also section 1.2.2.1.

5.20 SHA-1

Primitive set and number

Set zero, number 0xca

Arguments

None

In Parameters

{DT[-0x6], 2} = plaintext_length

{DT[-0x4], 2} = segment address of hash digest, which is of length 20

{DT[-0x2], 2} = segment address of plaintext, which is of length plaintext_length

Section 5

Out Parameters

None

Effect on DT

6 bytes are popped.

Condition Code

					C	V	N	Z
					-	-	-	-

C Previous value remains unchanged

V Previous value remains unchanged

N Previous value remains unchanged

Z Previous value remains unchanged

Side effects

None

Description

This primitive calculates the SHA-1 hash of the plaintext.

The primitive works correctly, generating a 20 byte hash, even if plaintext_length is zero, or a positive length less than 20.

Appendix A Instruction Map

op	t	b1 w1	b2 w2	b3 w3	b4	description	size
0x00 SYSTEM	0					no operation	1
	1	SW1, SW2				set SW1 and SW2	3
	2	La				set La	3
	3	SW1, SW2		La		set SW1, SW2 and La	5
	4					exit	1
	5	SW1, SW2				set SW1 and SW2, then exit	3
	6	La				set La, then exit	3
	7	SW1, SW2		La		set SW1, SW2 and La, then exit	5
0x01 BRANCH	0						
	1	relcp				branch by relcp if EQ	2
	2	relcp				branch by relcp if LT	2
	3	relcp				branch by relcp if LE	2
	4	relcp				branch by relcp if GT	2
	5	relcp				branch by relcp if GE	2
	6	relcp				branch by relcp if NE	2
	7	relcp				branch by relcp	2
0x02 JUMP	0					pop code address from the stack, jump to it	1
	1	cp				jump to cp if EQ	3
	2	cp				jump to cp if LT	3
	3	cp				jump to cp if LE	3
	4	cp				jump to cp if GT	3
	5	cp				jump to cp if GE	3
	6	cp				jump to cp if NE	3
	7	cp				jump to cp	3
0x03 CALL	0					pop code address from the stack, call it	1
	1	cp				call cp if EQ	3
	2	cp				call cp if LT	3
	3	cp				call cp if LE	3
	4	cp				call cp if GT	3

Appendix

	5	cp			call cp if GE	3
	6	cp			call cp if NE	3
	7	cp			call cp	3
0x04 STACK	0	n			push n bytes of zeroes on to the stack	2
	1	b			push byte b on to the stack	2
	2	w			push word w on to the stack	3
	3					
	4	n			pop n bytes from the stack	2
	5				pop one byte from the stack	1
	6				pop one word from the stack	1
	7					
0x05 PRIMRET	0	p			call primitive p in set zero	2
	1		p	a	call primitive p in set one, passing argument a	3
	2	p	a	b	call primitive p in set two, passing arguments a and b	4
	3	p	a	bc	call primitive p in set three, passing arguments a, b and c	5
	4				return from called function	1
	5	in			return from called function, discarding in parameters	2
	6	out			return from called function, preserving out parameters	2
	7	in	out		return from called function, discarding in parameters and preserving out parameters	3
0x06 (unused)	0					
	1					
	2					
	3					
	4					
	5					
	6					
	7					
0x07 LOAD	0	n			duplicate top n bytes of stack	2
	1	n	offset		load n bytes from SB[offset] on to stack	4
	2	n	offset		load n bytes from ST[offset] on to stack	4
	3	n	offset		load n bytes from DB[offset] on to stack	4
	4	n	offset		load n bytes from LB[offset] on to stack	4

	5	n	offset	load n bytes from DT[offset] on to stack	4
	6	n	offset	load n bytes from PB[offset] on to stack	4
	7	n	offset	load n bytes from PT[offset] on to stack	4
0x08 STORE	0	n		pop n bytes from stack and store them to DT[-2*n]	2
	1	n	offset	pop n bytes from stack and store them to SB[offset]	4
	2	n	offset	pop n bytes from stack and store them to ST[offset]	4
	3	n	offset	pop n bytes from stack and store them to DB[offset]	4
	4	n	offset	pop n bytes from stack and store them to LB[offset]	4
	5	n	offset	pop n bytes from stack and store them to DT[offset]	4
	6	n	offset	pop n bytes from stack and store them to PB[offset]	4
	7	n	offset	pop n bytes from stack and store them to PT[offset]	4
0x09 LOADI	0	n		load n bytes from the segment address at DT[-2] on to stack	2
	1	n	offset	load n bytes from the segment address in SB[offset] on to stack	4
	2	n	offset	load n bytes from the segment address in ST[offset] on to stack	4
	3	n	offset	load n bytes from the segment address in DB[offset] on to stack	4
	4	n	offset	load n bytes from the segment address in LB[offset] on to stack	4
	5	n	offset	load n bytes from the segment address in DT[offset] on to stack	4
	6	n	offset	load n bytes from the segment address in PB[offset] on to stack	4
	7	n	offset	load n bytes from the segment address in PT[offset] on to stack	4
0x0a STOREI	0	n		pop n bytes from stack and store them to the segment address in DT[-2-n]	2
	1	n	offset	pop n bytes from stack and store them to the segment address in SB[offset]	4
	2	n	offset	pop n bytes from stack and store them to the segment	

Appendix

				address in ST[offset]	4
	3	n	offset	pop n bytes from stack and store them to the segment address in DB[offset]	4
	4	n	offset	pop n bytes from stack and store them to the segment address in LB[offset]	4
	5	n	offset	pop n bytes from stack and store them to the segment address in DT[offset]	4
	6	n	offset	pop n bytes from stack and store them to the segment address in PB[offset]	4
	7	n	offset	pop n bytes from stack and store them to the segment address in PT[offset]	4
0x0b LOADA	0				
	1		offset	load the segment address of SB[offset] on to the stack	3
	2		offset	load the segment address of ST[offset] on to the stack	3
	3		offset	load the segment address of DB[offset] on to the stack	3
	4		offset	load the segment address of LB[offset] on to the stack	3
	5		offset	load the segment address of DT[offset] on to the stack	3
	6		offset	load the segment address of PB[offset] on to the stack	3
	7		offset	load the segment address of PT[offset] on to the stack	3
0x0c INDEX	0				
	1	b	offset	load the segment address of a record of SB[offset] on to the stack	4
	2	b	offset	load the segment address of a record of ST[offset] on to the stack	4
	3	b	offset	load the segment address of a record of DB[offset] on to the stack	4
	4	b	offset	load the segment address of a record of LB[offset] on to the stack	4
	5	b	offset	load the segment address of a record of DT[offset] on to the stack	4
	6	b	offset	load the segment address of a record of PB[offset] on to the stack	4
	7	b	offset	load the segment address of a record of PT[offset] on to the stack	4
0x0d SETB	0	b		set the byte at DT[-1] to b	2
	1	b	offset	set the byte at SB[offset] to b	4

	2	b	offset	set the byte at ST[offset] to b	4
	3	b	offset	set the byte at DB[offset] to b	4
	4	b	offset	set the byte at LB[offset] to b	4
	5	b	offset	set the byte at DT[offset] to b	4
	6	b	offset	set the byte at PB[offset] to b	4
	7	b	offset	set the byte at PT[offset] to b	4
0x0e CMPB	0	b		compare the byte at DT[-1] with b	2
	1	b	offset	compare the byte at SB[offset] with b	4
	2	b	offset	compare the byte at ST[offset] with b	4
	3	b	offset	compare the byte at DB[offset] with b	4
	4	b	offset	compare the byte at LB[offset] with b	4
	5	b	offset	compare the byte at DT[offset] with b	4
	6	b	offset	compare the byte at PB[offset] with b	4
	7	b	offset	compare the byte at PT[offset] with b	4
0x0f ADDB	0	b		add b to the byte at DT[-1]	2
	1	b	offset	add b to the byte at SB[offset]	4
	2	b	offset	add b to the byte at ST[offset]	4
	3	b	offset	add b to the byte at DB[offset]	4
	4	b	offset	add b to the byte at LB[offset]	4
	5	b	offset	add b to the byte at DT[offset]	4
	6	b	offset	add b to the byte at PB[offset]	4
	7	b	offset	add b to the byte at PT[offset]	4
0x10 SUBB	0	b		subtract b from the byte at DT[-1]	2
	1	b	offset	subtract b from the byte at SB[offset]	4
	2	b	offset	subtract b from the byte at ST[offset]	4
	3	b	offset	subtract b from the byte at DB[offset]	4
	4	b	offset	subtract b from the byte at LB[offset]	4
	5	b	offset	subtract b from the byte at DT[offset]	4
	6	b	offset	subtract b from the byte at PB[offset]	4
	7	b	offset	subtract b from the byte at PT[offset]	4

Appendix

0x11 SETW	0	w		set the word at DT[-2] to w	3
	1	w	offset	set the word at SB[offset] to w	5
	2	w	offset	set the word at ST[offset] to w	5
	3	w	offset	set the word at DB[offset] to w	5
	4	w	offset	set the word at LB[offset] to w	5
	5	w	offset	set the word at DT[offset] to w	5
	6	w	offset	set the word at PB[offset] to w	5
	7	w	offset	set the word at PT[offset] to w	5
0x12 CMPW	0	w		compare the word at DT[-2] with w	3
	1	w	offset	compare the word at SB[offset] with w	5
	2	w	offset	compare the word at ST[offset] with w	5
	3	w	offset	compare the word at DB[offset] with w	5
	4	w	offset	compare the word at LB[offset] with w	5
	5	w	offset	compare the word at DT[offset] with w	5
	6	w	offset	compare the word at PB[offset] with w	5
	7	w	offset	compare the word at PT[offset] with w	5
0x13 ADDW	0	w		add w to the word at DT[-2]	3
	1	w	offset	add w to the word at SB[offset]	5
	2	w	offset	add w to the word at ST[offset]	5
	3	w	offset	add w to the word at DB[offset]	5
	4	w	offset	add w to the word at LB[offset]	5
	5	w	offset	add w to the word at DT[offset]	5
	6	w	offset	add w to the word at PB[offset]	5
	7	w	offset	add w to the word at PT[offset]	5
0x14 SUBW	0	w		subtract w from the word at DT[-2]	3
	1	w	offset	subtract w from the word at SB[offset]	5
	2	w	offset	subtract w from the word at ST[offset]	5
	3	w	offset	subtract w from the word at DB[offset]	5
	4	w	offset	subtract w from the word at LB[offset]	5
	5	w	offset	subtract w from the word at DT[offset]	5
	6	w	offset	subtract w from the word at PB[offset]	5
	7	w	offset	subtract w from the word at PT[offset]	5

0x15 CLEARN	0	n		clear the n bytes at DT[-n]	2
	1	n	offset	clear the n bytes at SB[offset]	4
	2	n	offset	clear the n bytes at ST[offset]	4
	3	n	offset	clear the n bytes at DB[offset]	4
	4	n	offset	clear the n bytes at LB[offset]	4
	5	n	offset	clear the n bytes at DT[offset]	4
	6	n	offset	clear the n bytes at PB[offset]	4
	7	n	offset	clear the n bytes at PT[offset]	4
0x16 TESTN	0	n		test the n bytes at DT[-n]	2
	1	n	offset	test the n bytes at SB[offset]	4
	2	n	offset	test the n bytes at ST[offset]	4
	3	n	offset	test the n bytes at DB[offset]	4
	4	n	offset	test the n bytes at LB[offset]	4
	5	n	offset	test the n bytes at DT[offset]	4
	6	n	offset	test the n bytes at PB[offset]	4
	7	n	offset	test the n bytes at PT[offset]	4
0x17 INCN	0	n		increment the n bytes at DT[-n]	2
	1	n	offset	increment the n bytes at SB[offset]	4
	2	n	offset	increment the n bytes at ST[offset]	4
	3	n	offset	increment the n bytes at DB[offset]	4
	4	n	offset	increment the n bytes at LB[offset]	4
	5	n	offset	increment the n bytes at DT[offset]	4
	6	n	offset	increment the n bytes at PB[offset]	4
	7	n	offset	increment the n bytes at PT[offset]	4
0x18 DECN	0	n		decrement the n bytes at DT[-n]	2
	1	n	offset	decrement the n bytes at SB[offset]	4
	2	n	offset	decrement the n bytes at ST[offset]	4
	3	n	offset	decrement the n bytes at DB[offset]	4
	4	n	offset	decrement the n bytes at LB[offset]	4
	5	n	offset	decrement the n bytes at DT[offset]	4
	6	n	offset	decrement the n bytes at PB[offset]	4
	7	n	offset	decrement the n bytes at PT[offset]	4
0x19 NOTN	0	n		invert the n bytes at DT[-n]	2
	1	n	offset	invert the n bytes at SB[offset]	4
	2	n	offset	invert the n bytes at ST[offset]	4
	3	n	offset	invert the n bytes at DB[offset]	4
	4	n	offset	invert the n bytes at LB[offset]	4
	5	n	offset	invert the n bytes at DT[offset]	4
	6	n	offset	invert the n bytes at PB[offset]	4
	7	n	offset	invert the n bytes at PT[offset]	4

0x1a CMPN	0	n		compare the n bytes located at DT[-2*n] with the n bytes at DT[-n]	2
	1	n	offset	compare the n bytes at SB[offset] with the n bytes at DT[-n]	4
	2	n	offset	compare the n bytes at ST[offset] with the n bytes at DT[-n]	4
	3	n	offset	compare the n bytes at DB[offset] with the n bytes at DT[-n]	4
	4	n	offset	compare the n bytes at LB[offset] with the n bytes at DT[-n]	4
	5	n	offset	compare the n bytes at DT[offset] with the n bytes at DT[-n]	4
	6	n	offset	compare the n bytes at PB[offset] with the n bytes at DT[-n]	4
	7	n	offset	compare the n bytes at PT[offset] with the n bytes at DT[-n]	4
0x1b ADDN	0	n		add the n bytes at DT[-n] to the n bytes located at DT[-2*n]	2
	1	n	offset	add the n bytes at DT[-n] to the n bytes at SB[offset]	4
	2	n	offset	add the n bytes at DT[-n] to the n bytes at ST[offset]	4
	3	n	offset	add the n bytes at DT[-n] to the n bytes at DB[offset]	4
	4	n	offset	add the n bytes at DT[-n] to the n bytes at LB[offset]	4
	5	n	offset	add the n bytes at DT[-n] to the n bytes at DT[offset]	4
	6	n	offset	add the n bytes at DT[-n] to the n bytes at PB[offset]	4
	7	n	offset	add the n bytes at DT[-n] to the n bytes at PT[offset]	4
0x1c SUBN	0	n		subtract the n bytes at DT[-n] from the n bytes located at DT[-2*n]	2
	1	n	offset	subtract the n bytes at DT[-n] from the n bytes at SB[offset]	4
	2	n	offset	subtract the n bytes at DT[-n] from the n bytes at ST[offset]	4
	3	n	offset	subtract the n bytes at DT[-n] from the n bytes at DB[offset]	4
	4	n	offset	subtract the n bytes at DT[-n] from the n bytes at LB[offset]	4
	5	n	offset	subtract the n bytes at DT[-n] from the n bytes at DT[offset]	4
	6	n	offset	subtract the n bytes at DT[-n] from the n bytes at PB[offset]	4
	7	n	offset	subtract the n bytes at DT[-n] from the n bytes at PT[offset]	4

0x1d ANDN	0	n		bitwise-and the n bytes at DT[-n] into the n bytes located at DT[-2*n]	2
	1	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at SB[offset]	4
	2	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at ST[offset]	4
	3	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at DB[offset]	4
	4	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at LB[offset]	4
	5	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at DT[offset]	4
	6	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at PB[offset]	4
	7	n	offset	bitwise-and the n bytes at DT[-n] into the n bytes at PT[offset]	4
0x1e ORN	0	n		bitwise-or the n bytes at DT[-n] into the n bytes located at DT[-2*n]	2
	1	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at SB[offset]	4
	2	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at ST[offset]	4
	3	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at DB[offset]	4
	4	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at LB[offset]	4
	5	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at DT[offset]	4
	6	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at PB[offset]	4
	7	n	offset	bitwise-or the n bytes at DT[-n] into the n bytes at PT[offset]	4
0x1f XORN	0	n		bitwise-xor the n bytes at DT[-n] into the n bytes located at DT[-2*n]	2
	1	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at SB[offset]	4
	2	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at ST[offset]	4
	3	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at DB[offset]	4
	4	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at LB[offset]	4
	5	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at DT[offset]	4
	6	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at PB[offset]	4
	7	n	offset	bitwise-xor the n bytes at DT[-n] into the n bytes at PT[offset]	4

Appendix B List of Primitives

Set zero Mandatory:

0x01 Check Case
 0x02 Reset WWT
 0x05 Load CCR
 0x06 Store CCR
 0x07 Set ATR File Record
 0x08 Set ATR Historical Characters
 0x09 Get Memory Reliability
 0x0a Lookup
 0x0b Memory Compare
 0x0c Memory Copy

Optional:

0x80 Delegate
 0x81 Reset Session Data
 0x82 Checksum
 0x83 Call Codelet
 0x84 Query Codelet
 0xc1 DES ECB Encipher
 0xc2 Modular Multiplication
 0xc3 Modular Reduction
 0xc4 Get Random Number
 0xc5 DES ECB Decipher
 0xc6 Generate DES CBC Signature
 0xc7 Generate Triple DES CBC Signature
 0xc8 Modular Exponentiation
 0xc9 Modular Exponentiation CRT
 0xca SHA-1

Set one Mandatory

0x00 Query0
 0x01 Query1
 0x02 Query2
 0x03 Query3
 0x08 DivideN
 0x09 Get DIR File Record
 0x0a Get File Control Information
 0x0b Get Manufacturer Data
 0x0c Get **MULTOS** Data
 0x0d Get Purse Type
 0x0e Memory Copy Fixed Length
 0x0f Memory Compare Fixed Length
 0x10 MultiplyN

Optional:

0x80 Set Transaction Protection
 0x81 Get Delegator AID
 0xc4 Generate Asymmetric Hash

Set two Mandatory:

0x01 Bit Manipulate Byte
 0x02 ShiftLeft
 0x03 ShiftRight
 Optional:
 0x80 Return From Codelet

Set three Mandatory:

0x01 Bit Manipulate Word
 Optional:
 (none)

Appendix C Implementation-Dependent Features

The MEL instruction set and primitive set are fixed. However, there are a number of areas where different implementations may differ. These include:

Memory size and addresses

Static size and session data size are determined by the application, but Public size and Dynamic size (that is, the maximum size of Dynamic) are determined by the implementation.

The segment address of SB[0] is always zero and that of ST[0] is always equal to the size of the Static data. The segment addresses of other registers are implementation-dependent. Moreover, some implementations may provide different segment addresses to different applications and/or to different transactions processed by the same application. For this reason, applications should not expect segment addresses in Dynamic or Public to have a duration longer than one command-response pair.

Block sizes and temporary space

The effect of specifying a zero block size in an instruction or in the literal arguments of a primitive is implementation-dependent. Some implementations may abend, some may ignore the instruction, some may treat zero as meaning 256, others may react in yet a different way. It is permissible to specify a zero block size to those primitives where the block size is passed as data; the effect of this is defined in the description of those primitives, and is not implementation-dependent.

The unary and binary instructions (such as CLEARN and ADDN) may require temporary space. Some implementations may allocate this space in the unused part of Dynamic. Wherever the space is actually allocated, an instruction will never fail due to lack of temporary space as long as the number of free bytes of Dynamic space is at least equal to the block length of the instruction.

Most implementations should allow block sizes of up to 255 bytes to be specified in instructions, provided that this does not cause a memory violation. However, there may be some implementations that limit the

block size for some instructions and/or for some memory segments. Such implementations abend if the block size is exceeded in a particular instruction.

Condition code register

Some implementations do not change the N and V flags of the CCR. However, implementations that do change them do so exactly as shown in this document. The Store CCR primitive always writes the entire CCR, regardless of whether the N and V flags are supported specifically.

Primitives

Some primitives have an implementation-dependent effect if their literal arguments are set to a value other than those specified in this document.

Some primitives are optional: they may not be supported by all implementations. If they are supported, implementation-dependent restrictions may apply.

Although the Get Memory Reliability primitive is mandatory, some implementations may not keep track of memory reliability, and may always return the response, "memory reliable".

Some implementations may impose a limit on the number of Delegate operations that may be pending, and/or may forbid delegation that is directly or indirectly recursive.

Some implementations may impose a limit on the number or extent of writes that may be held pending while transaction protection is enabled.