INTERNATIONAL STANDARD

ISO/IEC 20060

Second edition
2010-07-01

Information technology — Open Terminal Architecture (OTA) — Virtual machine

Technologies de l'information — Architecture des terminaux ouverte (OTA) — Machine virtuelle

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Reference number
ISO/IEC 20060:2010(E)

© ISO/IEC 2010

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2010

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 20060 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 17, *Cards and personal identification*.

This second edition cancels and replaces the first edition (ISO/IEC 20060:2001), which has been technically revised.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Introduction

This International Standard specifies the Open Terminal Architecture (OTA) consistent with requirements and capabilities defined by documents [1] thru [8] in the Bibliography.

The overall architecture of the OTA is described in Annex F and is based on a virtual machine (VM) that can be programmed using high-level languages such as Forth or C. For compactness and efficiency, a tokenised form has been developed for delivering compiled programs to terminals of all CPU types. This and other virtual machine related issues are explained in Clause 5.

This International Standard defines a set of functions to be implemented in terminals in terms of instructions for a virtual machine. With these functions the application programmer is able to generate application software that is compact, portable and certifiable on all OTA terminals.

The inclusion of a function is determined by three main criteria:

— core compactness,

— execution speed,

— security requirements.

In this International Standard, the word “shall” indicates mandatory behaviour. The word “will” indicates predicted or consequential behaviour. The word “may” indicates permitted behaviour. The phrase “may not” indicates prohibited behaviour.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Information technology — Open Terminal Architecture (OTA) — Virtual machine

1 Scope

This International Standard provides the specifications for the standard Open Terminal Architecture (OTA) kernel in several layers:

— definition of the virtual machine (VM);

— description of the services provided by the VM to terminal programmers;

— specification of a set of tokens representing the native machine language of the VM;

— specification of the format in which token modules are delivered to an OTA kernel for processing.

OTA defines a standard software kernel whose functions and programming interface are common across all terminal types. This kernel is based on a standard “virtual machine,” which is implemented on each CPU type and which provides drivers for the terminal's I/O and all low-level CPU-specific logical and arithmetic functions. High-level libraries, terminal programs and payment applications may be developed using these standard kernel functions.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

None.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
aligned address
address of a memory location at which a character or cell can be accessed

NOTE The OTA virtual machine requires that aligned addresses be exact multiples of 4.

3.2
ANS Forth
programming language as defined by the American National Standard X3.215, 1994

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

3.3
big-endian
byte ordering system in which the highest-order byte of a cell is at the lowest address (i.e. appears first in a data stream)

NOTE The OTA virtual machine uses big-endian byte order in token modules and card communication.

3.4
binary
data element that is a number, to be interpreted as an unsigned integer

3.5
binary
bit string

3.6
C
C programming language

3.7
card selected services
CSS
card-resident code providing functions supporting terminal transactions, usually service functions that are used as part of a terminal selected services application

3.8
cell
primary unit of information storage in the architecture of an Open Terminal Architecture system

NOTE The standard size of a cell in the OTA virtual machine is four bytes.

3.9
compile
transform higher-level specifications of software and/or data into executable form

NOTE The executable form for the OTA virtual machine is OTA tokens.

3.10
compressed numeric
number represented in binary-coded decimal format, left justified and padded with trailing hexadecimal F's

3.11
counted string
data structure consisting of one character containing the length followed by zero to 255 data characters

3.12
data space
logical area of the virtual machine that can be accessed by Open Terminal Architecture tokens

3.13
data stack
stack that may be used for passing parameters between functions

cf. **return stack** (3.28)

NOTE When there is no possibility of confusion, the data stack is referred to as “the stack”.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

3.14
EMV
Integrated Circuit Card Specification for Payment Systems (see Bibliography [1] – [4])

NOTE EMV is managed, maintained and enhanced by EMVCo, a consortium of American Express, JCB International, MasterCard Worldwide and Visa Incorporated (see Bibliography [1] - [4]).

3.15
exception frame
implementation-dependent set of information recording the current execution state necessary for the layered exception processing used in the virtual machine

3.16
exception stack
stack used for the nesting of exception frames

NOTE It may be, but need not be, implemented using the return stack.

3.17
execution pointer
value that identifies the execution semantics of a function

3.18
implementation conformance statement[1)]
statement made by the supplier of an implementation or system claimed to conform to a given specification, stating which capabilities have been implemented

3.19
instantiate
register a local instance of a data structure with the virtual machine

NOTE At power-up in a terminal, initialised data items and kernel databases are instantiated. Further data, databases and TLV definitions may be instantiated when a module is loaded.

3.20
interpret
at run-time identify the function associated with a token value in the code and execute it

3.21
kernel
standardised set of functions mandated to be present on every terminal to implement the Open Terminal Architecture virtual machine

NOTE The kernel implementation for each CPU type is optimised for that processor.

3.22
library module
set of software functions in Open Terminal Architecture token code with a published interface, providing general support for Terminal Programs and/or Applications

3.23
LISP
family of programming languages, developed since the late 1960s, which the American National Standard working group X3J13 standardised as Common Lisp, starting 1986

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

1) For a further discussion, see ISO/IEC 9646 (all parts), *Information technology — Open Systems Interconnection — Conformance testing methodology and framework*.

3.24
module
collection of software functions and/or data compiled together and presented in token form as a package whose delivery format is specified in Clause 9 of this International Standard

3.25
module repository
non-volatile medium for storing Open Terminal Architecture modules within a terminal

3.26
non-volatile
guaranteed to be preserved across module loading or terminal power-down and rebooting

3.27
numeric
single-cell binary signed integer

NOTE The term numeric specifies that a number is represented in BCD format, right justified and padded with leading hexadecimal zeroes.

3.28
return stack
stack that may be used for program execution nesting, do-loop execution, temporary storage, and other purposes

3.29
stack
area in memory containing a Last In, First Out list of items

NOTE Stacks in the OTA virtual machine are discussed in 6.2.4.

3.30
terminal
any POS (Point of Sale) machine, ATM (Automated Teller Machine), vending machine, etc.

3.31
terminal program
overall supervisory program that a terminal executes and which contains code to select among one or more applications

3.32
terminal resident services
TRS
software in a terminal that includes terminal-specific functions, program loading functions, and the main terminal program loop defining the terminal's behaviour

NOTE TRS are usually provided by a terminal's manufacturer.

3.33
terminal selected services
TSS
terminal-independent code which implements service functions, also known as applications and libraries

NOTE The TRS main program loop selects and calls TSS functions as needed by a particular application.

3.34
token
one- or two-byte code representing a CPU-independent instruction for the Open Terminal Architecture virtual machine

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

3.35
token compiler
compiler that produces Open Terminal Architecture token modules

3.36
token loader/interpreter
software component on the terminal that processes downloaded tokens for execution on that terminal's CPU

3.37
virtual machine
theoretical microprocessor architecture

NOTE In order to provide a single standard kernel interface, all OTA terminals are coded to emulate a common virtual machine whose parameters are given in this specification.

4 Symbols and abbreviated terms

ANS	American National Standard
ASN	Abstract Syntax Notation
ATM	Automated Teller Machine
BCD	Binary Coded Decimal
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSS	Card Selected Services
DOL	Data Object List
DPB	Database Parameter Block
EMV	Europay-MasterCard-Visa
H	Hexadecimal (base 16) when used as a subscript
ICC	Integrated Circuit Card
K	1024
LSB	Least Significant Bit
MDF	Module Delivery Format
MID	Module ID
OTA	Open Terminal Architecture
PAN	Primary Account Number
POS	Point Of Sale (terminal)
RAM	Random Access Memory
RFU	Reserved for Future Use

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

ROM	Read-Only Memory
RSA	Rivest, Shamir, Adleman
SHA	Secure Hash Algorithm (SHA-1)
TLV	Tag-Length-Value data object (per ISO/IEC 8825:1990)
TRS	Terminal Resident Services
TSS	Terminal Selected Services
VM	virtual machine

5 Data types, stack notation and flags

5.1 Data Types

Table 1 — Data type designations used in OTA

Symbol	**Data type**	**Size on stack**
a-addr	Aligned address. Shall be a non-zero value exactly divisible by four.	1 cell
addr	Address. May not be zero.	1 cell
c-addr	Character-aligned address. May not be zero.	1 cell
char	Character (occupying the low-order byte of a stack item).	1 cell
d	Double-cell integer.	2 cells
dev	Device number, an integer identifying a logical I/O device supported by the kernel. A list of defined device numbers is given in Table 23.	1 cell
echar	Extended character (occupying the 2 low-order bytes of a stack item).	1 cell
flag	Flag (true or false). See Section 5.3 for conventional interpretations.	1 cell
fmt	TLV parameter, containing its format indicator and status. See Section 7 and Section 8.25 for usage.	1 cell
fn	Function code.	1 cell
ior	Result of an I/O operation. See Section 5.3 for conventional interpretations.	1 cell
len	Length of a string (0-65535).	1 cell
loop -sys	Loop-control parameters. These include implementation-dependent representations of the current value of the loop index, its upper limit, and a pointer to a “termination location” where execution continues following an exit from the loop.	Implementation dependent
nest -sys	Implementation-dependent information kept on the return stack by function calls.	Implementation dependent
num	Signed integer (when used to represent an in-line value in a token sequence, may occupy less than one cell).	1 cell
u	Unsigned integer (when used to represent an in-line value in a token sequence, may occupy less than one cell).	1 cell
ud	Unsigned double-cell integer.	2 cells
x	Unspecified cell.	1 cell
xp	Execution pointer.	1 cell

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

5.2 Stack Notation

OTA is based on an architecture incorporating push-down stacks (Last In First Out lists). The *data stack* is used primarily for passing parameters between procedures. The *return stack* is used primarily for system functions such as procedure return addresses, loop parameters, etc. The VM contains other specialised stacks that are described later.

Stack parameters input to and output from a procedure are described using the notation: (stack-id before — after)

Individual items are listed using the notation in Table 1 above.

Where an operation is described that uses more than one stack, the data stack stack-id is S:, and the return stack stack-id is R:. When there is no confusion possible, the data stack stack-id may be omitted.

If the parameters that a procedure uses or returns may vary, the options are separated by a vertical bar, indicating "or". For example, the notation (— addr | 0) indicates that the procedure takes no input and returns either a valid address or zero.

The notation i*x or j*x indicates the presence of an undefined number of cells of any data type (x); this is normally used to indicate that the state of the stack is preserved during or restored after an operation.

5.3 Flags

Procedures that accept flags as input arguments shall treat zero as false, and any non-zero value as true. A flag returned as an argument is a "well-formed" flag with all bits zero (false), or all bits one (true).

Certain device control and other functions return an *ior* to report the results of an operation. See Annex C for specific *ior* values. An *ior* may be treated as a flag in the sense that a non-zero value is *true;* however, it is not necessarily a well-formed flag, as its value may be used to convey additional information. By convention a value of zero for an *ior* implies successful completion (i.e., no error), while non-zero values may indicate an error condition or other abnormal status.

6 OTA virtual machine

6.1 General principles

The software in every OTA terminal is written in terms of a common "virtual machine." This is a theoretical microprocessor with standard characteristics that define addressing mode, stack usage, register usage, address space, etc. (see 6.2 Virtual Machine CPU below for details). The OTA kernel for each particular CPU type shall be written to make that processor emulate the virtual machine. The virtual machine concept makes a high degree of standardisation possible across widely varying CPU types, and simplifies program portability, testing, and certification issues.

The virtual machine provides the following services:

— standard CPU and instruction set, represented by OTA tokens;

— standard I/O support for common devices, with provisions for generic I/O to support additional devices that may be added;

— database management functions;

— TLV management, including format conversions and other functions;

— management of token modules, including maintaining them in storage (updating as necessary) and executing them on demand.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

These services are described in detail in Section 5 of this International Standard.

The fundamental design concepts underlying OTA technology are discussed in Annex F.

OTA provides program portability across heterogeneous terminals by treating terminal pro-grams as the intermediate code of a compiler. This code consists of streams of byte-codes, called OTA tokens. Terminals then process this code by interpreting it or by other means such as native code compilation.

It is assumed that together with an implementation of an OTA virtual machine an Implementation Conformance Statement is made available.

The OTA token set covers two areas. The first is the instruction set of a processor architecture (the virtual machine), which provides the instructions necessary for the efficient execution of programs. The second provides what are normally called “operating system functions”. In those terminals for which OTA is designed, system functions include specific functions such as I/O drivers and TLV management; and also inter-module access and access control mechanisms.

6.2 Virtual Machine CPU

The OTA virtual machine (VM) is based on a multiple stack architecture. This architecture is most commonly seen in the Forth programming language (see ANSI X3.215-1994 for the ANS Forth standard). This architecture has been further modified for portability, code density, security, ease of compilation, and for use with other programming languages. For example, it contains provisions for local variable frames used in C. Thus, OTA token compilers can be written not only for Forth but also for C and other languages.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

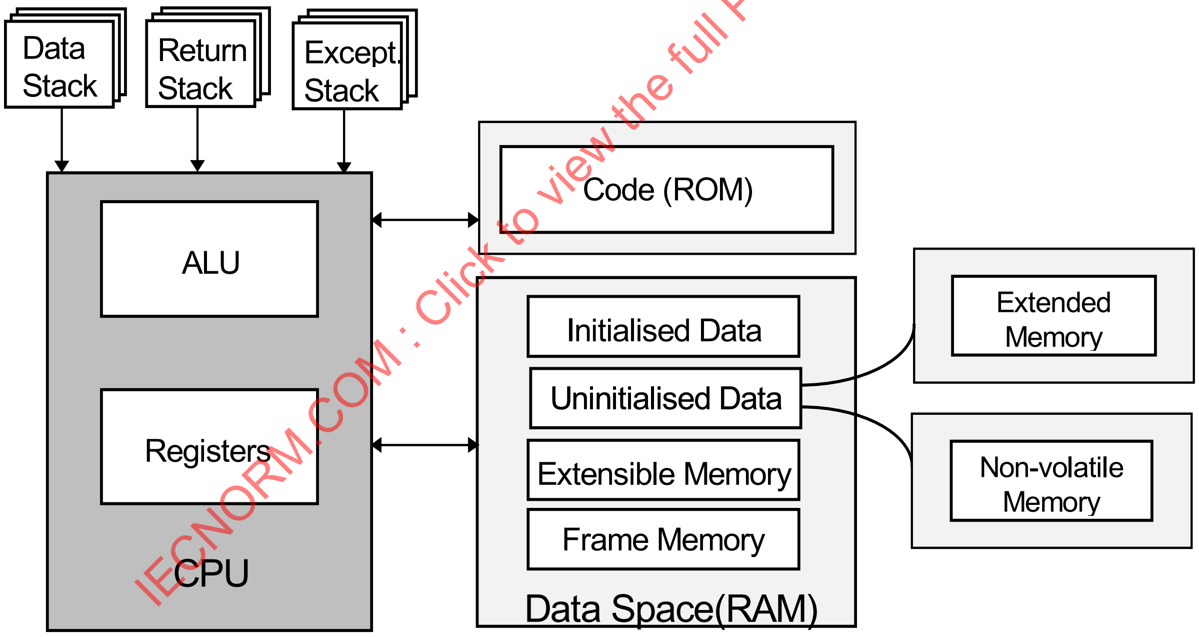


Figure 6-1 — Virtual machine architecture

The data, return and exception stacks are not in accessible memory space. Code memory is not avail-able to token programs. Extended memory and Non-volatile memory are available only through tokens providing access to buffers in uninitialised data space. See Section 6.2.3 Memory.

© ISO/IEC 2010 – All rights reserved

6.2.1 Registers

The virtual machine's registers are described in Table 2 below.

Table 2 — Virtual machine registers

Reg.	Name	Description
TP	Token Pointer	Points to the next token to be executed
DSP	Data Stack Pointer	Points to the current top of stack location. Stack space is not directly addressable.
RSP	Return Stack Pointer	Points to the current top of stack location. Stack space is not directly addressable.
FP	Frame Pointer	Points to the frame start in Data space.
FEP	Frame End Pointer	Points to the frame end in Data space.
QRR	Quote Return Register	Contains the return address for the “quote” operation.

The data and return stacks are not in the VM address space and cannot be addressed directly; they are only accessible via stack operations. No assumptions may be made regarding how data are stored physically.

The local variable frame requires two pointers to allow the traditional C calling sequences. Frames grow down in memory, matching the behaviour of the majority of CISC processor machine stacks.

The “quoting” mechanism, a technique for re-using token sequences, is discussed in Section 8.4.3 Quoting.

How VM registers are mapped onto the target CPU architecture is up to the kernel implementer for that CPU. Although these registers are conceptually present, for security no direct access is provided to them by the token set.

6.2.2 Virtual Machine Size and Cells

The OTA virtual machine is defined as a byte-addressed, two's complement, 32-bit machine, with 32-bit registers and stack elements. The register/stack size is referred to as the cell size of the virtual machine, a cell being the basic unit of manipulation on the stacks and by the virtual machine registers.

6.2.3 Memory

OTA defines a single address space available to programs. This address space shall be accessible for data storage only, and is hence referred to as “data space.” Programs may not assume that executable code is in this address space, and programs may not directly access code space.

Directly addressable data space is divided into four logical regions, each of which is individually contiguous:

1) *initialised data space,* which contains initial values specified at compile time and set when the kernel is activated and subsequently when a module containing initialised data is loaded;

2) *uninitialised data space,* which contains variables and static buffers allocated during pro-gram compilation. This data space is initialised to zero by the virtual machine;

3) *frame memory,* which is managed by the frame tokens (see Section 6.2.5 Frame Mechanism and Usage and Section 8.13 Frame Tokens).

4) *extensible memory,* which contains buffers allocated dynamically during program execution.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

There are two additional data regions, which are not directly addressable:

1) *extended memory* is used to contain TLV data and volatile databases;

2) *non-volatile memory* is used to contain data that is guaranteed by the VM to survive module loading or power-down and rebooting (within the limitations of the terminal hard-ware), including the module repository and non-volatile databases. This may be implemented in battery-backed RAM, disk, or other persistent storage.

Extended and non-volatile memory is accessed only through tokens that provide “windows” to selected data in the form of buffers in uninitialised data space.

All VM data regions may be, but are not required to be, physically separate.

6.2.4 Stacks

The data stack is used to contain parameters to procedures and temporary results from expression evaluation.

The return stack may be used by the virtual machine to contain return addresses, and also may be used for temporary storage. Application programs may only retrieve from the return stack what they have explicitly put on it inside the current procedure, and shall remove data placed on the return stack during the current procedure before exiting the procedure.

Because of the variety of implementation techniques and processor architectures, application programs may not assume that accessing the top of the return stack gives the return address of the current procedure.

Major categories of data that the virtual machine is permitted to hold on the return stack, in addition to data placed there explicitly for temporary storage, are shown in Table 3; the references cited give a fuller account.

Table 3 — Data that the VM may hold on the return stack

Category	**References**
Exception stack	Section 6.5, Section 8.16
Frames	Section 6.2.5 Frame Mechanism and Usage, Section 8.13 Frame Tokens
Loop Control	Section 8.15.3 Loop Tokens
Database context	Section 8.23 Database Services Tokens

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

All programs shall observe the following rules:

— Programs may not assume that the data in any of these categories is guaranteed to be held on the return stack, and so may use only the retrieval mechanism specified explicitly for a given category in the sections referenced in Table 3.

— Programs that pass data in one of these categories to the virtual machine shall take appropriate action to recover the data storage from the virtual machine before exiting from the procedure in which it is passed.

— Programs shall assume that any data previously placed on the return stack is rendered inaccessible by passing data in one of these categories to the virtual machine.

— Programs shall assume that any data in one of these categories passed to the virtual machine is rendered inaccessible by executing code that places values on the return stack, until such time as those values are removed.

© ISO/IEC 2010 – All rights reserved

6.2.5 Frame Mechanism and Usage

The frame mechanism allows OTA to satisfy the requirements of languages such as C that permit local variables to be defined at run time. A frame holds procedure parameters passed on the stack as well as “local” variables (temporary data storage that shall be freed automatically when the frame is released, normally at the end of procedure execution). Frame start and end pointers are maintained automatically by the virtual machine within the frame.

Virtual machine implementations are permitted to implement the frame mechanism on the return stack, and so frame usage shall conform to the rules given in Section 6.2.4 Stacks.

A frame is constructed with the MAKEFRAME or SMAKEFRAME tokens. Each of these tokens specifies the number of parameters and the number of temporary cells to allocate. For example, SMAKEFRAME 2 4 requests that a frame be built taking two items from the data stack as parameters and to allocate four cells for temporary storage. There is a two-cell overhead associated with each frame, which is used by the virtual machine implementation to save control information required by the RELFRAME token.

As an example, consider the token code sequence:

```
LIT 5
LIT 10
LIT 20
SMAKEFRAME 2 4
```

Before **SMAKEFRAME** executes, the data stack contains the values *5, 10,* and *20.* On execution of **SMAKEFRAME,** the frame is constructed using two parameter values of 10 and 20, two cells for control information, and four cells for temporaries. Thus, execution of this particular **SMAKEFRAME** token has the net stack effect (*5 10 20 – 5).*

After execution, the constructed frame appears as shown in Figure 2.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

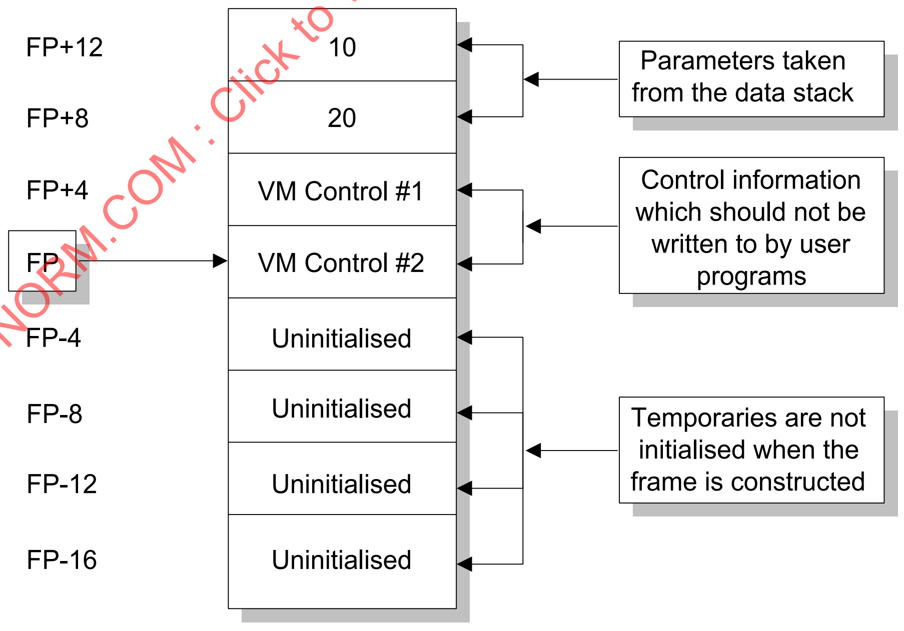

FP+12
FP+8
FP+4
FP
FP-4
FP-8
FP-12
FP-16
10
20
VM Control #1
VM Control #2
Uninitialised
Uninitialised
Uninitialised
Uninitialised
Parameters taken from the data stack
Control information which should not be written to by user programs
Temporaries are not initialised when the frame is constructed

Figure 6-2 — Frame management example

The frame pointer register, **FP,** points to the logical base of the frame. The two cells at **FP** and **FP+4** contain the control information that the virtual machine implementation uses for management of its frames. Usually these contain the value of **FP** before the frame was constructed and the address of the extent of the frame, but their exact contents are implementation-defined and a programmer should not rely on them having any specific meaning.

Parameters can be fetched from the frame using the frame access tokens (see Section 5.13). For instance the cell at **FP+8,** which is **FP+2*CELLS,** can be loaded onto the data stack using the single-byte token **PFRFETCH2;** the cell at **FP-12,** which is **FP-3*CELLS,** can be stored to using **TFRSTORE3.**

Frames with large numbers of temporaries can access them using **SFRFETCH** and **FRFETCH** tokens, where in-line offsets define the address within the frame. As an example, the cell loaded by **TFRFETCH3** can also be loaded using the token sequence **SFRFETCH -12.**

There is no requirement for the virtual machine to make two or more frames contiguous in memory; the only requirement is that cells within the frame shall be contiguous.

6.2.6 Extensible Memory

The virtual machine provides a dynamically allocated pool of memory as a single extensible buffer appearing in the program's uninitialised data space. Programs may request an allocation of a specified amount of memory and are returned a base address for that memory. Subsequently programs may release memory from a given address, causing *all* allocations beyond that address to be released. The tokens used to manage extensible memory are found in Section 5.14.

All tokens except the extensible memory management tokens EXTEND, CEXTEND, and RELEASE and the exception handling tokens THROW and QTHROW are required to have no net effect on the extensible memory pointer. If a token allocates extensible memory it shall also release it, including any effect of cell-alignment. Successive allocations of extensible memory shall be contiguous within a module but are not guaranteed to be contiguous between modules, except that inter-module calls using IMCALL or DOSOCKET shall preserve contiguity. An automatic release of dynamically allocated memory shall occur when a module's execution is completed, limiting the effects of program failure to release memory cleanly; see Section 4.8. In addition, if a THROW occurs, the allocation of dynamically allocated memory will be restored in some cases to its condition at the time of the governing CATCH (see Section 6.5).

Exception Handling and Section 8.16 Exception Tokens).

6.2.7 User Variables

User variables are cell-sized variables in which the virtual machine holds contextual information for the client programs. Storage for user variables is pre-allocated by the virtual machine. Sixteen variables (referenced as 0 to 15) are provided. The user variables listed in Table 4 are defined for use by the virtual machine and shall therefore be used only for the purposes described.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Table 4 — User variables in the virtual machine

Offset	Name	Description
0	BASE	The number base used for conversions between numbers and textual strings of numeric digits.
1	DEVOP	Current output device number.
2	DEVIP	Current input device number.
3	DBCURRENT	Pointer to the current Database Parameter Block (see Section 7.3.1 The Database Parameter Block).
4	DBRECNUM	Current record number in the current database.

© ISO/IEC 2010 – All rights reserved

6.3 Virtual Machine Execution Features

Code written to run on the virtual machine (including Terminal Resident Services compiled as token modules) may assume that following power-up the terminal-specific kernel software supporting the VM has performed any necessary terminal-specific power-up initialisation, and has started execution of the main processing loop of Terminal Resident Services (TRS) through a module loading process described in Section 7.9 Module Handling Services. On entry to the beginning of Terminal Resident Services, conditions listed in Table 5 are guaranteed. The action taken by the terminal-specific kernel software if it does not have resident Terminal Resident Services for the main Terminal Program loop is outside the scope of this specification.

Table 5 — Initial condition of the VM on entry to the TRS

Resource	**Condition**
Data Stack	Any specific contents explicitly enumerated.
Return Stack	Specified depth for this terminal is available.
Frames	No current frame
Extensible Memory	Specified amount for this terminal is available.
Devices	All devices are closed.
Current Output Device	DEVOP = 1
Current Input Device	DEVIP = 0
Number conversion base	BASE = 10 (decimal)
Sockets	Socket zero contains a procedure enabling all sockets to be plugged; all other sockets contain no-ops (see Section 8.22 Socket Tokens).
Current database	DBCURRENT = 0 (invalid address)
Current database record	DBRECNUM = -1 (no record selected)

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

If the main processing loop of the TRS is exited, control shall return to the terminal-specific layer of software responsible for reloading the TRS and re-entering its main loop. All VM resources shall be released whenever the TRS exits, except for data in non-volatile databases. Resource releasing occurs when the terminal is powered down, the TRS exits, or the TRS is restarted by the terminal's Operating System (if any). If an updated version of any TRS module has been acquired since the TRS main loop was last entered, all TRS resources *including* data in its non-volatile databases shall be released when it exits. See Section 7.9 Module Handling Services and Section 8.28 Module Management Tokens for details of the module management services provided by the virtual machine. See Section 9.1 Module ID Format for the ID of the start-up module called from the virtual machine.

6.4 Arithmetic

Addition and subtraction operations are performed modulo 2^{32}.

Store operations whose destination storage is smaller than the size of the value passed shall store the value truncated to the width of the destination. In the case of a string, this shall cause the last bytes to be discarded; in the case of a binary integer, it will result in an incorrect value.

Division operations are *symmetric;* that is, rounding shall always be towards zero regardless of sign.

Division overflow may cause exception -11 (Result out of range) to be thrown.

6.5 Exception Handling

OTA provides a single exception handling mechanism via the tokens **CATCH, THROW** and **QTHROW** (see Section 8.16 Exception Tokens). These tokens are derived from the Lisp exception handling mechanism, and appear in ANS Forth as **CATCH** and **THROW.** Exception codes are defined in Annex B.

The purpose of this mechanism is to provide for local handling of exceptions under program control at various levels in the software. The concept is that the program passes a function's execution pointer to the token **CATCH,** which will execute that function and return a code indicating what, if any, exception occurred during its execution. **CATCH** records implementation-dependent information sufficient to restore its current execution state should a **THROW** occur in the function passed to **CATCH** for execution. This includes (but is not limited to) data and return stack depths, the frame pointer and, in some cases, the extensible memory pointer. The collection of information representing an execution state is referred to as an “exception frame.” Exception frames are kept on the exception stack.

Following a **CATCH,** the program can examine any exception code that may have been returned, and elect to handle it locally or **THROW** it to a higher level for processing.

The virtual machine shall provide a default outermost level at which exceptions will be trapped. This outermost level will be activated when no inner level of **CATCH** has been established. The default exception handler shall abort any current terminal transaction and attempt to reload TRS modules and re-enter the TRS main loop (see Section 7 System Services for details of standard TRS processing facilities to be provided by all terminals).

The virtual machine is mandated to throw the general exception codes -10 (Division by zero), -17 (Pictured numeric string overflow) and -21 (Unsupported operation) should these conditions occur. For many of the tokens specified in Section 8 Token Set Definition, general exception codes defined in ANS Forth and listed in Table 6 may apply but are not mandatory. The virtual machine is entitled to throw these codes under the given conditions, and they are not listed individually under those tokens. Note that all optional exceptions not implemented by a specific virtual machine have to be listed in the Implementation Conformance Statement.

The virtual machine is mandated to throw all the OTA-specific exceptions which are International Standarded for the individual tokens and which are listed in Table B.2 — OTA THROW codes of Annex B.

Table 6 — Optional general exceptions from ANS Forth

Code	Condition	Origin
-3	Stack overflow	Token that adds to number of cells on data stack; data stack full.
-4	Stack underflow	Token that references one or more items on data stack; referenced item below stack base.
-5	Return stack overflow	Token that adds to number of items on return stack; return stack full.
-6	Return stack underflow	Token that references one or more items on return stack; referenced item below return stack base.
-9	Invalid memory address	Token that accesses or branches to memory at a given or computed address; address outside code or data space.
-11	Result out of range	Token involves arithmetic operation other than addition or subtraction; result overflows specified return value.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

-23	Address alignment exception	Token that references data at an address specified by a-addr; address is not cell-aligned
-24	Invalid numeric argument	Value outside prescribed range, such as a len larger than 65535.
-53	Exception stack overflow	Nested **CATCH**es used too much exception stack space.

6.6 Resources

The virtual machine provides resources to programs to support the architectural features described above, and in support of the System Services described in Section 4 and specific functions detailed in Section 8 Token Set Definition. These resources are identified in Table 7. Each VM implementation shall International Standard the amount of each resource in this table that it will provide to application programs, and each OTA program shall International Standard the amount of each resource in this table that it requires from a VM. Note that the "extensible memory space" requirement in Table 7 is exclusive of the other specific data areas cited in the table; this data space is for general usage.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Table 7 — Virtual machine resources

Resource	Units
Extensible memory space	Bytes
Data stack	Cells
Return stack	Cells
Exception frames	Frames
Procedure call nesting	Calls
Frame space including exception frames	Bytes
Number formatting scratchpad size	Characters
Scratchpad buffer for compressed numeric strings	Bytes
Module storage space	Bytes
Number of stored modules	--
Non-volatile database storage space	Bytes
Volatile storage space for databases, and TLV data	Bytes
Hot card list data storage	Entries
Number of user variables	--
Number of languages supported	--

6.7 Programs and Tokens

The instruction set of the virtual machine is coded as a byte stream of tokens. Single-byte tokens are referred to as *primary* tokens; these refer to primitive instructions such as are commonly found in any instruction set. Multi-byte tokens are referred to as *secondary* tokens, and are used for less-frequently-used services. The complete token set for the OTA virtual machine is described in detail in Section 8 Token Set Definition.

A *token compiler* compiles source code into a string of tokens, some of which may be followed by in-line data. This token string is separately combined with other data needed by the program and a header, and encapsulated in a module delivery file that may be downloaded to the target terminal. See Section 9 Module Delivery Format for a detailed description of a module format.

After a module is downloaded, it is stored in a module repository. When its function is required, the terminal uses a token loader/interpreter to process the tokens for execution on the terminal's CPU. This process consists of executing the function associated with each token.

7 System Services

The OTA software on a terminal can be divided into four major categories:

— *The Kernel,* which includes machine-dependent implementations of I/O drivers and all functions required in this specification for supporting applications.

— *Terminal Resident Services* (TRS) include terminal-specific functions, libraries supporting these functions, program loading functions, and the main loop defining the terminal's behaviour.

— *Terminal Selected Services* (TSS) include service functions, also known as applications, and libraries supporting these functions. TSS contain only terminal independent code resident on the terminal. The TRS main program loop will select and call TSS functions as needed by a particular application.

— *Card Selected Services* (CSS) include functions supporting terminal applications, e.g. payment service functions that are used as part of a TSS application. CSS are resident on an ICC and downloaded to a terminal as requested. For terminals with two ICC readers (e.g., one for normal transactions and one for maintenance) there may be two independent sets of CSS (CSS1 and CSS2).

The kernel and TRS contain all of the terminal-dependent code and are usually provided by a terminal vendor. The terminal vendor software may include additional features that are out-side the scope of the OTA specification.

All software on an OTA terminal above the VM kernel is organised as a set of separate *modules.* The fundamental characteristic of a module is that it is a collection of definitions or pro-gram functions that have been passed through a token compiler and encapsulated as a single package for delivery to a terminal. The main Terminal Program (TRS), each application, each library, and each CSS download are examples of modules.

All modules use the Module Delivery Format discussed in Section 9.

The kernel in an OTA system provides a variety of high-level services to these modules. The basic principles underlying these services, and their rules of usage, are described in this section.

7.1 Time Handling

Two timers are provided by the virtual machine. The first is a milliseconds timer, which is used by programs for delays. It uses the full 32-bit range and wraps from 2^{32}-1 to zero. It is guaranteed to be updated at least every 100 milliseconds. The second timer provides a calendar function providing date and time to the nearest second. Timer handling functions are International Standarded in Section 8.17 Date, Time, and Timing Tokens.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

7.2 Devices and I/O Services

This section describes the general principles of device access and control under OTA. Details of specific devices are given in Annex C.

Each device, including those devices whose lower-level operation is hidden by the virtual machine behind device-specific functions, is assigned a device type (used to categorise result codes) and a unique device number. Device numbers are arbitrary; however, references to device numbers -1 through 15 may be made with only a single token, and so these are assigned to the most common devices (see Annex C). It is not required that all these devices be available on all terminals.

General I/O facilities are provided by functions taking the device number as an input parameter. These functions are described in Section 8.18 Generic Device I/O Tokens. Some of the devices like the ICC and magstripe card readers have specific tokens assigned. To be able to deal with ISO/IEC 7816-4 compatible ICC cards in a protocol independent way (T=0 or T=1), the token **CARD** shall be used. The format of the input and output buffers is defined as the ISO/IEC 7816-4 APDU command response message structure. On the ICC card reader device the tokens **DEVREAD, DEVTIMEDREAD** and **DEVWRITE** may be used for cards that are not ISO/IEC 7816-4 compatible.

The same holds for the magstripe card reader where the generic **DEVREAD, DEVTIMEDREAD** and **DEVWRITE** tokens have been replaced by **MAGREAD** and **MAGWRITE** for ISO/IEC 7813 compatible magstripe cards. The **DEVREAD, DEVTIMEDREAD** and **DEVWRITE** tokens should only be used for magstripe cards that are not ISO/IEC 7813 compatible. Note that for a given terminal implementation **DEVREAD, DEVTIMEDREAD** and **DEVWRITE** applied on the ICC card reader and magstripe devices, may throw -21 (Unsupported operation), which may happen as well for the magstripe device, when **MAGWRITE** is used for tracks other than track 3.

General result codes (ior codes) defined for use with I/O functions are given in Annex C. A device access procedure may return an ior and may also **THROW.** A procedure that does return an ior may not use the same code as a **THROW** code; however a higher-level procedure may elect to do so.

The guideline for usage of ior codes and **THROW**s is that a procedure should **THROW** on a global error, but return an ior for situations that are correctable at or near the device level. Two **THROW**s are appropriate for all device access procedures:

— If a procedure is called with a device number that does not exist on the terminal, exception -32763 (Unsupported device) shall be thrown.

— If a procedure is called with a valid device number but an invalid function for that device, exception -21 (Unsupported operation) shall be thrown.

The following **THROW**s may also be appropriate in many cases:

— If a procedure takes memory address(es) as parameter(s) and an invalid address is detected, exception -9 (Invalid memory address) should be thrown.

— If a procedure takes numerical value(s) as parameter(s) and an invalid value is detected, exception -24 (Invalid numeric argument) should be thrown.

All other device procedure errors should be returned as ior codes for those device procedures that use ior codes.

7.3 Database Services

An OTA terminal will contain at least three major databases: an application-specific transaction log, a database of messages in one or more languages and a database of modules. Generic database support for these and other functions is described in this section.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

The virtual machine provides a mechanism for handling databases that conceals implementation details (handled by the virtual machine as "server") from the application software (the "client"). The services implement the following features:

— At any given time the client has access to one currently selected database **(DBCURRENT)** and one currently-selected record number **(DBRECNUM)** that is shared across all defined databases.

— Information about each database is shared between client and server through a Database Parameter Block (DPB), which the server can access for reading and writing. The client "owns" the DPB, in the sense that it is in the client's data space; but the client is not allowed to access it directly. Instead, specific methods are used as described in Section 8.23 Database Services Tokens.

— The address of a window onto the current record (a record buffer) is provided by the server to the client for each client database; for certain database operations the client may pass the addresses of strings and key buffers to the server. For each database that has been made known to the server by a client module, a record buffer shall be provided by the VM. This buffer starts at an aligned address. The content of the record buffer associated with a particular database after a record selection remains available until the client selects another record from that database and should be considered to be read-only. If changes are made to the contents of the record buffer without using a database store token **(DBSTORE, DBSTRSTORE,** etc), the terminal may throw a -9 (Invalid memory address) or those changes may be lost when any database access token is executed. Any fetch from or store to a database record will cause the selected database record to be read from the database and placed into the record buffer. In this manner, any changes previously made to the data-base buffer will be lost and replaced with the contents of the record as held in database. Therefore, the database record always reflects the actual contents of the record as held in the database after a database fetch or store is performed. Except for these record buffers, compiled databases, and parameters passed on the stack by specific database functions, no other data space is shared between client and server. Programs may not assume that records in a database are contiguous in memory.

— A database is instantiated by the loading process for the module in which its DPB is defined. Volatile databases installed by application modules are uninstantiated automatically and transparently by the server when the application is terminated by Terminal Resident Services, when all server-allocated data storage relating to those databases is released.

— The server shall delete non-volatile databases when the module that defined them is replaced. If the module is loaded (as described in Section 7.9 Module Handling Services) when replaced, e.g., in the case of a TRS module, the server shall delete the module's non-volatile databases when the module is unloaded.

Figure 3 below illustrates this sharing of data space.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

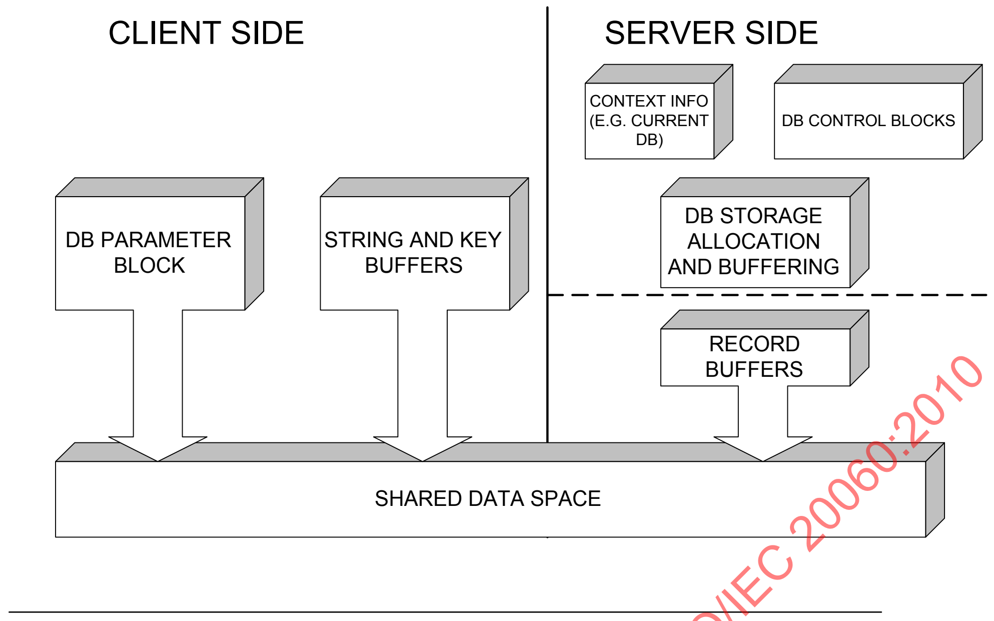


Figure 7-1 — Database memory access

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

The database model allows for databases whose storage requirements are maintained by the server, as well as databases that exist as pre-set data structures in the client's address space. It distinguishes between:

— *volatile* databases (those whose contents will be lost on module unload or a power-down condition) and *non-volatile* databases (those whose contents shall be preserved across module loads or a power-down condition);

— *non-ordered* and *ordered* databases (ordered by a client-defined key);

— *read-only* and *read/write* databases.

This database model also provides for future expansion to other database attributes and facilities.

7.3.1 The Database Parameter Block

The DPB for a database shall be created with the following structure, where each field is one cell in size. All databases have fields 0-4; fields 5 and 6 exist only for ordered databases.

Table 8 — DPB Structure

Cell Offset	Field
0	DPB link
1	DB pointer
2	DB type
3	Record size
4	Available (next available record number)
5	Key offset
6	Key length

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

All information to specify a database shall be preset in the DPB. Client software may not make any subsequent direct access to the DPB and shall make no assumptions about the values directly held in the DPB after the module defining that DPB has been loaded for execution (see Section 8.23 Database Services Tokens). The list below provides a detailed description of each of the DPB fields.

DPB Link Database Parameter Blocks are passed to the token loader/interpreter in the form of a pointer to a linked list in the initialised data section of a module. This field shall be preset to the address in initialised data of the next DPB in the list; or zero if this DPB is the last or only DPB in the list.

DB Pointer For compiled databases that exist in the client's initialised data space (see “DB Type” below), this field shall be preset to the “origin” address in initialised data. For databases whose storage is controlled by the server, the field shall be preset to zero.

DB Type This cell is organised into two subfields, as follows, where “bit 0” designates the least significant bit of the cell:

Bits 24-31	Bits 0-23
DB attributes	DB kind

At present four **DB attributes** (bits 28-3 1) are defined:

Bit 31 = 1	Database is ordered, and the DPB contains cells 5 and 6 (key specifiers).
Bit 31 = 0	Database is not ordered.
Bit 30 = 1	Database is read-only. Any write operation shall cause Exception -4094 (Invalid function) to be thrown.
Bit 30 = 0	Database can be read from or written to.
Bit 29 = 1	Database is optimised for linear search order. Kernel implementations are not required to support this attribute.
Bit 29 = 0	Database is optimised for random search order. Kernel implementations are not required to support this attribute.
Bit 28 = 1	Database has to be stored in secure memory. Kernel implementations are not required to support this attribute. If a module is loaded that requires this feature in a database and the kernel does not support this attribute, the module loading will fail and return *false*.

© ISO/IEC 2010 – All rights reserved

Bit 28 = 0	Database has no security requirements on storage.
Bits 24-27	are reserved for further non-exclusive attributes to be defined in the future.

DB kind holds a single numeric value that defines an exclusive characteristic of the database, with values defined as follows:

0 “Volatile” database whose content does not need to be preserved between module loads or across a power-down of the terminal on which it resides.

1 “Non-volatile” database whose content shall be preserved between module loads or across a power-down. If the module defining a non-volatile database is replaced, the database is destroyed when the old module is unloaded.

2 (Reserved for future use.)

3 “Compiled database” constructed by the compiler in a contiguous area of initialised data as fixed-length records. Records may not be added to or deleted from a compiled database, and the database may not be initialised at run-time, but otherwise full read-write capability shall be sup-ported.

4 Reserved for optional internal use by kernel implementations (e.g. in TLV management functions). Token programs may not refer to or make any assumptions about this kind.

(Values 5 and up are reserved for future use.)

Record Size	This field shall be preset to the size of the record in bytes.
Available	For compiled databases, the compiler shall set this field to one plus the last allocated record number in the database. For any other database, the compiler shall set this field to zero.
Key Offset	If ordering is required on the database **(DB type** bit 31 = 1), this field shall be preset to the byte offset within the database record at which the key begins.
Key Length	If ordering is required on the database **(DB type** bit 31 = 1), this field shall be preset to the size in bytes of the key within the database record.

7.3.2 Database Instantiation

The action taken when a database is instantiated at module load time depends on the value of **DB type** and **DB pointer** in the DPB, and whether the database is volatile or non-volatile.

If the database is a non-volatile type, the DPB address is used in conjunction with the Module ID to identify any prior data that belongs to the database. If prior data exists, the next available record number shall be restored to its previous value. Otherwise, the server instantiates new non-volatile storage space and sets the next available record number to zero. In both cases a buffer shall be provided for the current record in the database.

If **DB pointer** is zero and **DB type** is not a compiled type, then the server instantiates or makes available the storage required for the database, initialises the storage to all zeros, provides a buffer for the “current record” of the database, and sets the “available” record number of the database (see `DBAVAIL)` to zero.

If **DB pointer** is non-zero and **DB type** is a compiled type, then the server sets up internal structures to make use of the client data structure whose origin address has been passed at **DB pointer** and sets the “available” record number (see `DBAVAIL)` to the value passed in the **Available** field of the DPB.

Exception -4093 (Database creation error) shall be thrown for all other conditions on instantiation.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

7.3.3 Database Exception Handling

The **THROW** codes defined for handling exceptions detected by the database tokens are described in Annex B, "OTA Throw Codes", -4095 through -4087. Codes -4093 through -4088 apply only to non-volatile databases.

7.4 Language and Message Handling

Language selection, message selection and message display are provided by a set of dedicated tokens International Standarded in Section 8.24 Language and Message Tokens. Up to 255 messages are accessed by message numbers. An example for organisation is shown in Table 9.

Table 9 — Messages, by number and origin

Message numbers (hex)	Typical Origin
1 -3F	Terminal (TRS provider)
40-BF	Terminal (Application provider)
C0-FF	ICC

NOTE This organisation reflects the needs given in [4]

The terminal maintains an internal language database providing access to up to 255 messages (numbered 1 - FF$_H$) in one or more languages. Messages in this range for a current or new language are installed by **MSGUPDATE** and deleted by **MSGDELETE.**

Messages may be provided by an application or ICC using the token **MSGLOAD,** which puts them in a transient buffer. This buffer is cleared by **MSGINIT.**

Messages installed in the terminal's internal database by **MSGUPDATE** and selected by the token **CHOOSELANG** are the default messages returned by **MSGFETCH.** If messages in a particular message number range have been installed in the transient buffer by **MSGLOAD, MSGFETCH** returns these instead of the corresponding default messages. In case of overlap-ping **MSGLOAD** operations, the more recently installed messages replace previously installed ones. If no message has been installed for a particular message number, **MSGFETCH** returns a zero-length string.

The format of a message table processed by **MSGUPDATE** or **MSGLOAD** is as follows:

Symbol Two-byte string giving the ISO 639 code for the language. **Code Page** One-byte ISO/IEC 8859 code for the character set used by these messages.

Code Name Counted string giving the name of the language in Code Page-specific characters.

ASCII Name Counted string giving the name of the language in 7-bit ASCII characters.

Number 1 byte message number.

Message Counted string which holds the characters for the message.

The Number and Message fields repeat for each message within this table. After the last message, a final one-byte long message with a 0 (null) as its first and only character signals that the message table is complete.

The format of the accessible fields in the internal **LANGUAGES** database is given in Table 10.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Table 10 — Message table format

Item	Size (bytes)	Description
Symbol	2	ISO 639 code for the language.
ASCII Name	16	Name of the language in 7-bit ASCII characters padded with trailing spaces.
Code Name	16	Name of the language in Code Page-specific characters padded with trailing spaces.
Code Page	1	ISO/IEC 8859 code for the character set used by these messages.

7.5 TLV Services

The TLV data format is described in ISO/IEC 8825 (1990). The general term for this data is Basic Encoding Rules — Tag, Length, Value (BER-TLV, or just TLV).

7.5.1 Basic Principles

TLV data objects consist of two or three consecutive fields: a *tag* field specifying its class, type and number, a *length* field specifying the size of the data, and if the length is not zero, a *value* field containing the data. Because ICC card responses are limited to 255 bytes or less, this sets the maximum size of a TLV object in this system. The tag field is one or two bytes, the length field is one or two bytes, and thus the maximum size of the value field is 252 bytes (a field this long requires two length bytes, as explained below).

The first byte of the *tag* field is broken into three fields. Bits 7 and 8 specify the class of the object. Bit 6 determines if the value field contains "primitive" data or if it is a "constructed" object consisting of other TLV-encoded fields. Constructed objects are also called *templates.* They cause their value fields to be parsed for TLV sequences when they are encountered. Bits 1 to 5 specify the number of the object or, if all these bits are set, they indicate that additional tag bytes follow. The additional tag bytes have their eighth bit set if yet another byte follows. All bits in up to two bytes are used to determine a tag name. A tag field containing more than two bytes is undefined in OTA, and will result in an ambiguous identification.

The *length* field consists of one to three consecutive bytes (not more than two are needed in this implementation). If bit 8 of the first byte is 0, then bits 1 to 7 indicate the size of the value field. If bit 8 of the first byte is 1, then bits 1 to 7 indicate the number of bytes that follow. The following bytes, if any, indicate the size of the value field and occur with the most significant byte first.

The *value* field can consist of "primitive" data or be "constructed" with additional TLV encoded sequences. If bit 6 of the first byte in the tag field is set, then the value field contains additional TLV sequences. The primitive objects can be encoded in several different formats: Binary Coded Decimal nibbles with leading zeros or trailing nibbles with all bits set, binary numbers or sequences of bytes, character sequences of alpha/numeric or ASCII bytes, or undefined formats. Each is handled differently as it is used.

An ICC may also use a Data Object List (DOL) to request values of specified tag names. The card sends a DOL consisting of a list of tag and length fields, and the terminal returns the corresponding value fields, without delimiters.

7.5.2 TLV Definitions

Each TLV that will be used shall be defined by the terminal or application programs to establish its data type and name. Since the terminal program and the application programs are developed separately, OTA uses a linked structure (a balanced binary tree) to allow rapid addition and removal of tag names from the global tag name list. This requires that the following structure be compiled for each TLV in initialised data space in the module defining the TLV:

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Link	A cell with "left" (high-order two bytes) and "right" (low-order two bytes) components providing links to elements of the tree.
Link left	A 16-bit signed offset from this TLV's access parameter to the access parameter of a TLV record whose tag is numerically less than this record's tag. A value of zero indicates that this TLV is not linked to a TLV with a tag numerically less than this one.
Link right	A 16-bit signed offset from this TLV's access parameter to the access parameter of a TLV record whose tag is numerically greater than this record's tag. A value of zero indicates that this TLV is not linked to a TLV with a tag numerically greater than this one.
Tag	A two-byte string whose big-endian numeric value is the TLV tag.
Type	A single byte that specifies control information. The byte contains one of the following data format indicators: 0 = BCD nibbles padded with leading zeros, 1 = Sequence of bytes, application defined, 2 = Binary number, most significant byte first, 3 = BCD nibbles padded with trailing Fs, 4 = Alpha/numeric characters with trailing zeros, 5 = Full ASCII character set with trailing zeros, or 6 = Variable format.
Reserved	A byte that shall be initialised to zero by the compiler.
Data	A cell that holds VM-specific information including access to the length and value fields of this TLV. This field shall be initialised to zero by the compiler.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

The system shall also maintain a status byte for each TLV (see Section 8.25.3 TLV Sequence Access). This may be the Reserved byte in the above structure. The low-order bit of this byte shall be set if the TLV has been assigned a value as a result of being in a sequence that has been processed by **TLVPARSE** or **TLVSTORE.** The purpose of maintaining an assigned status is to identify TLV values that contain valid data (which may be zero) and distinguish them from TLV values that have never been set and are therefore invalid.

The OTA kernel manages a global list of TLV tags by maintaining a list of pointers to the initialised data space containing their actual definitions as described above. When a module is loaded, its TLV definitions are added to this list as part of its initialisation; when it is unloaded, its TLV definitions shall be removed from the list automatically by the virtual machine. If the module contains a TLV definition that already exists, the Load Module sub-routine will fail and return *false.* The address of the **Link** field described above is returned as the "access parameter" for TLV references. The programmer should not access these fields directly, nor make any assumption about their contents, after the VM has instantiated the TLV definitions.

The OTA tokens that support these TLV services are described in Section 8.25 TLV Tokens.

7.5.3 TLV References

References to TLV definitions in the source code are compiled as either direct references to the definition structures defined above, or numerical tag values. Within certain binary TLV definitions, individual bits or groups of bits are defined to have certain meanings. These are referred to in OTA as "TLV bits". References to TLV bits may be compiled with a literal pointing to a bit within the value field of the TLV. Bit 0 is the least significant bit of the first byte, bit 7 is the most significant bit of that same byte, bit 8 is the least significant bit of the second byte and so forth.

TLV types 0 and 2 are translated into 32 bit stack values for internal manipulation, and both have size constraints that shall be observed to make this possible. Type 0 is encoded as BCD digits and thus its maximum *value* is limited to about 9.5 decimal digits, or, more precisely, 4,294,967,295. Type 2 is encoded as binary bytes and thus is limited to the size of 1 cell (4 bytes). In both cases, exceeding these limits shall cause values to lose their most significant portions. Type 3 values are encoded as decimal digits that are too long to be held in one cell, and thus are converted to strings of digits. This string is created in a temporary location that shall be moved to a more permanent location immediately after it is accessed. The string parameters returned by the other TLV types do not have any persistence constraints.

Only one TLV can be accessed at a time. In the OTA implementation, the maximum size of the value field in a TLV is 252 bytes.

The data assigned to a TLV definition is exposed to the application through a 252-byte scratch area maintained by the VM. The application program is permitted to change the con-tents of this scratch area. If changes are to be retained, an address and length within the scratch area shall be passed back to the **TLVSTORE** token. The address and contents of the scratch area may become invalid when any TLV token is subsequently executed.

7.6 Hot Card List Management

Hot Card List management is provided by a set of dedicated functions that are specific to the management of a large hot card list. A typical list may contain 10,000 PAN entries of up to 10 bytes each, or 20 BCD digits. The PAN entries are stored in compressed numeric (cn) for-mat, right padded with hexadecimal F_H's. As a PAN is a maximum of nineteen BCD digits, an entry in the list will always be padded with at least one F_H. When searching in the hot card list, F_H's in a list entry are considered as wild cards or “don't care” digits, but any F_H's in the PAN used as input are not wild cards. Wild cards can only appear at the right-hand end of an entry. A PAN shall be considered found in the hot card list if there is a list entry that is identical up to the first F_H in the entry.

For example, assume the following Hot Card list in the terminal:

5413278000404808FFFF

5413278000404763FFFF

3625FFFFFFFFFFFFFFFF

45066367FFFFFFFFFFFF

The following input PANs would be found in the list:

5413278000404808

4506636700422497

362567810001

3625

The following input PANs would not be found in the list:

541327800041234F

5413

45066F

Specific tokens are International Standarded in Section 8.26 Hot Card List Tokens. Tokens are provided to initialise the hot card list to an empty state, add entries, delete entries, and find entries.

7.7 Cryptographic Services

This chapter describes a number of cryptographic and arithmetic support functions provided by the VM. The algorithms are selected using a code from Table 11 as the top-of-stack argument to the token **CRYPTO** (see

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Section 8.27 Cryptographic Algorithm Token). Exception -21 (Unsupported operation) shall be thrown if a particular function is not supported by the VM. The algorithms are further described below.

Table 11 — Cryptographic algorithm codes

Value	Description
1	Modulo multiplication
2	SHA-1
3	Modulo exponentiation
4	Long Shift
5	Long Subtract
6	Incremental SHA- 1
7	Cyclic Redundancy Check (CRC)
8	DES Key Schedule
9	DES encryption/decryption
>9	RFU

7.7.1 Modulo Multiplication

This function performs a multiplication of two unsigned values *x* and *y*, where the product is reduced using the modulus *z*. The formula is:

result = mod(x*y,z)

The input values (x,y,z) have all the same length. They are represented by byte strings (*c-addr len*) and can be any multiple of 8 bits up to and including 1024 bits. The values shall be in big-endian byte order.

The stack parameters for the **CRYPTO** token in this case are as follows: (*c-addr*$_1$ *len*$_1$ *c-addr*$_2$ *len*$_2$ *c-addr*$_3$ *len*$_3$ *c-addr*$_4$ *1* — *c-addr*$_4$ *len*$_3$)

where *c-addr*$_1$ *len*$_1$ and *c-addr*$_2$ *len*$_2$ are the values *x* and *y* for the multiplication, and *c-addr*$_3$ and *len*$_3$ is the modulus *z*. The result is stored in *c-addr*$_4$. There shall be enough room at *c-addr*$_4$ to store *len*$_3$ bytes. The buffer *c-addr*$_4$ *len*$_3$ is returned as a result.

Note that *c-addr*$_1$, *c-addr*$_2$ and *c-addr*$_3$ may be the same, and *len*$_1$, *len*$_2$, and *len*$_3$ shall be the same. The buffer *c-addr*$_4$ *len*$_3$ shall not overlap with any of the input buffers.

7.7.2 Secure Hash Algorithm (SHA-1)

This algorithm is standardised as FIPS 180-1. SHA- 1 takes as input messages of arbitrary length and produces a 20-byte hash value. The stack parameters for the **CRYPTO** token in this case are as follows:

(*c-addr*$_1$ *len*$_1$ *c-addr*$_2$ *2* — *c-addr*$_2$)

where *c-addr*$_1$ *len*$_1$ is the input buffer for computation and *c-addr*$_2$ is the buffer containing the result, which is always 20 bytes in length.

The 20-byte result buffer *c-addr*$_2$ shall not overlap the input buffer *c-addr*$_1$ *len*$_1$.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

7.7.3 Modulo Exponentiation

This function raises an unsigned value *x* to a power given by an unsigned exponent *y*, where the product is reduced using the modulus *z*. The formula is:

result = mod(*x*^*y*,*z*)

The input value *x* and modulus *z* are represented by byte strings *(c-addr len)* and may be any multiple of 8 bits up to and including 1024 bits. The values shall be in big-endian byte order.

The stack parameters for the **CRYPTO** token in this case are as follows:

(*c-addr*$_1$ *len*$_1$ *u c-addr*$_2$ *len*$_2$ *c-addr*$_3$ *3 — c-addr*$_3$ *len*$_2$)

where *c-addr*$_1$ *len*$_1$ is the argument *x*, *u* is the exponent *y*, and *c-addr*$_2$ *len*$_2$ is the modulus *z*. The result is stored in *c-addr*$_3$, and the buffer *c-addr*$_3$ *len*$_2$ is returned as a result. It shall not overlap with any of the input buffers. There shall be enough room at *c-addr*$_3$ to store *len*$_2$ bytes.

7.7.4 Long Shift

This function shifts a binary string by *n* bits. If *n* is positive the string is shifted *n* places to the left, and the least significant bits are filled with zero. If *n* is negative, the string is shifted |*n*| places to the right, propagating the most significant bit. If |*n*| is greater than the length of the string, the returned string is filled with zeros (*n*>0) or filled with the input's most significant bit (*n*<0).

The input value is represented by a byte string *(c-addr len)* and may be any multiple of 8 bits up to and including 1024 bits. The value shall be in big-endian byte order.

The stack parameters for the **CRYPTO** token in this case are as follows: (*c-addr len num 4 — c-addr len*) where *c-addr len* is the starting address and length of the number to be shifted and *num* is the number of places to shift (left if *num* > 0, right if *num* < 0). The result is returned at the same *c-addr len.*

7.7.5 Long Subtract

This function subtracts two numbers. The input values are represented by byte strings *(c-addr len)* and may be any multiple of 8 bits up to and including 1024 bits. Both inputs shall be the same length. The values shall be in big-endian byte order.

The stack parameters for the **CRYPTO** token in this case are as follows: (*c-addr*$_1$ *c-addr*$_2$ *len 5 — c-addr*$_2$ *len*) where the value at *c-addr*$_2$ *len* is subtracted from the value at *c-addr*$_1$ *len*, and the result is stored at *c-addr*$_2$ *len.*

7.7.6 Incremental Secure Hash Algorithm (SHA-1)

This algorithm is standardised as FIPS 180-1. The incremental SHA- 1 calculation is divided in three steps: the initialisation step, the update step and the termination step. The update step may be repeated as often as necessary. The stack parameters for the **CRYPTO** token in this case are as follows:

(*c-addr*$_1$ *len*$_1$ *c-addr*$_2$ *num 6 — c-addr*$_2$)

where *c-addr*$_1$ *len*$_1$ is the input buffer for computation and *c-addr*$_2$ is the buffer containing the intermediate result, which is always 20 bytes in length. *Num* is 1 for the initialisation step, 2 for the update steps and 3 for the termination step.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

For the initialisation step *c-addr₁* and *len₁* are not taken into account. The initialisation step initialises *c-addr₂* with the five 32 bit offsets required by this algorithm.

For the update function the length of the input buffer has to be a multiple of 64 bytes. For the termination step the input buffer contains the remaining bytes. The length does not have to be a multiple of 64.

The 20-byte result buffer $c\text{-}addr_2$ shall not overlap the input buffer $c\text{-}addr_1$ len_1 and is passed between the different steps.

7.7.7 Cyclic Redundancy Check (CRC)

The stack parameters for the **CRYPTO** token in the case of the CRC algorithm are as follows: (*c-addr len num u1 u2 7 — u3*) *c-addr len* represents the block of input data. *Num* defines the order in which the bits of data bytes are shifted: zero is least significant bit first and non-zero is most significant bit first. *u1* is the input to the CRC algorithm, which allows chained computation over multiple blocks and also for preconditioning data for CRC-32. *u2* is the polynomial, where the bit representing the most significant term has been discarded (e.g.: the polynomial $x^{16} + x^{12} + x^5 + 1$ is represented by the binary value 1000000100001). *u3* is the 32 bit value of the new CRC computed over the input data.

7.7.8 DES Key Schedule

The stack parameters for the **CRYPTO** token in the case of the DES key schedule algorithm are as follows:

($c\text{-}addr_1$ $c\text{-}addr_2$ *8 —* $c\text{-}addr_2$)

where $c\text{-}addr_1$ is the 8 byte key buffer including parity bits and $c\text{-}addr_2$ the 128 byte key schedule output buffer.

7.7.9 DES encryption/decryption

The stack parameters for the **CRYPTO** token in the case of the DES encryption/decryption algorithm are as follows:

($c\text{-}addr_1$ $c\text{-}addr_2$ $c\text{-}addr_3$ *num 9 —* $c\text{-}addr_3$)

where $c\text{-}addr_1$ is the 8 byte input buffer, $c\text{-}addr_2$ the 128 byte key schedule buffer computed with **CRYPTO** function 8 and $c\text{-}addr_3$ the 8 byte result buffer. *Num* is 1 for encryption and 2 for decryption.

7.8 Vectored Execution Sockets

The plug and socket software mechanism in OTA is a convenient and flexible way to provide on-line configuration of the different modules that make up terminal programs and applications. Sockets hold execution pointers, also known as procedure pointers that allow the creation of a procedure whose behaviour may be changed at execution time. Sockets may be viewed (and implemented) as an array of procedures that are accessed through the **DOSOCKET** token, which takes the socket number as an in-line byte, or by the **IDOSOCKET** token, which takes the socket number from the stack.

Sockets enable re-configuration of a terminal program or application to provide variations or enhancements in the transaction processing flow. Sockets provide an interface between soft-ware modules and procedures that may be coming from several different sources (acquirer, issuer, etc.). Since an acquirer and an issuer have a contractual relationship, they may agree to use specific sockets provided by the acquirer's program in a terminal so that an issuer may extend the behaviour of the program, for example to provide a loyalty function (air miles, coupons, etc.).

A module may specify that sockets be reconfigured automatically when it is loaded for execution, or a client program may programmatically assign a new procedure to a socket at run-time. Provided security conditions permit it, sockets in an application may be assigned a default behaviour and then may be re-plugged with new procedures by subsequent modules, in order to provide specialised behaviours.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Part of the specification for using sockets is that all procedures vectored to use a particular socket shall have no stack effect (except for socket zero — see Section 8.21). This ensures program continuity no matter which vectored version of the procedure is executed. The default action of all sockets before modification shall be that of a no-op.

7.8.1 CSS Functions

An acquirer may allow transaction enhancements by code on an ICC (the CSS referred to above). If so, they are implemented with sockets. A library or application module may include the definition of new sockets for later **`PLUGSOCKET`** tokens coming from an ICC. In this case the module would define a socket and then use **`PLUGSOCKET`** to assign a default behaviour to it (often a null behaviour). If access control allows it, an ICC could later down-load tokens that define a new behaviour and then use **`PLUGSOCKET`** to store it in this same socket, overriding the default behaviour.

7.8.2 Socket Security

For security, the terminal software can specify a socket control procedure that controls whether or not each individual socket can be modified; thus, for example, the execution of code downloaded from an ICC can be strictly controlled by the acquirer.

This is achieved by specifying the socket control procedure to be applied on subsequent attempts to plug a socket (see **`PLUGSOCKET,`** Section 8.22 Socket Tokens). The procedure **`PLUGGABLE`** shall be written to return, for a given socket number, whether that socket may now be modified. When a module's socket list is subsequently processed at module load time, or when a socket is plugged programmatically, the virtual machine shall first execute the user-written **`PLUGGABLE`** procedure to determine whether the socket really can be plugged, and shall retain the existing behaviour of the socket if it cannot.

A module that wishes to restrict access to any sockets before another module is loaded for execution may execute a **`PLUGSOCKET`** on the **`PLUGGABLE`** socket (socket zero) with the chosen access predicate as a parameter, before loading that module. When the next module and any other modules loaded for its execution have their socket lists processed, sockets to which access is denied by the **`PLUGGABLE`** procedure shall retain their existing behaviour. This condition shall not be considered an error.

Code that wishes to restrict access to any sockets before further code is executed may execute a **`PLUGS OCKET`** with the chosen **`PLUGGABLE`** procedure as a parameter, at an appropriate point in program flow. A programmatic request to plug a socket (see **`PLUGSOCKET,`** Section 8.22 Socket Tokens) can determine whether the request was accepted or denied by the call to **`PLUGSOCKET.`**

Any socket whose behaviour was modified, either by the module loading process or dynamically by programmed command, shall be restored to the behaviour it had when the last executable module was loaded for execution, as part of the termination procedure packaged within the **`MODEXECUTE`** token. See Section 7.9 Module Handling Services for further details of module loading and execution.

7.8.3 Socket Organisation

An OTA terminal has 64 sockets, numbered 0 through 63. If a given socket is plugged with a procedure by more than one module, the latest operation simply replaces any earlier ones.

7.9 Module Handling Services

OTA software is managed by the virtual machine in the form of one or more *modules,* where each module may contain any of the following categories of information:

— Tokenised code

— Initialised data

— Uninitialised data allocation

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

— Database definitions

— TLV definitions

— Socket list

— Module interdependencies

The OTA Module Delivery Format (MDF) in which an OTA module is delivered to a terminal is International Standarded in Section 9. How this delivery is accomplished is beyond the scope of this International Standard.

The virtual machine maintains a non-volatile *repository* of modules that have been delivered and installed on the terminal. Each module in the repository shall be identified by a *module identifier* or *module ID.* The significance of the module ID is explained in Section 9.1 Module ID Format. Following registration in the module repository, module information is available through a non-volatile **`MODULES`** database maintained by the VM. The first fields in each record shall contain the first 20 bytes of the OTA Module Delivery Format. How the rest of the module information is stored within the VM is beyond the scope of this International Standard. The VM shall, however, provide the following features:

— f modules are not stored in MDF as described in Section 9, then they shall behave as though they were.

— The VM shall protect modules within the repository from modification by any other module.

— The VM shall make provision for a new version of a given module to be placed in the repository while a module of the same module ID exists for execution purposes.

There are two basic types of modules: *executable* modules, which have an entry point that is called directly by the VM when it is loaded, and *library* modules, which act as resources to other modules by providing exported procedures that may be executed individually by inter-module calls. A value of -1 in the **`MDF-ENTRY`** field of the module header defines the module as a library module.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

7.9.1 Module Loading by `MODEXECUTE`

There are conceptually two phases in processing a module: first it is "loaded," which means it is made accessible and its data, databases, etc. instantiated; then if it is an executable module the VM begins processing its tokens starting at its entry point. The execution procedure is described by the flowchart in Figure 4. Individual "subroutines" referenced in this flowchart are discussed in the following paragraphs.

Mark and Save Resources

Before execution of a module, the virtual machine shall mark its state and save any resources needed so that this state can be restored later. The state includes:

— The position of the extensible memory pointer, the frame pointer, and the frame end pointer.

— The contents of the entire current socket list.

— The TLVs registered in the TLV tag name list.

— Other internal data the VM implementation needs in order to manage the activation and execution of modules. *Load Module*

The module ID of the module to execute is passed to the *Load Module* subroutine, which is described in Section 7.9.2 Module Loading Procedure. If the module is loaded without error, it can be executed.

Executable?

If the **`MDF-ENTRY`** field is -1, the module is a library module and is not executable. Other-wise, the **`MDF-ENTRY`** field specifies the entry point of the module.

© ISO/IEC 2010 – All rights reserved

Execute Module

The VM starts the module by calling the token specified by the **`MDF-ENTRY`** field in the header. The module shall terminate using a **`RETURN`** token.

Release Resources

The resources needed for execution of the module are released. This requires:

— All volatile memory required to load the module, and any module that it required to be loaded, shall be released and cleared to zero. This includes, but is not limited to:

  — The space needed for all module's initialised and uninitialised data.

  — The space needed for any internal TLV buffers and management data structures de-fined by these modules.

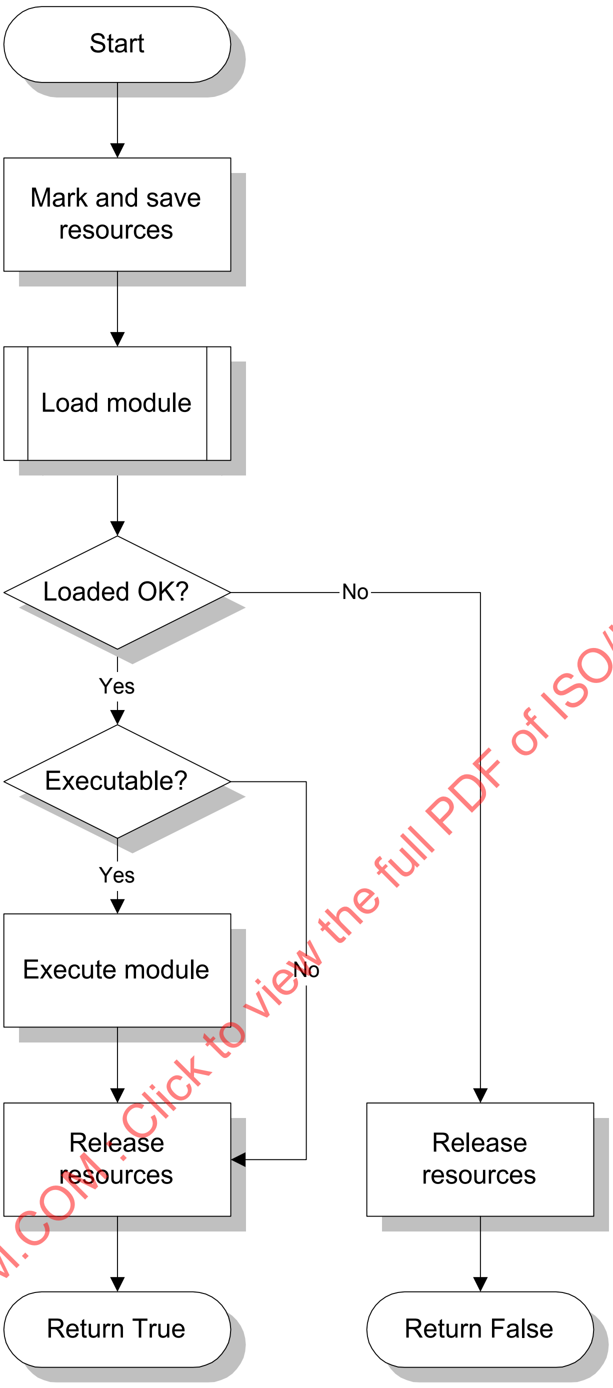
  — The space needed for any internal buffers and management data structures required by databases defined by these modules.

— The TLV name list maintained by the VM for tag lookup shall be restored to its state immediately before module execution.

— The contents of the socket list maintained by the VM shall be restored to its state immediately before module execution.

— The contents of the frame pointer, frame end pointer, and extensible memory pointer are restored to their values immediately before module execution.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010



IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Figure 7-2 — Module execution procedure

7.9.2 Module Loading Procedure

The process required to load a module, the “Load Module” subroutine, is shown in Figure 5. Individual “subroutines” appearing in this flowchart are discussed in the following paragraphs.

© ISO/IEC 2010 – All rights reserved

General Errors While Loading

If an error is detected during loading of a module, this causes the Load Module subroutine to immediately return *false.* A general error is one such as "out of memory" where there are insufficient resources to provide space for initialised data, uninitialised data, databases, or TLVs; when a duplicate TLV tag is discovered; and so on.

Order of Flowchart Items

The flowchart in Figure 5 shows the logical steps necessary to load a module; an implementation may wish to change the order to suit its needs as it sees fit. However, initialised data shall be set up before processing the database and TLV sections as these are part of the initialised data section.

Module Loaded?

If the module is already loaded into memory, it is not loaded a second time and Load Module immediately succeeds, returning *true.*

Module in Repository?

If the module is not in the repository, it cannot be loaded so Load Module subroutine fails, returning *false.*

Allocate Uninitialised Data Area and Zero

The field **MDF-ULEN** is used to reserve that many bytes of data for the module's uninitialised data area. This area then shall be set to all zeroes by the virtual machine.

Allocate and Prepare Module's Initialised Data

The field **MDF-ILEN** is used to reserve that many bytes of data for the module's initialised data area. Then, the initialised data are copied from the MDF file field **MDF- II** into this area. The relocation section, **MDF-RELOC** is used to apply any relocation required by the virtual machine at this time.

Process Module's List of TLVs

The TLVs defined in the module to be loaded are added by the virtual machine to its internal name list used for TLV lookup. The root of the TLV data structure is defined by the field **MDF-TLVROOT.** A duplicate tag that is entered twice into the TLV name list by different modules is considered as a general error. In this case the Load Module subroutine fails, returning *false.*

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

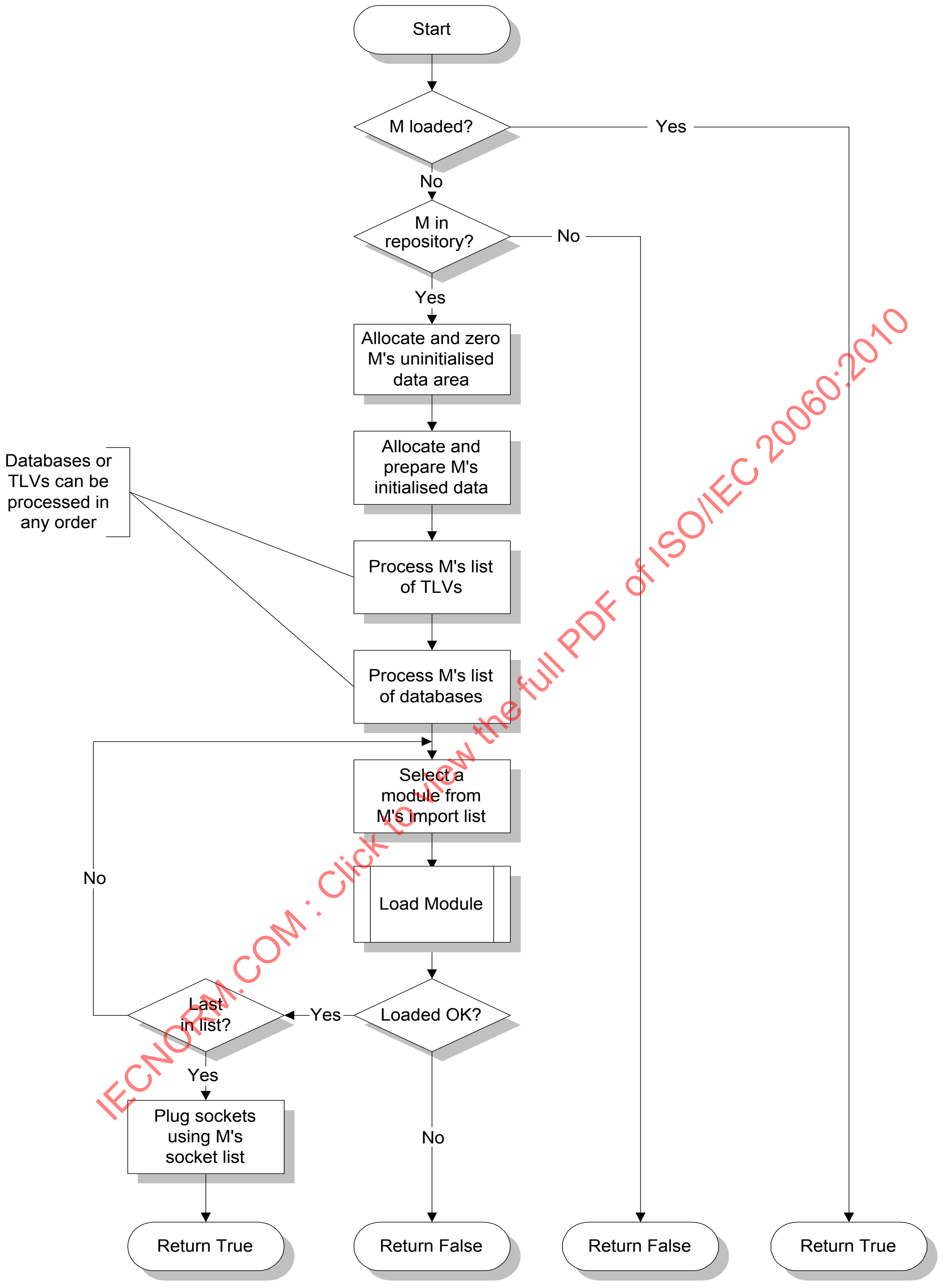

Start
M loaded?
Yes
No
M in repository?
No
Yes
Allocate and zero M's uninitialised data area
Allocate and prepare M's initialised data
Databases or TLVs can be processed in any order
Process M's list of TLVs
Process M's list of databases
Select a module from M's import list
No
Load Module
Last in list?
Yes
Loaded OK?
Yes
Plug sockets using M's socket list
No
Return True
Return False
Return False
Return True

Figure 7-3 — Module loading procedure

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Process Module's List of Databases

The databases defined in the module to be loaded are instantiated by the virtual machine. Instantiation of databases is covered in Section 7.3.2 Database Instantiation.

Select a Module from the Import List

The list of imported modules is traversed, recursively loading each one in turn.

Loaded OK?

If an imported module cannot be loaded for any reason, the module that imported the module is also deemed to have failed to load, as it cannot access the imported module's services. In this case, Load Module returns *false.*

Last Module?

After the last imported module has been recursively loaded, the module has had its resources allocated, loaded, and instantiated without error, so Load Module plugs the sockets in its list as described below and then returns *true* indicating that the module was loaded successfully.

Plug Sockets in Module's Socket List

The field **MDF-SLEN** is used to plug the sockets defined in the field **MDF- SOCK.** Any attempt to plug socket zero in the **MDF- SOCK** field shall be ignored by the virtual machine. If socket zero needs to be plugged, it may be accomplished using the **PLUGSOCKET** token.

7.9.3 Module Loading by MODCARDEXECUTE

Modules that are loaded from an ICC by **MODCARDEXECUTE** shall be handled differently than those loaded from the repository using **MODEXECUTE.** The flowchart for **MODCARDEXECUTE** is shown in Figure 6.

Mark and Save Resources

Before execution of a card module, the virtual machine shall mark its state and save any resources needed so that this state can be restored later. The state shall include:

— The position of the extensible memory pointer, the frame pointer, and the frame end pointer.

— The contents of the entire current socket list.

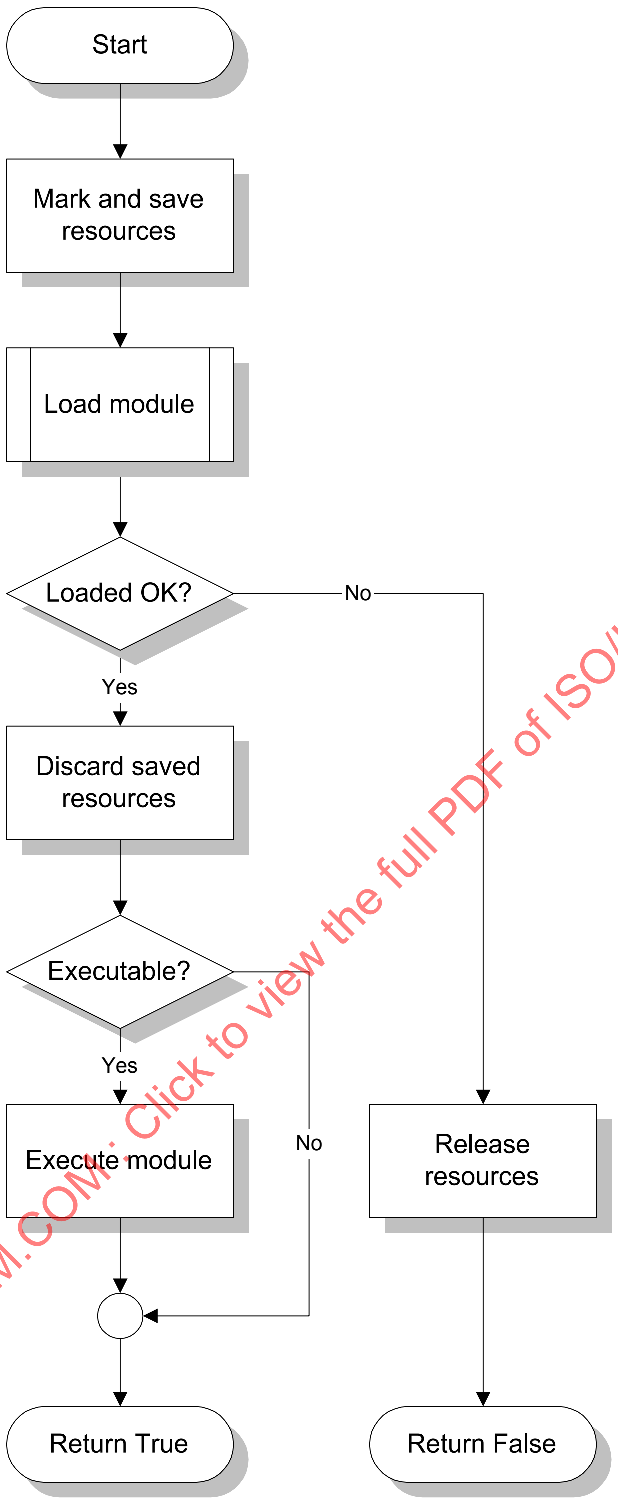
— The TLVs registered in the TLV tag name list.

— Other internal data the VM implementation needs in order to manage the activation of card modules.

Load Module

The module is loaded using Load Module; the difference is that the module is not in the repository and is not already loaded.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010



IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Figure 7-4 — ICC module execution procedure

Release Resources

If the card module is not loaded successfully, all resources shall be returned to the state they had immediately before execution of the **`MODCARDEXECUTE`** token. This requires:

© ISO/IEC 2010 – All rights reserved

- All volatile memory required to load the module, and any module that it required to be loaded, shall be released and cleared to zero. This shall include, but is not limited to:
  - The space needed for all module's initialised and uninitialised data.
  - The space needed for any internal TLV buffers and management data structures de-fined by these modules.
  - The space needed for any internal buffers and management data structures required by databases defined by these modules.
- The TLV name list maintained by the VM for tag lookup shall be restored to its state immediately before module execution.
- The contents of the socket list maintained by the VM shall be restored to its state immediately before module execution.
- The contents of the frame pointer, frame end pointer, and extensible memory pointer are restored to their values immediately before module execution.

Discard Resources

If the card module is loaded successfully, the state saved in the "Mark and save resources" step is simply discarded. Thus, a card module has been grafted onto a running system. To be useful, a card module shall plug sockets using the **`MDF- SOCK`** list; otherwise there is no way to execute any code that is present in the card module.

Full details of module support functions and formats are found in Section 8.28 Module Management Tokens and Section 9 Module Delivery Format.

8 Token Set Definition

8.1 Overview

OTA tokens are the instruction set of a virtual machine. The tokens may also be treated as an intermediate language of a compiler. Some implementations of the token loader/interpreter may actually translate OTA tokens to machine code.

OTA tokens are byte tokens, permitting a maximum of 256 tokens. One-byte prefix tokens allow the range of tokens to be extended to a theoretical maximum of 65536 tokens, regarding prefixes as defining pages of 256 tokens each. In fact, only three pages are defined.

Tokens without a prefix (single-byte tokens) are referred to as *primary* tokens, whereas those with prefixes (two-byte tokens) are referred to as *secondary* tokens.

The execution of any primary or secondary token that is not defined in the list below shall cause exception -511 (Illegal operation) to be thrown.

8.2 Conventions

8.2.1 Number Formats

Numbers larger than one byte are transmitted in token programs in big-endian two's complement form, with the most significant byte first. Within a token program, numbers should always be accessed by operators of the correct format, in order to allow programs to store numbers in the form most suited to the underlying architecture.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Double number types are held on the stack with the most significant cell topmost, and pro-grams may assume this stack order. In memory, double number types are held with the most significant cell at the lowest-addressed cell within the multi-cell type.

8.2.2 Token Descriptions

Tokens are split into several logical sets for the sake of convenience and are shown in separate sections below. All token values are in hexadecimal.

A typical token description is:

DROP **90**

(*x* —)

Remove the top item on the stack.

This describes a token named **DROP** whose "opcode" or "token value" is 90_H. The stack effect of the token is shown on the following line, and then a natural language description of the token's execution semantics is provided.

For tokens that require inline data, the following notation is used:

LIT **6E** *u:16*

(— u)

Push *u* onto the data stack.

The stack effect and natural language parts are much the same as before. However, after the token value $6E_H$, we have *"u:16"* which describes the format of the data following.

— The *u* part is the data type part and describes how the data are to be interpreted. In this case the data are interpreted as an unsigned integer.

— The *16* part describes the size of the data. In this case 16 bits are encoded in two bytes.

Other data types are taken from Table 1, and other sizes such as 8 and 32 are also used. Inline values are always held in big-endian format.

One additional data type is distinguished: *offset.* This defines the signed offset used in branch-type tokens and is described in the following section.

8.2.3 Branch and Code Offsets

Control structures are formed by a control token followed by a signed four-byte, two-byte, or single-byte offset. The offset size appears in the token descriptions as *offset :32, offset:16,* or *offset:8.* The offset following the control token is added to the token pointer register after the offset has been fetched. Thus, if a single-byte token is at *addr* and a two-byte offset follows it, the effective code address of the destination is *addr+1+2+ offset.*

8.2.4 Addresses

User-defined procedures are defined by their addresses within the OTA module. The token loader/interpreter shall adjust these to be actual addresses when the module is executed. If the tokens are translated or native-code compiled for larger processors, the token space address will not correspond to the actual address of the code.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

8.3 Data Typing

Most tokens operate on quantities with a data size and a signed or unsigned interpretation determined by the token, but certain instructions that access memory may take a **BYTE** data type prefix token. If a **BYTE** data type is specified, the data shall be zero-extended to cell width.

8.4 Token Compression

Three methods of token compression have been developed to reduce the size of the token image. They involve optimised data access tokens, procedure calling tokens, and quoting tokens. Any or all of these may be used by a compiler, token optimiser, or other means.

8.4.1 Optimised Data Access

In order to provide 10-bit addressing in data space without using more than one byte of in-line data, special tokens of the form **FETCH**xx and **STORE**xx have been provided. There are operators for memory fetch and store by cells and by bytes, in both initialised and uninitialised data space. These tokens are described individually in Section 8. Similarly, 10-bit address offsets in both initialised and uninitialised data space are provided by tokens of the form **SLIT**xx**,** described in Section 8.9 Address Generation Tokens.

For each of these tokens, the highest two bits of the data offset are encoded in the lower two bits of the token itself, and the lower eight bits of the offset follow as in-line data. For example, the token **FETCHU0** fetches cells from Uninitialised data space with offsets 0-1020, **FETCHU1** cells with offsets 1024-2044, and so on.

8.4.2 Special Procedure Calls

The group of tokens named **CALL0** … **CALL39** (tokens **0-27**H**)** provide a further opportunity to reduce module size by allowing a module to specify its 40 most frequently called procedures in such a way that they require only single-byte calls (rather than the other **CALL** forms, which all require some form of offset and hence occupy multiple bytes). The procedures selected for this purpose are specified in the Module Procedure List (see Section 9.6 Module Procedure List) in the module. The actual selection may be performed by the compiler, an optimiser, or other implementation-dependent means.

Since the actual association of these tokens with procedures is performed by the module loading process, programmers may not attempt to use them directly.

8.4.3 Quoting

Quoting is a technique to re-use common token sequences. Quoting resembles compression algorithms, and may also be seen as a general form of common sub-expression elimination.

Quoting tokens are used to mark common code sequences. Instead of replicating the code sequence again (or several times), it may be quoted using the **QUOTE** token. For example, consider the following two code sequences:

```
CALL AMOUNT
LTNMBR
NMBR NMBR NMBR
NMBR NMBRGT
LIT2 DEVWRITE
…
CALL AMOUNT
LTNMBR
NMBR      NMBR
NMBRGT
LIT0 DEVWRITE
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

These two sequences contain a common sub-sequence: **NMBR NMBR NMBRGT .** Rather than emit the sequence a second time, it may be replaced by a quote of the first sequence, as follows:

```
CALL AMOUNT
LTNMBR
NMBR
NMBR
L1: NMBR NMBR
NMBRGT
ENDQUOTE LIT2

DEVWRITE …
CALL AMOUNT
LTNMBR
QUOTE L1 LIT0
DEVWRITE
```

The **QUOTE** token has an offset that points to the start of the sequence to quote; the sequence extends to the **ENDQUOTE** token. When the virtual machine executes a **QUOTE,** it branches to the target token, in this case the token **NMBR** labelled **L1,** and remembers the address of the token immediately after **QUOTE** in a *quote return register.* The virtual machine continues executing tokens until it reaches the **ENDQUOTE** token, at which time it returns to the token immediately after **QUOTE,** stored in the quote return register. The quoting mechanism may be viewed as a subroutine call without using the return stack; but quotes cannot be nested as there is only a single quote return register.

If an **ENDQUOTE** token is encountered without being "called" by **QUOTE (i.e.,** the quote return register contains a value of zero), it functions as a **NOOP** and does not alter the semantics of the original code sequence.

8.5 Prefix Tokens

This group allows the limitations of 8-bit tokens to be overcome. Note that their stack action depends on the following token. The pair of tokens is referred to as a *secondary* token.

BYTE **E6**
(—)

Specify an unsigned byte operation. This may only prefix the frame tokens **TFRFETCHn, PFRFETCHn, TFRSTOREn, PFRSTOREn, SFRFETCH, FRFETCH, SFRSTORE,** and **FRSTORE,** and the direct access tokens **FETCHU*n*, FETCHD*n*, STOREU*n*,** and **STORED*n*.**

SECONDARY **FE**
(—)

Identify an expansion token to be treated as the first byte of a secondary token, such as **QTHROW** which is encoded as **FE F0.**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

8.6 Stack Manipulation Tokens

DROP 90

(x —)

Remove x from the stack.

DUP 91

(x — x x)

Duplicate x.

SWAP 92

(x_1 x_2 — x_2 x_1)

Exchange the top two stack items.

OVER 93

(x_1 x_2 — x_1 x_2 x_1)

Place a copy of x_1 on top of the stack.

NIP 94

(x_1 x_2 — x_2)

Remove the second item from the stack.

TUCK 95

(x_1 x_2 — x_2 x_1 x_2)

Save a copy of the top item on the stack under the second item.

ROT 96

(x_1 x_2 x_3 — x_2 x_3 x_1)

Bring the third item on the stack to the top.

MINUSROT 97

(x_1 x_2 x_3 — x_3 x_1 x_2)

Put the top item of the stack under the next two.

QDUP 98

(x — 0 | x x)

Duplicate x if x is non-zero.

RFROM 99

(– x) ; (R: x —)

Move x from the return stack to the data stack.

TOR 9A

(x —) ; (R: — x)

Move x from the data stack to the return stack.

RFETCH 9B

(– x) ; (R: x — x)

Copy x from the return stack to the data stack.

TWOS WAP 9C

($x1$ $x2$ $x3$ $x4$ — $x3$ $x4$ $x1$ $x2$)

Exchange two pairs of items on the stack.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

TWODROP **9D**

(*x1 x2 —)*

Discard two items from the stack.

TWODUP **9E**

(*x1 x2 — x1 x2 x1 x2)*

Duplicate a pair of items on the stack.

TWOTOR **9F**

(*x1 x2 —) ; (R: — x1 x2)*

Move x_1 x_2 from the data stack to the return stack.

TWORFROM **A0**

(– *x1 x2) ; (R: x1 x2 —)*

Move x_1 x_2 from the return stack to the data stack.

TWOOVER **A1**

(*x1 x2 x3 x4 — x1 x2 x3 x4 x1 x2)*

Copy x_1 x_2 to the top of the stack.

TWORFETCH **A2**

(– *x1 x2) ; (R: x1 x2 — x1 x2)*

Copy *x1 x2* from the return stack to the data stack.

PICK FE **43**

(x_u*..x1 x0 u —* x_u*..x1 x0 xu)*

Remove *u*. Copy x_u to the top of the stack. An ambiguous condition exists if there are fewer than *u*+2 items on the stack before **PICK** is executed.

TWOROT **FE 44**

(x_1 x_2 x_3 x_4 x_5 x_6 — x_3 x_4 x_5 x_6 x_1 x_2)

Move the third pair of items to the top of the stack.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

8.7 Data Access Tokens

FETCH **A3**

(a-addr — x)

Fetch a cell from *a-addr.*

STORE **A4**

(x a-addr —)

Store x in the cell at a-addr.

CFETCH **A5**

(c-addr — char)

Fetch the character stored at *c-addr,* zero-extending it to cell width.

CSTORE **A6**

(char c-addr —)

Store char at c-addr.

TWOFETCH **FE 30**

(a-addr — x1 x2)

Fetch two items stored at a-addr, where x2 is stored at a-addr, and x1 at the next consecutive cell.

TWOSTORE **FE 31**

(x1 x2 a-addr —)

Store two items on the stack into memory with x2 at a-addr and x1 at the next consecutive cell.

BCDFETCH **A7**

(c-addr len — u)

Fetch number u from a binary-coded-decimal sequence at c-addr for len bytes. The number is formatted with each digit representing 4-bit nibbles in the input string. Exception -506 (Digit too large) shall be thrown if any nibble is not a valid BCD digit. If the number in the input string has a value greater than 32 bits, the least-significant 32 bits shall be returned on the stack.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

BCDSTORE **A8**

(*u c-addr len* —)

Store number *u* as a binary-coded-decimal sequence into the memory at *c-addr* for *len* bytes. The number is formatted with each digit representing 4-bit nibbles in the output string. Leading nibbles shall be filled with zeroes if needed. The most significant part of the number shall be truncated if *len* is not long enough to hold all the digits.

BNFETCH **CD**

(*c-addr len* — *u*)

Fetch number *u* as a binary number from memory *c-addr* for *len* bytes. The value is stored in big-endian format. If there are more than four bytes of data at that location, the most significant bytes are lost.

BNSTORE **CE**

(*u c-addr len* —)

Store number *u* as a binary number into memory at *c-addr* for *len* bytes. The value is stored in big-endian format. Leading bytes shall be filled with zeroes if needed. The most significant part of the number shall be truncated if *len* is not long enough to hold all the bytes.

FETCHU0 … FETCHU3 **64** *u:8* … **67** *u:8*

(— *x*)

Fetch the cell from uninitialised data at offset *u**4 + *n**1024, where *n* is 0 through 3.

BYTE FETCHU0 … BYTE FETCHU3 **E6 64** *u:8* … **E6 67** *u:8*

(– *char*)

Fetch and zero-extend the byte from uninitialised data at offset *u* + *n**256, where *n* is 0 through 3.

STOREU0 … STOREU3 **68** *u:8* … **6B** *u:8*

(*x* —)

Store *x* to the cell in uninitialised data at offset *u**4 + *n** 1024, where *n* is 0 through 3.

BYTE STOREU0 … BYTE STOREU3 **E6 68** *u:8* … **E6 6B** *u:8*

(*char* —)

Store *char* to the byte in uninitialised data at offset *u* + *n**256, where *n* is 0 through 3.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

FETCHD0 … FETCHD3 74 *u:8* … 77 *u:8*

(— *x*)

Fetch the cell from initialised data at offset *u**4 + *n**1024, where *n* is 0 through 3.

BYTE FETCHD0 … BYTE FETCHD3 E6 74 *u:8* … E6 77 *u:8*

(– *char*)

Fetch and zero-extend the byte from initialised data at offset *u* + *n**256, where *n* is 0 through 3.

STORED0 … STORED3 78 *u:8* … 7B *u:8*

(*x* —)

Store *x* to the cell in initialised data at offset *u**4 + *n**1024, where *n* is 0 through 3.

BYTE STORED0 … BYTE STORED3 E6 78 *u:8* … E6 7B *u:8*

(*char* —)

Store *char* to the byte in initialised data at offset *u* + *n**256, where *n* is 0 through 3.

8.8 Literal Tokens

LIT0 … LIT15 30 … 3F

(-- *num*)

Push one of the values zero through 15 onto the stack.

LITMINUS1 7E

(-- *num*)

Push the value -1 onto the stack.

SLIT 6D *u:8*

(— *u*)

Push *u* onto the stack.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

NLIT **7F** *u:8*

(— num)

Push the negative value of *u* onto the stack as *num*.

LIT **6E** *u:16*

(— u)

Push *u* onto the stack.

ELIT **6F** *num:32*

(— num)

Push *num* onto the stack.

8.9 Address Generation Tokens

LITC **7D** *offset:16*

(— xp)

Push the execution pointer *xp* using the branch *offset.*

ELITC **FE F6** *offset:32*

(— xp)

Push the execution pointer *xp* using the branch *offset.*

LI TD **7C** *u:16*

(— c-addr)

Push the address of offset *u* in this module's initialised data section.

ELITD **FE F7** *num:32*

(— *c-addr)*

Push the address of offset *num* in this module's initialised data section.

LITU 6C *u:16*

(— c-addr)

Push the address of offset *u* in this module's uninitialised data section.

ELITU **FE F8** *num:32*

(— c-addr)

Push the address of offset *num* in this module's uninitialised data section.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

SLITU0 … SLITU3 **60** *u:8* … **63** *u:8*

(— *a-addr*)

Push the address of offset *u**4 + *n**1024 in this module's uninitialised data section, where *n* is 0 through 3.

SLITD0 … SLITD3 **70** *u:8* … **73** *u:8*

(— *a-addr*)

Push the address of offset *u**4 + *n**1024 in this module's initialised data section, where *n* is 0 through 3.

USERVAR **FD**

(*u* — *a-addr*)

Push the address of user variable *u,* where *u* is in the range 0 through 15. See Section 3.2.7 for an explanation of user variables.

8.10 Arithmetic Tokens

ADD **A9**

($num_1|u_1$ $num_2|u_2$ — $num_3|u_3$)

Add $num_1|u_1$ to $num_2|u_2$ to give $num_3|u_3$.

SADDLIT **BE** num_2**:8**

(num_1 — num_3)

Add num_1 to num_2 to give num_3.

ADDLIT1 **DD**

($num_1|u_1$ — $num_2|u_2$)

Add one to $num_1|u_1$ to give $num_2|u_2$.

SUB **AA**

($num_1|u_1$ $num_2|u_2$ — $num_3|u_3$)

Subtract $num_2|u_2$ from $num_1|u_1$ to give $num_3|u_3$.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

SUBLIT1 **DE**

($num_1|u_1$ — $num_2|u_2$)

Subtract one from $num_1|u_1$ to give $num_2|u_2$.

MUL **AB**

($num_1|u_1$ $num_2|u_2$ — $num_3|u_3$)

Multiply $num_1|u_1$ by $num_2|u_2$ to give $num_3|u_3$.

SMULLIT **BF** ***u:8***

($num_1|u_1$ – $num_2|u_2$)

Multiply $num_1|u_1$ by *u* to give $num_2|u_2$.

DIV **FE 51**

(num_1 num_2 — num_3)

Divide num_1 by num_2 to give num_3.

DIVU **FE 52**

(u_1 u_2 — u_3)

Divide u_1 by u_2 to give u_3.

MOD **AC**

(num_1 num_2 — num_3)

Divide num_1 by num_2 to give remainder num_3.

MODU **FE 53**

(u_1 u_2 — u_3)

Divide u_1 by u_2 to give remainder u_3.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

AND **AD**

(x_1 x_2 — x_3)

Perform the bitwise "and" of x_1 and x_2 to give x_3.

OR **AE**

(x_1 x_2 — x_3)

Perform the bitwise inclusive-or of x_1 and x_2 to give x_3.

XOR **FE 50**

(x_1 x_2 — x_3)

Perform the bitwise exclusive-or of x_1 and x_2 to give x_3.

NEGATE **F1**

(num_1 — num_2)

Negate num_1 giving num_2.

DADD **FE 20**

(*d1 d2|ud1 ud2 — d3|ud3*)

Add d_1 |ud_1 to d_2|ud_2 giving d_3|ud_3.

DNEGATE **FE 22**

(d_1 — d_2)

Negate the 64-bit number d_1 giving its additive inverse d_2.

MMUL **FE 55**

(num_1 num_2 — *d*)

Multiply num_1 by num_2 to give *d*. This is a multiply of two single numbers to produce a double product.

MMULU **FE 56**

(*u1 u2 — ud*)

Multiply u_1 by u_2 to give *ud*. This is an unsigned multiply of two single numbers to produce a double product.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

MSLMOD **AF**

(*d* num_1 — num_2 num_3)

Divide a double number by a single number and return both the quotient and the remainder. $num_2 = mod(d, num_1)$, $num_3 = d/num_1$.

MSLMODU **FE 54**

(*ud* u_1 — u_2 u_3)

Divide a double unsigned number by a single unsigned number and return both the quotient and the remainder. $u_2 = mod(ud, u_1)$, $u3 = ud/u_1$.

ABS **FE 12**

(*num* — *u*)

Return the absolute value of *num*.

MIN **FE 10**

(num_1 num_2 — $num_1|num_2$)

Return the smaller of num_1 and num_2.

MAX **FE 11**

(num_1 num_2 — $num_1|num_2$)

Return the larger of num_1 and num_2.

SHL **B1**

(x_1 — x_2)

Shift x_1 left one bit, filling the right-most bit with 0.

SHLN **B2**

(x_1 *u* — x_2)

Shift x_1 left by mod(*u*,32) places, filling the right-most bits with zeros, to give x_2.

SHRNU **B0**

(x_1 *u* — x_2)

Shift x_1 right by mod(*u*,32) places, inserting zero bits, to give x_2.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

SHRN **FE 57**

(*x_1 u — x_2*)

Shift *x_1* arithmetically right by mod(*u*,32) places, propagating the most significant bit.

WIDEN **E7**

(*char — num*)

Sign extend *char* from an 8-bit value to a 32-bit value by propagating bit 7 of *char* into bits 8 through 31.

INCR **CC**

(*num|u a-addr —*)

Add *num|u* to the cell at *a-addr.*

8.11 Relational Tokens

SETNE **BC**

(*x — flag*)

Return true if *x*><0.

SETEQ **BA**

(*x — flag*)

Return true if *x* = 0.

SETGE **FE 15**

(*num — flag*)

Return true if *num >= 0*

SETLT **BB**

(*num — flag*)

Return true if .*num* < 0

SETGT **FE 18**

(*num — flag*)

Return true if .*num* >0

SETLE **FE 19**

(*num — flag*)

Return true if .*num =< 0*

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

CMPNE **B4**

(x_1 x_2 — *flag*)

Return true if $x_1 >< x_2$

CMPEQ **B3**

(x_1 x_2 — *flag*)

Return true if $x_1 = x_2$.

CMPGEU **B8**

(u_1 u_2 — *flag*)

Return true if . $u_1 >= u_2$

CMPLTU **B6**

(u_1 u_2 — *flag*)

Return true if . $u_1 < u_2$

CMPGTU **B9**

(u_1 u_2 — *flag*)

Return true if .$u1 > u2$

CMPLEU **FE 14**

(u_1 u_2 — *flag*)

Return true if $u1 =< u2$

CMPGE **FE 17**

(*num1 num2* — *flag*)

Return true if $num_1 >= num_2$

CMPLT **B5**

(*num1 num2* — *flag*)

Return true if. $num_1 < num_2$

CMPGT **B7**

(*num1 num2* — *flag*)

Return true if. $num_1 > num_2$

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

CMPLE **FE 13**

(num1 num2 — flag)

Return true if. num_1 =< num_2

DCMPLT **FE 21**

(*d1 d2 — flag)*

Return true if.d1 < d2

WITHIN **BD**

(*num1 num2 num3|u1 u2 u3 — flag)*

Compare a test value *num1|u1* with a lower limit *num2|u2* and an upper limit *num3|u3,* returning *true* if either *($num_2|u_2 < num_3|u_3$* and *($num_2|u_{2=<}$ $num_1|u_1$* and *$num_1|u_1 \leq num_3|u_3$))* or *(num2|u2 > num3|u3* and *(num2||u2 =< num1|u1* or *num1|u1 < num3|u3))* is true, returning *false* otherwise. An ambiguous value of *flag* exists if *num1|u1, num2|u2,* and *num3|u3* are not all the same type.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

8.12 String Tokens

MOVE **C5**

(addr1 addr2 len —)

If *len* is greater than zero, copy *len* bytes from *addr1* to *addr2,* non-destructively if the areas overlap. After the transfer the *len* bytes at *$addr_2$* are the same as the *len* bytes at *addr1* before the transfer.

FILL **C6**

(c-addr len char —)

If *len* is greater than zero, fill the string *c-addr len* byte-by-byte with *char.*

COMPAR **C7**

(*$c\text{-}addr_1$ len_1 $c\text{-}addr_2$ len_2 — num)*

Compare two strings in memory *$c\text{-}addr_1$* and *$c\text{-}addr_2$* byte-by-byte, up to the shorter of *len1* and *len2* or until a difference is found. The return value *num* may be interpreted as shown in Table 12.

Table 12 — Result codes from COMPARE

Strings equal and $len_1 = len_2$	0
Strings identical up to shorter of len_1 and len_2, and $len_1 < len_2$	-1
Strings identical up to shorter of len_1 and len_2, and $len_1 > len_2$	1
Strings not identical up to shorter of len_1 and len_2, and first non-matching byte in string at $addr_1$ < equivalent byte in $addr_2$	-1
Strings not identical up to shorter of len_1 and len_2, and first non-matching byte in string at $addr_1$ > equivalent byte in $addr_2$	1

COUNT **CB**

(*c-addr*$_1$ — *c-addr*$_2$ *len*)

Return the character string specification for the counted string stored at *c-addr*$_1$. *c-addr*$_2$ is the address of the first character after *c-addr*$_1$. *len* is the contents of the character at *c-addr*$_1$, which is the length in characters of the string at *c-addr*$_2$.

SCAN **FE 40**

(*c-addr*$_1$ *len*$_1$ *char* — *c-addr*$_2$ *len*$_2$)

Parse the character string at *c-addr*$_1$ for *len*$_1$ bytes for bytes containing *char*. *c-addr*$_2$ is the address of the byte where *char* is found or the end of the string. *len*$_2$ is the length in characters of the remaining string at *c-addr*$_2$ which will be zero if *char* was not found.

SKI P **FE 41**

(*c-addr*$_1$ *len*$_1$ *char* — *c-addr*$_2$ *len*$_2$)

Parse the character string at *c-addr*$_1$ for *len*$_1$ bytes skipping bytes that contain *char*. *c-addr*$_2$ is the address of the first byte that differs from *char*, or the end of the string. *len*$_2$ is the length in characters of the remaining string at *c-addr*$_2$ which will be zero if the string was completely filled with *char*.

PLUSSTRING **CA**

(*c-addr*$_1$ *len*$_1$ *c-addr*$_2$ *len*$_2$ — *c-addr*$_2$ *len*$_3$)

Store the string at *c-addr*$_1$ for *len*$_1$ bytes at the end of the string at *c-addr*$_2$ for *len*$_2$ bytes. Return the beginning of the destination string (*c-addr*$_2$) and the sum of the two lengths (*len*$_3$). It is the programmer's responsibility to assure that there is room at the end of the destination string to hold both strings.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

MINUSTRAILING **C8**

(*c-addr* len_1 — *c-addr* len_2)

Remove trailing spaces from a string. If len_1 is greater than zero, len_2 is equal to len_1 less the number of spaces (ASCII 20_H) at the end of the character string specified by *c-addr* len_1*.* If len_1 is zero or the entire string consists of spaces, len_2 is zero.

MINUSZEROS **FE 35**

(*c-addr* len_1 — *c-addr* len_2)

Remove trailing nulls from a string. If len_1 is greater than zero, len_2 is equal to len_1 less the number of binary zeros (nulls) at the end of the character string specified by *c-addr* len_1. If len_1 is zero or the entire string consists of nulls, len_2 is zero.

SLASHSTRING **FE 38**

($c\text{-}addr_1$ len_1 *num* — $c\text{-}addr_2$ len_2)

Adjust the character string at $c\text{-}addr_1$ by *num* characters. The resulting character string, specified by $c\text{-}addr_2$ len_2, begins at $c\text{-}addr_1$ plus *num* characters and is len_1 minus *num* characters long.

CNFETCH **FE 45**

($c\text{-}addr_1$ len_1 — $c\text{-}addr_2$ len_2)

Fetch an ASCII string to the temporary location $c\text{-}addr_2$ for len_2 bytes that represents the compressed number in the string at $c\text{-}addr_1$ for len_1 bytes. The number is formatted with each character of the output string representing a 4-bit nibble in the input string according to the current number conversion BASE. For BASE set to 10, the output string shall be terminated when a nibble with all bits set or the end of the string is encountered. For every other BASE, the output string shall be terminated when the end of the string is en-countered. Exception -506 (Digit too large) shall be thrown if a nibble in the input string is not a number in the current BASE. The output string shall be moved to a more permanent location immediately.

CNSTORE **FE 46**

($c\text{-}addr_1$ len_1 $c\text{-}addr_2$ len_2 —)

Store the number represented by the ASCII string at $c\text{-}addr_1$ for len_1 bytes as a compressed number into the string at $c\text{-}addr_2$ for len_2 bytes. The number is formatted with each character representing a 4-bit nibble in the output string according to the current number conversion BASE. Trailing nibbles shall be filled with F's if needed. The number shall be truncated if len_2 is not long enough to hold all the characters $(len_2<[len_1+1\ ]/2)$. Exception -506 (Digit too large) shall be thrown if a character in the input string is not a number in the current BASE.

STRLIT **F2** *counted-string*

(– *c-addr len*)

Return *c-addr len* which contains the string of characters following this token. In the token stream the string is stored as a count byte followed by that many characters. The returned string may be contained

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

in a temporary buffer. The maximum length of this temporary buffer is implementation dependent but shall be no less than 255 characters. The output string shall be moved to a more permanent location immediately. A program shall not alter the returned string.

8.13 Frame Tokens

The following tokens provide access to the frame store.

SMAKE FRAME **E8** *$u1$:8 $u2$:8*

($x_{u1} \ldots x_1$ —)

Create frame. u_1 is the number of cells containing the procedure parameters, and u_2 is the number of cells of temporary variables. **SMAKEFRAME** allocates u_1+u_2+2 cells, and then sets the frame pointer to point to the new frame. This token allows procedure parameters and temporary variables to be accessed by **FRFETCH** and **FRSTORE.** See Section 3.2.5 for a discussion of the frame mechanism.

The virtual machine is permitted to build frames on the return stack, so use of frames is constrained by the rules that apply to return stack usage in general (see Section 6.2.4). Procedure parameters are moved from the data stack into the frame by **SMAKE FRAME** so that they can be accessed by **FRFETCH** and **FRS TORE.**

Exception -3066 (Frame stack error) shall be thrown if it is not possible to build a frame of the requested size.

MAKE FRAME **FE 64** *$u1$: 16 $u2$: 16*

($x_{u1} \ldots x_1$ —)

Create frame. See **SMAKEFRAME** for further details; u_1 and u_2 are 16-bit values rather than 8-bit values.

RELFRAME **E9**
(–)

Restore the frame pointer to its previous value and release the current frame. See Section 6.2.5 for a discussion of the frame mechanism.

PFRFETCH2 … PFRFETCH5 **40 … 43**

(-- *x*)

Fetch the cell at offset n*4 in the active frame, where n is 2 through 5.

TFRFETCH12 … TFRFETCH1 **44 … 4F**

(— x)

Fetch the cell at offset -n*4 in the active frame, where n is 1 through 12.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

BYTE TFRFETCH12 … BYTE TFRFETCH1 **E6 44 … E6 4F**

(-- *char*)

Fetch and zero-extend the byte at offset -*n* in the active frame, where *n* is 1 through 12.

PFRSTORE2 … PFRSTORE5 **50 … 53**

(*x* —)

Store *x* to the cell at offset *n**4 in the active frame, where *n* is 2 through 5.

TFRSTORE12 … TFRSTORE1 **54 … 5F**

(*x* —)

Store *x* to the cell at offset -*n**4 in the active frame, where *n* is 1 through 12.

BYTE TFRSTORE12 … BYTE TFRSTORE1 **E6 54 … E6 5F**

(*char* —)

Store *char* to the byte at offset -*n* in the active frame, where *n* is 12 through 1.

SFRFETCH **E1** `num:8`

(— *x*)

Fetch the cell at the offset num*4 in the active frame.

BYTE SFRFETCH **E6 E1** `num:8`

(— *char*)

Fetch and zero-extend the byte at the offset num in the active frame.

SFRSTORE **E2** `num:8`

(*x* —)

Store x at the offset num*4 in the active frame.

BYTE SFRSTORE **E6 E2** `num:8`

(*char* —)

Store char into the byte at the offset num in the active frame.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

FRFETCH **E4** *num:16*

(— *x*)

Fetch the cell at the offset num*4 in the active frame.

BYTE FRFETCH **E6 E4** *num:16*

(— *char*)

Fetch and zero-extend the byte at the offset *num* in the active frame.

FRS TORE **E5** num:16

(*x* —)

Store x at the offset num*4 in the active frame.

BYTE FRS TORE **E6 E5** num:16

(*char* —)

Store char at the offset num in the active frame.

SFRADDR **E0** num:8

(— *a-addr*)

Return the address of the offset num*4 in the active frame.

FRADDR **E3** num:16

(— *c-addr*)

Return the address of the offset num in the active frame.

8.14 Extensible Memory Tokens

The following tokens provide access to an extensible "rubber band" region of linear memory in the data space provided and managed by the virtual machine. The tokens in this section and the exception handling tokens **THROW** and **QTHROW** are the only exceptions to the requirement that a token shall have no net effect on the extensible memory pointer.

EXTEND **FE 36**

(*len* –– *a-addr*)

Increase extensible memory by *len* cells, returning the cell-aligned address *a-addr* of the first cell in the new allocation. If *len* is zero, return a pointer to the next unallocated cell. A new allocation is contiguous with an immediately previous use of **EXTEND** or **CEXTEND** (if any) in the same module, increasing the size of a previous **CEXTEND** if necessary to maintain cell alignment. Exception -3071 (Out of memory) shall be thrown if there is in-sufficient memory available.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

CEXTEND **EA**

(*len* — *c-addr*)

Increase extensible memory by *len* bytes, returning the address *c-addr* of the first byte in the new allocation. If *len* is zero, return a pointer to the next unallocated byte. A new allocation is contiguous with an immediately previous use of **EXTEND** or **CEXTEND** (if any) in the same module. Exception -3071 (Out of memory) shall be thrown if there is insufficient memory available.

RELEASE **EB**

(*addr* —)

Release storage acquired through **EXTEND** or **CEXTEND,** setting the "free pointer" to *addr*. If *addr* is invalid (before the start of extensible memory, or beyond the current "free pointer") an exception -9 (Invalid memory address) may be thrown. The contents of memory that has been released are indeterminate.

8.15 Flow of Control Tokens

8.15.1 Branch Tokens

I JMP **F0**

(*xp* —)

Branch unconditionally to the function whose *xp* is given.

SBRA **84** *offset:8*

(—)

Branch unconditionally using the supplied branch *offset*.

BRA **85** *offset:16*

(—)

Branch unconditionally using the supplied branch *offset*.

EBRA **FE 60** *offset:32*

(—)

Branch unconditionally using the supplied branch *offset*.

SBZ **80** *offset:8*

(*x* —)

Take branch if *x* = 0 using the supplied branch *offset*.

BZ **81** *offset:16*

(*x* —)

Take branch if *x* = 0 using the supplied branch *offset*.

EBZ **FE 63** *offset:32*

(*x* —)

Take branch if *x* = 0 using the supplied branch *offset*.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

SBNZ **82** *offset:8*

(x —)

Take branch if *x* >< 0 using the supplied branch *offset.*

BNZ **83** *offset:16*

(num —)

Take branch if x >< 0 using the supplied branch offset

EBNZ **FE 62** *offset:32*

(x —)

Take branch if *x* >< 0 using the supplied branch *offset*

ROF **FE 67** *offset:16 (num$_1$ num$_2$ — | num$_1$)*

If *num$_1$* = *num$_2$,* skip the inline *offset,* drop both *num$_1$* and *num$_2$,* and continue execution after the **ROF** token and *offset.* If *num$_1$* >< *num$_2$,* drop *num$_2$,* and take branch using *offset.*

SROFLIT **86** *u:8 offset:8*

(*num — | num)*

If *num* = *u,* skip the inline *offset,* drop *num,* and continue execution after the **SROFLIT** token and *offset.* If, If *num* >< *u* take branch using *offset.*

ROFLIT **87** *u:16 offset:8 (num — | num)*

If *num* = *u,* skip the inline *offset,* drop *num,* and continue execution after the **ROFLIT** token and *offset.* If *num* >< *u*, take branch using *offset.*

8.15.2 Call Tokens

CALL0 … CALL39 **00 … 27**

(—) ; (R: — nest-sys)

Call procedure n specified in the procedure table for this module.

S CALL **28 offset:8**

(—) ; (R: — nest-sys)

Call a procedure using the given offset from the token pointer.

CALL **29 offset:16**

(—) ; (R: — nest-sys)

Call a procedure using the given offset from the token pointer.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

ECALL **FE 61 offset:32**

(—) ; (R: — nest-sys)

Call a procedure using the given ofset from the token pointer.

IMCALL **2A u1:8 u2:8**

(--) ; (R: — nest-sys)

Execute a procedure in another module, where u1 is the index into the currently executing module's Import List and u2 is the index into the called module's Export List. Stack effects are dependent on the procedure called. See Section 7.9 for an explanation of modules.

ICALL **2B**

(xp —) ; (R: — nest-sys)

Call the procedure whose execution pointer is xp.

RETURN **2C**

(—) ; (R: nest-sys —)

Return from the procedure.

8.15.3 Loop Tokens

The tokens in this section manage or use loop control parameters. The stack comments for these tokens reflect the usual, but not mandatory, practice of saving these parameters on the return stack (see Section 6.2.4).

RDO **88 offset:16**

(num1 num2| u1 u2 —) ; (R: — loop -sys)

Set up loop control parameters representing a loop index whose initial value is num2|u2, an upper limit of num1|u1 and a termination location specified by the in-line offset. An ambiguous condition exists if num1|u1 and num2|u2 are not both the same type. Exception -7 (Do loops nested too deeply) may be thrown if the loop control parameters cannot be accommodated by the virtual machine. The "continuation location" for RLOOP or RPLUS LOOP is the token immediately following the offset. The virtual machine may store the parameters from the data stack on the return stack (see Section 6.2.4).

RQDO 89 **offset:16**

(num1 num2| u1 u2 —) ; (R: — loop -sys)

If num1|u1 = num2|u2, discard both values and branch to the termination location specified by the inline offset. Otherwise, set up loop control parameters representing a loop index whose initial value is num2|u2, an upper limit of num1|u1 and a termination location at the given offset. An ambiguous condition exists if num1|u1 and num2|u2 are not both the same type. Exception -7 (Do loops nested too deeply) may be thrown if the loop control parameters cannot be accommodated by the virtual machine. The "continuation location" for RLOOP or RPLUSLOOP is the token immediately following the offset. The virtual machine may store the parameters from the data stack on the return stack (see Section 6.2.4).

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

RI **8A**

(— num1|u1) ; (R: loop-sys — loop-sys)

Return the loop index from loop-sys. Since loop control information may be held on the return stack, the use of any other tokens that may place data on the return stack within a loop renders the loop index inaccessible.

RJ **FE 37**

(— num1|u1) ; (R: loop-sys$_1$ loop-sys$_2$ — loop-sys$_1$ loop-sys$_2$)

Return the loop index from the outer do-loop structure loop-sys$_1$. Since loop control in-formation may be held on the return stack, the use of any other tokens that may place data on the return stack within a loop renders the loop index inaccessible.

RLEAVE **8B**

(—) ; (R: loop-sys —)

Discard the loop control parameters loop-sys, and branch to the termination location specified in loop -sys.

RLOOP **8C**

(—) ; (R: loop-sys$_1$ — |loop -sys$_2$)

Add one to the loop index represented in loop-sys1 giving loop -sys$_2$. If the loop index equals the upper limit, exit the loop and discard the loop parameters. Otherwise branch to the continuation location (see description of RDO and RQDO).

RPLUSLOOP **8D**

(*num —) ; (R: loop-sys$_1$ — |loop-sys$_2$)*

Add *num* to the loop index represented in *loop-sys*$_1$. If the loop index crossed the boundary between the loop limit minus one and the loop limit, exit the loop and discard the loop control parameters. Otherwise branch to the continuation location (see description of **RDO** and **RQDO).**

8.15.4 Hybrid Tokens

These tokens are designed to support the implementation of data types and structures in high-level languages.

DOCREATE **FE F1 u:16**

(— a-addr) ; (R: nest-sys —)

Return the address in data space whose offset is u and perform a RETURN.

EDOCREATE **FE F5 num:32**

(— a-addr) ; (R: nest-sys —)

Return the address in data space whose offset is num and perform a RETURN.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

```
DOCLAS S                                          DF u:16 offset:16
    ( — a-addr )
```

Push onto the data stack the address resulting from adding u to the base address of initialised data space, then branch using offset.

```
EDOCLAS S                                    FE F4 num:32 offset:32
    ( — a-addr )
```

Push onto the data stack the address resulting from adding num to the base address of initialised data space, then branch using offset.

8.15.5 Quoting Tokens

See Section 8.4.3 for a discussion of token sequence re-use or “quoting”.

```
QUOTE                                                 8E offset:16
    ( — )
```

Set the quote return register to point to the next token to be executed after the QUOTE token and branch to the address specified by offset.

```
ENDQUOTE                                                        8F
    ( — )
```

If the quote return register has been set using QUOTE, set the token pointer to the contents of the quote return register and reset the quote return register such that it contains no value. If an ENDQUOTE token is encountered without being “called” by QUOTE (i.e., the quote return register contains no value), it functions as a NOOP.

8.16 Exception Tokens

```
CATCH                                                           2D
    ( i*x xp — j*x num )
```

Push an exception frame on the exception stack and call the procedure at xp, in such a way that control can be transferred to a point just after this token if a THROW is executed during the execution of the procedure. If execution of xp completes normally (i.e., the exception frame pushed by this CATCH is not popped by an execution of THROW) pop the exception frame and return zero on top of the data stack, above whatever items have been returned by execution of xp. See THROW for definition of the return state otherwise.

```
THROW                                                           2E
    ( k*x num — k*x | i*x num )
```

If num is non-zero, pop the topmost exception frame from the exception stack, along with everything on the return stack above that frame. Then adjust the depths of all stacks so that they are the same as the depths saved in the exception frame (i.e., i is the same number as i in the input arguments to the corresponding CATCH), put num on top of the stack, and return control to a point just after the last executed CATCH. Restore the frame pointers to the values they had before the corresponding CATCH. In addition, if the

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

word executed by CATCH causes the execution of MODEXECUTE or MODCARDEXECUTE tokens, the allocation of extensible memory will be restored to its condition at the time of the execution of those tokens.

QTHROW **FE F0**

(k*x num1 num2 – k*x | i*x num2)

If num1 is non-zero, THROW with value num2. If num1 is zero, discard num1 and num2.

8.17 Date, Time, and Timing Tokens

GETTIME **EC**

(– u1 u2 u3 u4 u5 u6)

Return the current time and date. u1 is the second {0...59}, u2 is the minute {0...59}, u3 is the hour {0.. .23 }, u4 is the day { 1.. .31 }, u5 is the month { 1... 12 }, and u6 is the year {0...9999}.

SETTIME **FE 79**

(u1 u2 u3 u4 u5 u6 –)

Set the current time and date. u1 is the second {0...59}, u2 is the minute {0...59}, u3 is the hour {0.. .23 }, u4 is the day { 1... 31 }, u5 is the month { 1... 12 }, and u6 is the year {0.. .9999}. Some terminals may not be able to support this function. On these terminals, Exception -21 (Unsupported operation) shall be thrown.

GETMS **ED**

(– u)

Return current system free-running milliseconds timer value. The timer value wraps from 232-1 to zero.

MS **EE**

(u –)

Wait for at least u milliseconds but not more than u plus twice the timer resolution. In systems that have a multitasking operating system, u = 0 may cause a scheduling operation, whereas in other systems an immediate return shall be performed.

8.18 Generic Device I/O Tokens

Each instance of a device type shall be assigned a unique device number. A table of standard device number assignments is given in Annex C. Status ior codes returned by device control functions are device dependent, except that an ior code of zero always indicates success. Tables of standard ior codes for each device type are given in Annex C.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

DEVOPEN **FE 93**

(dev — ior)

Open device dev. On buffered devices DEVOPEN clears all data from the buffers. Ior -32759 (Device shall be opened) shall be returned by all DEVtokens applied on a device that is not open. All devices are closed by start-up of the terminal. Applying the DEVOPEN token on a device that is already open, results in a no-op and an ior -32758 (Device already open).

DEVCLOSE **FE 9E**

(dev — ior)

Close the given device dev. After closing the device, the device is not accessible any-more and all DEVtokens applied on a closed device return ior -32759 (Device shall be opened). Note that closing a modem device with an open connection implicitly performs a hangup of the modem.

DEVEKEY **FE 90**

(dev — echar)

Read an extended character from input device dev. This token does not take into account the time-out set by the DEVIOCTL token. It waits for ever until an extended character is received.

Because this device token does not return an ior, the ior codes listed in Table 22 shall be thrown as exceptions.

DEVEKEYQ **FE 91**

(dev — flag)

Return true if a character is ready to be read from input device dev.

Because this device token does not return an ior, the ior codes listed in Table 22 shall be thrown as exceptions.

DEVEMIT **FE 92**

(char dev —)

Transmit char to output device dev.

Because this device token does not return an ior, the ior codes listed in Table 22 shall be thrown as exceptions.

DEVREAD **FE 94**

(*addr len dev — ior)*

Read a string of len bytes from input device dev, returning a device-dependent ior. An ior of zero means success, and any other value is both device and implementation dependent. The size of an element within the string is device dependent. For example, on a keyboard each element is two bytes, an extension code in the first byte and the key value in the second byte. This token does not take into account the time-out set by the DEVIOCTL token. It waits forever until len bytes are received.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

If an odd buffer length is given to DEVREAD on a keyboard device, the terminal should behave as in the case of a buffer length that equals the original odd buffer length minus one.

DEVT IMEDREAD **FE 95**
(c-addr len1 dev – c-addr len2 ior)

Read a string of bytes from input device dev, returning a device-dependent ior. An ior of zero means success, and any other value is both device and implementation dependent. The size of an element within the string is device dependent. c-addr is the destination address for the string, and len1 is its maximum length in address units. On return, len2 gives the actual length of the string read in address units. If the requested number of elements are not received within the specified period of time, the function returns with ior -32766 (Time-out). A time-out of 0 will cause the function to return immediately. If the virtual machine has buffered enough data, the function will return immediately with ior 0 (Success). See Annex C for method of setting the time-out value.

If an odd buffer length is given to DEVTIMEDREAD on a keyboard device, the terminal should behave as in the case of a buffer length that equals the original odd buffer length minus one.

DEVWRITE **FE 96**
(*addr len dev — ior)*

Write a string to output device dev, returning a device-dependent ior. An ior of zero means success, and any other value is both device and implementation dependent.

DEVSTATUS **FE 97**
(dev – ior)

Return the status ior of the resource associated with device dev, where in the general case "ready" and "serviceable" is indicated by zero and "not ready" is indicated by any other value. A specific device may return non-zero ior values that have significance for that device, as detailed in Annex C. If the device is currently occupied because it has been selected by a previous execution of the DEVOUT PUT token, DEVSTATUS shall return the standard ior code for "device busy" (see Annex C) until execution of the procedure passed to DEVOUTPUT completes.

IECNORM.COM: Click to view the full PDF of ISO/IEC 20060:2010

DEVIOCTL **FE 98**
(a-addr num fn dev -- ior)

Perform DEVIOCTL function fn for device dev with num cell-sized arguments in the array at a-addr. Individual operations are device dependent and are defined against sup-ported devices in Annex C. Exception -21 (Unsupported operation) shall be thrown when in a particular implementation device dev is supported buf function fn is not. It is mandatory to implement every DEVIOCTL function that is supported by the underlying hardware.

DEVOUTPUT **FE 99 (xp**
dev – ior)

Execute the procedure whose execution pointer is given by xp with output being directed to device dev. On return from DEVOUT PUT the current output device is unaffected. Ior is zero if the procedure xp can be started on the device dev, and an appropriate non-zero value from Annex C if it cannot. All exceptions

© ISO/IEC 2010 – All rights reserved

arising from the execution of xp that are not trapped by a CATCH within the code of xp cause immediate termination of xp without an exception being thrown to any exception handler external to xp. xp is required to have no stack effect; if it does, terminal action is undefined.

DEVATXY **FE 9A**

(num1 num2 dev --)

Execute a device-dependent “set absolute position” action on device dev using num1 as the horizontal co-ordinate and num2 as the vertical co-ordinate. The co-ordinates of the upper left corner of a display are (0,0). Exception -32239 (Outside border) shall be thrown if the co-ordinates are outside the device's display region.

Because this device token does not return an ior, the ior codes listed in Table 22 shall be thrown as exceptions.

DEVCONNECT **FE 9B**

(c-addr len dev – ior)

Connect to remote system using device dev. As long as the connection is not established, DEVSTATUS applied on device dev returns ior -31720 (Connection in progress). If the connection is successfully established, DEVSTATUS returns ior 0. Every other ior indicates an error.

c-addr len is a string containing the access parameters (for example, telephone number plus modem command characters).

DEVHANGUP **FE 9C**

(dev – ior)

End the current modem session on the given device.

DEVBREAK **FE 9D**

(dev – ior)

Send a break on the connected modem session for the given device.

SETOP **F8**

(dev --)

Set the current output device (in user variable DEVOP) to dev.

GETOP **FE 9F**

(-- dev)

Return the device code dev for the current output device (stored in user variable DEVOP).

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

8.19 Formatted I/O Tokens

The following tokens provide support for Forth standard number conversion functions. The **NMBR** in the token names is pronounced “number”. Tokens **LTNMBR, NMBRS** and **TONUMBER** employ the user variable BASE (see Section 6.2.7) as the conversion number base.

LTNMBR **FA**

(--)

Initialise an internal buffer for number formatting.

NMBR **F9**

(ud_1 -- ud_2)

Convert the least significant digit of ud1 in the current number BASE to ASCII representation and add the resulting character to the left-hand end of the internal buffer for number formatting, leaving ud2 as the quotient of ud1 divided by the current value of BASE. Exception -17 (Pictured numeric string overflow) shall be thrown if the internal buffer for numeric output formatting is overrun. Exception -509 (Out of context) shall be thrown if token execution is not bracketed within execution of LTNMBR and NMBRGT tokens.

NMBRS **FE 32**

(ud1 -- ud2)

Convert at least one digit of ud1 according to the rule for NMBR, continuing to convert digits until ud2 becomes zero. Exception -17 (Pictured numeric string overflow) shall be thrown if the internal buffer for numeric output formatting is overrun. Exception -509 (Out of context) shall be thrown if token execution is not bracketed within execution of LTNMBR and NMBRGT tokens.

NMBRGT **FB**

(d – c-addr len)

Discard d, and return the address c-addr of the internal buffer for number formatting and the size len of the character string held there. Exception -509 (Out of context) shall be thrown if token execution is not preceded by execution of an LTNMBR token with no intervening NMBRGT tokens.

HOLDFE **33**

(char --)

Add char to the left-hand end of the internal buffer used for number formatting. Exception -17 (Pictured numeric string overflow) shall be thrown if the internal buffer for numeric output formatting is overrun. Exception -509 (Out of context) shall be thrown if token execution is not bracketed within execution of LTNMBR and NMBRGT tokens.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

SIGNFE **34**

(num --)

If num is less than zero, add an ASCII '-' character to the left-hand end of the internal buffer used for number formatting. Exception -17 (Pictured numeric string overflow) shall be thrown if the internal buffer for numeric output formatting is overrun. Exception -509 (Out of context) shall be thrown if token execution is not bracketed within execution of LTNMBR and NMBRGT tokens.

TONUMBER **FC**

(ud1 c-addr1 len1 -- ud2 c-addr2 len2)

Convert string to number. ud2 is the unsigned result of converting the characters within the string specified by c-addr1 len1 into digits, using the number in BASE, and adding each into ud1 after multiplying ud1 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any "+" or "-", is encountered or the string is entirely converted. c-addr2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. len2 is the number of unconverted characters in the string. Exception -24 (Invalid numeric argument) may be thrown if ud2 overflows during the conversion.

8.20 Integrated Circuit Card Tokens

Tokens in this group provide a mechanism for handling Integrated Circuit Card readers. Valid status codes returned in ior (I/O result) are given in Annex C.

CARDINIT **FE 72**

(dev – ior)

Select ICC reader *dev*. This does not affect the status of any other card reader in the system.

CARD **FE 73**

(c-addr1 len1 c-addr2 len2 – c-addr2 len3 ior)

Send the data in the buffer *c-addr*$_1$ *len*$_1$ to the card, and receive data at *c-addr*$_2$ *len*$_2$. The returned len3 gives the actual length of the string received. The two buffers may share the same memory region.

The format of the input and output buffer is the command response APDU format as specified in ISO/IEC 7816-4. The buffer *c-addr*$_2$ *len*$_2$ shall provide adequate space for the answer from the card plus two status bytes containing SW1 and SW2.

CARDON **FE 74**

(c-addr len1 – c-addr len2 ior)

Apply power to ICC and execute card reset function. The "Answer to Reset" message shall be returned in the buffer *c-addr len*$_2$; *len*$_1$ is the maximum length of this buffer.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

```
CARDOFF                                                   FE 75
     ( — )
```

Power off ICC. Executed when all transactions are complete.

Because this device token does not return an ior, the ior codes listed in Table 39 shall be thrown as exceptions.

```
CARDABSENT                                                   EF
     ( -- flag )
```

Return true if an ICC card is not present in the reader. This token should not be used to detect that a card has been inserted, only that it has been removed. CARDON should be used to detect initial insertion, because a card may be physically present but not responding.

Because this device token does not return an ior, the ior codes listed in Table 39 shall be thrown as exceptions.

8.21 Magnetic Stripe Tokens

Tokens in this group provide a mechanism for handling Magnetic Stripe devices. Valid status codes returned in ior (I/O result) are given in Annex C.

```
MAGREAD                                                   FE 76
     ( c-addr len1 u — c-addr len2 ior )
```

Read one or more ISO magstripes. The parameter track u is the ISO identifier of magstripe(s) to read, as shown in Table 13. Refer to ISO/IEC 7813 for a description of the format of this data. A particular device is not required to support all of these possibilities.

Table 13 — ISO parameter track selection codes

Track	Magstripes to read
1	ISO1
2	ISO2
3	ISO3
12	ISO1 and ISO2
13	ISO1 and ISO3
23	ISO2 and ISO3
123	ISO1, ISO2 and ISO3

c-addr is the destination address for the string and len1 is its maximum length. On return, len2 gives the actual length of the string read. Refer to ISO/IEC 7813 for a description of the data formats.

This token returns when either track data is available or the time-out set by the appropriate DEVIOCTL function is reached.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

The operation may return tracks read since the last execution of this token. If the VM has this track buffering capability, only one swipe of the card shall be buffered and all the track buffers will be cleared after their contents have been returned by this token (even if all tracks were not requested). The format returned by MAGREAD for each track is a track number (1,2 or 3), a length byte and the data whose size is specified by the length byte. If multiple tracks are requested, there are multiple instances of the above structure concatenated in the order they were requested. The data is returned in ASCII format with STX/ETX and LRC delimiters removed. Non-bcd digits are left unconverted.

For example:

```
BYTES FROM CARD         BYTES DELIVERED TO APPLICATION

B1 23 4D 7E 5F xx    => 31 32 33 34 0D 37 0E 35

 ^            ^ ^LRC

 STX          ETX
```

Exception -21 (Unsupported operation) shall be thrown if the reader does not support one or more of the requested tracks and the content of the returned buffer shall be undefined. If the requested tracks are supported by the reader but are not present on the card swiped, then ior 0 (Success) is returned and the buffer contains those tracks that are available on the card.

If an error occurs while reading one or more of the requested tracks supported by both reader and card swiped, ior -31215 (Transmission error between reader and magstripe) is returned and the length field of the corresponding tracks in the returned buffer is set to zero. In this case the data of the tracks that were read successfully is available in the returned buffer.

MAGWRITE **FE 77**

(c-addr len num – ior)

Write one ISO magstripe. The data is in the buffer c-addr len and shall be written to track num (1-3) when the user swipes the magstripe. If no card is swiped within the time-out period set by the appropriate DEVIOCTL function, the function returns with ior -32766 (Time-out).

The data is written in ASCII format with STX/ETX and LRC delimiters removed. Non-bcd digits are left unconverted.

For example :

```
BYTES DELIVERED BY APPLICATION    BYTES TO CARD

31 32 33 34 0D 37 0E 35         => B1 23 4D 7E 5F xx

                                   ^             ^ ^LRC

                                   STX           ETX
```

8.22 Socket Tokens

See Section 7.8 for a general account of the use of sockets.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

PLUGSOCKET **FE F3**

(xp u – flag)

Set the execution pointer *xp* to be the handler of socket procedure *u*. Subsequent execution of DOSOCKET *u* shall execute *xp*. Before the execution pointer is set, socket zero, PLUGGABLE, is run to determine whether the socket may be plugged with this new *xp*. *flag* is the value returned by PLUGGABLE. PLUGSOCKET shall set the handler only if *flag* is true, otherwise *xp* shall be discarded. Exception -3327 (Invalid socket) shall be thrown for values of u outside the range 0-63.

PLUGGABLE **CF 00**

(u -- flag)

Decide whether a socket *u* can be plugged. PLUGGABLE is itself socket zero (and is implemented as DOSOCKET followed by zero). *flag* is returned true if the socket can be plugged, or false if it cannot. Exception -3327 (Invalid socket) shall be thrown for values of u outside the range 0-63.

A default action of PLUGGABLE shall be installed by the virtual machine to return true for all values of *u*, enabling all sockets to be plugged.

DOSOCKET **CF u:8**

(–)

Perform the action of socket number *u,* where *u* is 1 through 63. All user-defined sock-ets, except socket zero, shall have no stack effect.

IDOSOCKET **FE F2**

(u –)

Execute the socket procedure whose socket number (0 through 63) is specified by u. All user-defined sockets, except socket zero, shall have no stack effect. Exception -3327 (Invalid socket) shall be thrown if *u* is outside the range 0-63.

8.23 Database Services Tokens

The following tokens provide a mechanism for handling databases as described in Section 7.3. Many of the tokens in this section involve user variables DBCURRENT (pointer to the current database DPB) and/or DBRECNUM (current record number).

DBMAKECURRENT **D0**

(a-addr --)

Make the database whose DPB is at *a-addr* the current database. This token sets DBCURRENT to *a-addr*.

DBSIZE **D3**

(-- len)

Return the size of the record buffer that provides the window onto the current record of the current database.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

DBFETCHLIT **D8 u:8**

(-- x)

Return the 32-bit value x from the cell at the in-line unsigned byte offset u in the current record of the current database. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

DBFETCH **FE E4**

(u -- x)

Return the 32-bit value x from the cell at byte offset u in the current record of the current database. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

DBSTORELIT **D9 u:8**

(x --)

Store the 32-bit value x in the cell at byte offset u in the current record of the current database and update the database record. Exception -4094 (Invalid function) shall be thrown if the current database is marked read-only. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

DBSTORE **FE E7**

(x u --)

Store the 32-bit value x in the cell at byte offset u in the current record of the current database and update the database record. Exception -4094 (Invalid function) shall be thrown if the current database is marked read-only. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

DBCFETCHLIT **D6 u:8**

(-- char)

Return the one-byte value char from the unsigned byte offset *u* in the current record of the current database. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

DBCFETCH **FE E5**

(u -- char)

Return the one-byte value char from byte offset *u* in the current record of the current database. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

```
DBCSTORELIT                                                  D7 u:8
    ( char -- )
```

Store the one-byte value *char* in the byte at the unsigned inline byte offset *u* in the current record of the current database and update the database record. Exception -4094 (Invalid function) shall be thrown if the current database is marked read-only. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL- 1 for the current database.

```
DBCSTORE                                                     FE E8
    ( char u -- )
```

Store the 1-byte value char in the byte at byte offset *u* in the current record of the current database and update the database record. Exception -4094 (Invalid function) shall be thrown if the current database is marked read-only. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

```
DBSTRFETCHLIT                                          D4 u:8 len:8
    ( -- c-addr len )
```

Return the string parameters *c-addr* and len of the byte sequence at in-line byte offset *u* in the current record of the current database. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

```
DBSTRFETCH                                                   FE E6
    ( u len -- c-addr len )
```

Return the string parameters *c-addr* and *len* of the byte sequence at byte offset u and with length *len* in the current record of the current database. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

```
DBSTRSTORELIT                                         D5 u:8 len2:8

    ( c-addr len1 -- )
```

Store at most *len2* bytes of the byte sequence at *c-addr* at byte offset *u* in the current record of the current database and update the database record. If *len1* is less than *len2* then the destination in the database record buffer shall be space filled to *len2*. Exception -4094 (Invalid function) shall be thrown if the current database is marked read-only. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL- 1 for the current database.

```
DBSTRSTORE                                                   FE E9
    ( c-addr len1 u len2 -- )
```

Store at most *len2* bytes of the byte sequence at *c-addr* at byte offset *u* in the current record of the current database and update the database record. If *len1* is less than *len2* then the destination in the database record buffer shall be space filled to *len2*. Exception -4094 (Invalid function) shall be thrown if the current database is marked read-only. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL- 1 for the current database.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

DBINIT **FE B4**

(--)

Delete all records in the database, erasing their data, and set “available” record number of the database (see DBAVAIL) to zero. Exception -4094 (Invalid database function) shall be thrown if the current database type is read-only or compiled. The currently selected record number DBRECNUM shall be set to -1 (invalid) by this operation.

DBAVAIL **D2**

(-- num)

Return the record number of the next available record in the current database.

DBADDREC **FE B0**

(--)

Add a record at the end of the current database, at the record number given by DBAVAIL. The new record becomes the current record for this database and its content is unspecified. Exception -4094 (Invalid database function) shall be thrown if the current database type is read-only, compiled, or ordered. Storage requirements are increased by this to-ken, and it shall cause Exception -3071 (Out of memory) to be thrown.

DBSELECT **D1**

(*num ––)*

Select record *num* in the currently selected database. This shall be within the range zero through DBAVAIL-1 inclusive for the current database. This token sets DBRECNUM to num.

DBMATCHBYKEY **FE B1**

(c-addr len -- flag)

Search the current database for a match on the key field against the string specified by *c-addr* and *len*. *len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 20H) characters. If the match is successful, the matching record becomes current and flag is true.

Exception -4094 (Invalid database function) shall be thrown if the current database type is not Ordered.

DBADDBYKEY **FE B5**

(c-addr len -- flag)

Search the current database for a match on the key field against the string specified by *c-addr* and *len*. *len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 20H) characters. If the match is successful, the matching record becomes current and *flag* is *false*. If the match is not successful, a new record shall be inserted at the correct position in the database and the *flag* is *true*. The content of the new record is unspecifed except for

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

its key field, which shall contain the given key, and becomes the current record for this database. Storage requirements are increased by this token, and it shall cause Exception -3071 (Out of memory) to be thrown.

Exception -4094 (Invalid database function) shall be thrown if the current database type is not ordered, or if it is read-only or compiled.

DBDELBYKEY **FE B6**

(c-addr len -- flag)

Search the current database for a match on the key field against the string specified by *c-addr* and *len*. *len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 20H) characters. If the match is successful, the matching record shall be deleted and flag is true. Otherwise, false is returned. The deletion action closes up any potential “hole” in a physical implementation by taking appropriate action to physically reposition or relink the records in the database. The currently selected record number shall be set to -1 (invalid) by this operation. Storage requirements are decreased by this token.

Exception -4094 (Invalid database function) shall be thrown if the current database type is not ordered, or if it is read-only or compiled.

DBSAVE **DA**

(R: -- u1 u2)

Save the current database context on the return stack. *u1* is DBCURRENT, and *u2* is DBRECNUM.

DBRESTORE **DB**

(R: u1 u2 --)

Restore the most-recently-saved database context information from the return stack. *u1* is DBCURRENT, and *u2* is DBRECNUM. The data were pushed by DBSAVE.

DBDELREC **FE B3**

(--)

Delete the current record from the current database. Reset the current record pointer to -1 (invalid). The deletion action closes up any potential “hole” in a physical implementation by taking appropriate action to physically reposition or relink the records in the database. Storage requirements are decreased by this token.

Exception -4094 (Invalid database function) shall be thrown if the current database type is read-only or compiled. Exception -4095 (Invalid record number) shall be thrown if the current record number DBRECNUM is not within the range of zero to DBAVAIL-1 for the current database.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

8.24 Language and Message Tokens

Tokens in this group provide a mechanism for handling language and message selection and display.

`CHOOSELANG` **`FE 84`**

`( c-addr -- flag )`

Select the language whose ISO 639 language code is given by the two characters at *c-addr*. If flag is true, the language was found and is now the current language. Otherwise, the calling program should select another language. At least one language (the terminal's native language) shall always be available. It is the responsibility of the program using CHOOSELANG to make the current language selection available to any client program using MSGLOAD.

`CODEPAGE` **`FE 85`**

`( u -- flag )`

Select the resident code page *u.* Code pages are numbered according to ISO/IEC 8859 (0 = common character set, 1 = Latin 1, etc.). *flag* is *true* if the code page has been selected.

`LOAD PAGE` **`FE 87`**

`( c-addr -- flag )`

Install the code page at *c-addr* in the terminal. A *true flag* indicates a successful installation.

`MSGINIT` **`FE 86`**

`( -- )`

Erase the transient messages, numbered from 40_H to FF_H.

`MSGLOAD` **`FE 83`**

`( c-addr – flag )`

Install a message table at *c-addr*, including message numbers in the range 40-FFH in the transient messages buffer, over-writing any previous messages with the same message numbers. Messages 40-BFH may be provided only by a terminal application; messages C0-FFH may be provided by an ICC. The message table shall be formatted as described in Section 4.4.

flag is *true* if the messages were loaded successfully, *false* if the message table's code page is not currently selected or if the messages are outside the range 40-FFH.

`MSGFETCH` **`FE 80`**

`( u – c-addr len )`

Return string parameters for message *u*. Trailing spaces are removed from the length *len* of the string. This string address becomes invalid if another language or another message is selected.

If message *u* has not been provided in the currently selected language, the VM shall return string parameters for a zero-length message.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

LANGUAGES **FE 81**

(—)

Make the language database the current database. This is a read-only, non-volatile data-base.

MSGUPDATE **FE 88**

(c-addr —)

Install a message table including message numbers in the range 1-FF H at *c-addr* into the resident language database. If a language with the same code is already present, new messages replace previous messages with the same number. Exception -510 (Language file full) shall be thrown if there is not sufficient space for the new language.

c-addr gives the location of the message table definition, formatted as described in Section 7.4.

MSGDELETE **FE 89**

(*c-addr — flag)*

Delete the language database entry and resident messages for the language whose ISO 639 language code is given by the two bytes at *c-addr*. The flag is true if the language was found and deleted successfully, *false* if the language was not found.

8.25 TLV Tokens

The tokens described in this section provide TLV management and access functions as de-scribed in Section 7.5.

8.25.1 TLV Buffer Access

TLVINIT **FE D2**

(—)

Clear all internally-maintained data associated with TLV definitions and set the status of all TLV definitions to *not assigned*. Only bit zero (the *not assigned* bit) is cleared in the TLV status character; other status bits are not affected. This command is provided to satisfy the security requirement that an application shall not be able to access data belonging to another application.

TLVFETCHRAW **FE D3**

(a-addr — c-addr len)

Return the data assigned to the TLV definition whose access parameter is *a-addr*, without applying any conversion, as the binary string *c-addr len*. The format of the data returned is the same as the value field format of the corresponding TLV. The *len* returned for a TLV which has not had data assigned to it shall be zero.

TLVSTORERAW **FE DE**

(c-addr len a-addr —)

Assign the data at the binary string *c-addr len*, without applying any conversion, to the TLV definition whose access parameter is *a-addr*. The format of the data supplied shall be the same as the value field format of the corresponding TLV.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

TLVFIND **C0**

(u — a-addr | 0)

Return the access parameter for the TLV definition whose tag is *u*. If there is no TLV defined for tag *u*, return zero.

TLVFETCH **C1**

(a-addr — u | c-addr len)

Return the data assigned to the TLV definition whose access parameter is *a-addr.* Apply a conversion according to the associated type field. Type codes 0 and 2 return an unsigned number u on the stack, while the others return string parameters *c-addr len*. For type 3, the address returned is temporary and the string becomes invalid when there is any other access to data associated with the same type, or the CNFETCH token is executed. The len returned for strings shall be the same as that last stored in the buffer. The *len* or *u* returned for a TLV which has not had data assigned to it shall be zero.

NOTE If any change is made to the data returned by this operator, it may not be recorded until a TLVS TORE is performed.

TLVSTORE **C2**

(u a-addr | c-addr len a-addr —)

Apply a conversion according to the associated type field of the TLV definition whose access parameter is a-addr. Type codes 0 and 2 take an unsigned number u on the stack, while the others take string parameters c-addr len. Assign the converted data to the TLV record and set the assigned status bit for this TLV. An ambiguous condition exists if the data format does not match the defined type.

TLVBITFETCH **FE D0**

(*u a-addr — flag)*

Return the status of bit u in the data assigned to the TLV definition whose access parameter is *a-addr*. *u* = (bit position) + 8 * (byte position); *u* = 0 for LSB of Byte 1. The *flag* is *true* if the referenced bit was set. Otherwise, *false* is returned.

An ambiguous condition exists if the TLV is not type 1 or if *u* is larger than the bits contained in the assigned data.

TLVBI TSTORE **FE D1**

(flag u a-addr —)

Modify bit u in the data assigned to the TLV definition whose access parameter is *a-addr*. *u* = (bit position) + 8 * (byte position); u = 0 for LSB of Byte 1. If *flag* is *false* (zero), then the bit shall be turned off (zero). Otherwise, it shall be turned on (one).

An ambiguous condition exists if the TLV is not type 1 or if *u* is larger than the bits contained in the assigned data.

TLVBI TSET **FE DC**

(u a-addr —)

Set (to one) bit *u* in the data assigned to the TLV definition whose access parameter is *a-addr. u* = (bit position) + 8 * (byte position); *u* = 0 for LSB of Byte 1.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

An ambiguous condition exists if the TLV is not type 1 or if *u* is larger than the bits contained in the assigned data.

TLVBI TCLEAR **FE DD**

(u a-addr —)

Clear (to zero) bit *u* in the data assigned to the TLV definition whose access parameter is *a-addr*. *u* = (bit position) + 8 * (byte position); *u* = 0 for LSB of Byte 1.

An ambiguous condition exists if the TLV is not type 1 or if *u* is larger than the bits contained in the assigned data.

8.25.2 TLV Processing

TLVPARSE **C3**

(c-addr len —)

Process *len* bytes at *c-addr* for TLV sequences, expecting a valid combination of primitive or constructed TLV definitions. For each TLV encountered in the string, the value field is assigned to the TLV definition which is associated with that tag. Exception -507 (String too large) shall be thrown if the tag, length, and value fields are not entirely contained within the input string, or if the length field indicates a value field larger than 252 bytes. For each successfully parsed TLV, the assigned status bit in the definition record shall be set. If a tag is not found no exception is thrown. When a constructed TLV is encountered, whether or not its tag is found, its value field shall be recursively parsed for embedded TLV sequences.

TLVTRAVERSE **F3**

(c-addr len xp —)

Process *len* bytes at *c-addr* for TLV sequences, expecting a valid combination of primitive or constructed TLV definitions. For each TLV encountered in the string, the tag and the whole of the value field is sent to the method xp. Exception -507 (String too large) shall be thrown if the tag, length, and value fields are not entirely contained within the input string. When a constructed TLV is encountered, its tag and value field shall be passed to method xp and then the value field recursively traversed for embedded TLV sequences which are, in turn, passed to xp.

The stack effect of *xp* is required to be (*c-addr len u* —), where *u* is the TLV's tag and *c-addr len* is the value field.

TLVPLUSDOL **FE DA**

(c-addr1 len1 c-addr2 len2 — c-addr2 len3)

Process *len1* bytes at *c-addr1* for a sequence of tag and length fields. For all tag fields encountered, the corresponding value fields are concatenated into the buffer at *c-addr2* up to its maximum size, *len2*.

The parts of the resulting string are filled or truncated where a specified length field differs from the size of the existing value. For more details see Annex E.

Exception -507 (String too large) shall be thrown if *len2* is exceeded. The token returns the beginning of the destination string (*c-addr2*) and the resulting output length (*len3*, not greater than *len2*).

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

TLVPLUS STRING **FE D9**

(c-addr len1 a-addr — c-addr len2)

Retrieve tag, length, and assigned value from the TLV definition whose access parameter is *a-addr*. Append a well-formed TLV sequence to the end of the output string at *c-addr len1*. Append an empty TLV if no data has been assigned. Return the beginning of the destination string (*c-addr*) and the sum of the two lengths (*len2*). It is the programmer's responsibility to ensure that there is sufficient room at the end of the output string to hold both strings.

TLVTAG **FE D8**

(a-addr — u)

Return the tag number for the TLV definition whose access parameter is *a-addr*.

TLVFORMAT **FE D7**

(a-addr -- fmt)

Return the format code associated with the TLV definition whose access parameter is *a-addr*. The returned *fmt* is the format indicator 0-6 (see Section 7.5.2).

TLVSTATUS **C4**

(a-addr — char)

Return the status of the TLV definition whose access parameter is *a-addr*. The bits in the returned char have the following significance, where bit 0 is the least significant bit:

0	0 = value field not assigned, 1 = value field assigned
1-7	Reserved for future use

8.25.3 TLV Sequence Access

TLVFETCHTAG **FE D5**

(c-addr1 len1 — c-addr2 len2 u flag)

Parse the input string at *c-addr1* for a tag field. Return the string *c-addr2 len2* and the decoded tag value *u*. *c-addr2* points to the byte immediately following the tag, and *len2* is *len1* minus the length of the tag. The flag is true if the tag is entirely contained within the input string.

TLVFETCHLENGTH **FE D6**

(c-addr1 len1 — c-addr2 len2 u flag)

Parse the input string at *c-addr1* for a length field. Return the string *c-addr2 len2* and the decoded length value *u*. *c-addr2* points to the byte immediately following the length, and *len2* is *len1* minus the size of the length field. The *flag* is *true* if the length is entirely contained within the input string.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

TLVFETCHVALUE **FE D4**

(c-addr1 len1 – c-addr2 len2 c-addr3 len3 flag)

Split the input string at *c-addr1*, which is expected to contain the length and value fields of a TLV. Decode the length field. Return the trailing string *c-addr2 len2*, the value field *c-addr3 len3* and a *true* flag if the information in the length and value fields is entirely contained within the input string. Otherwise, return *false* and the remaining parameters are undefined.

TLVCLEAR **FE DB**

(a-addr –)

For the TLV whose access parameter is *a-addr*, clear the assigned status bit and remove any assignment of data associated with the TLV definition.

8.26 Hot Card List Tokens

Tokens in this group provide a mechanism for handling the hot card list file. See the discussion in Section 7.6 for a complete description of the search mechanism. All of the *len* parameters in this section refer to bytes, not data elements, and so the *len* of a PAN is one-half the number of its digits, rounded up.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

HOTINIT **FE E0**

(–)

Initialise the hot card list to an empty state.

HOTADD **FE E1**

(c-addr len – flag)

Add an entry to the list, where *c-addr len* is the data for the entry. The data may be up to 10 bytes long. If it is shorter it shall be padded with trailing FH's. This token shall be used when updating the list.

The returned *flag* is *true* if the addition was successful. Possible reasons for failure to add include: no room in list; entry with exactly the same PAN (including wildcards) already exists; data for the entry is invalid (e.g. contains wildcard(s) followed by digit(s)).

HOTDELETE **FE E2**

(c-addr len – flag)

Delete the entry with data *c-addr len* from the list. An entry shall be deleted from the list only if there is an exact match; no wild card search is performed.

The returned *flag* is *true* if the deletion was successful (the entry was found).

HOTFIND **FE E3**

(c-addr len –flag)

Search for an entry in the list matching the input PAN *c-addr len*. The input data may be up to 10 bytes long.

The returned *flag* is *true* if the list contained a matching entry (i.e., identical up to the first FH in the entry; see Section 7.6).

© ISO/IEC 2010 – All rights reserved

8.27 Cryptographic Algorithm Token

This token provides support for using cryptographic services. The supported cryptographic algorithms are discussed in more detail in Section 7.7.

CRYPTO **FE 70**

(*i*x num — j*x)*

Apply the cryptographic algorithm specified by *num* with parameters *i*x* and return results *j*x.* The values of *num* are specified in Table 11. Specific stack parameters for each algorithm are given in Section 7.7.1 and following.

8.28 Module Management Tokens

The following tokens provide for the storage and execution of OTA modules in the virtual machine. See Section 7.9 for a general account of module handling.

MODEXECUTE **FE C1**

(i*x c-addr len – j*x flag)

Load and execute a module from the module repository using the module ID specified by *c-addr len*. *flag* is *true* if the module was loaded successfully. *flag* is only related to the load phase of the module ('Load Module' box in Figure 4). From the moment the module is successfully loaded, *flag* is set to *true*. When a token inside the module throws an exception, MODEXECUTE also throws the exception.

MODINIT **FE C2**

(–)

Prepare for reception of a new module into the repository. Details are implementation-dependent.

MODAPPEND **FE C4**

(c-addr len –)

Append the contents of the buffer defined by *c-addr* and *len* to the module acquisition buffer. Exception -3071 (Out of memory) shall be thrown if the module buffer capacity is exceeded.

MODREGISTER **FE C7**

(– ior)

Register the module buffer in the module repository under the module's ID, which is found in its header. The resources associated with managing the module buffer are automatically released. *ior* is zero if the operation succeeded; for other *ior* values, see Annex B.

MODRELEASE **FE C8**

(–)

Release the resources used by the internal module buffer. This is required if premature loading of a module shall be terminated by the application without registering the module in the module repository.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

MODDELETE **FE C0**

(c-addr len — ior)

Delete the module whose ID is specified by *c-addr len* from the module repository. *ior* is zero if the operation succeeded; for other *ior* values see Annex B.

MODCARDEXECUTE **FE C3**

(i*x a-addr — j*x flag)

Load the module at *a-addr*. *a-add*r is the address of an OTA module delivered from the card into internal storage. *flag* is *true* if the module loaded successfully. *flag* is only related to the load phase of the module ('Load Module' box in Figure 6). From the moment the module is successfully loaded, flag is set to true. When a token inside the module throws an exception, MODCARDEXECUTE also throws the exception.

MODCHANGED **FE C6**

(— u)

Return a value u indicating whether any classes of modules (see Section 6.1) have been updated. Bits 0 through 7 are separately set if a module in the class Fn has been registered in the module repository since the last execution of this token. For example, a module registered with an initial module ID byte of F4 will set bit 4 in the return status. Bits 8 through 31 are reserved for future expansion.

MODULES **FE C9**

(—)

Make the modules database the current database. This is a read-only non-volatile data-base.

8.29 Operating System Interface Tokens

OSCALL **FE 66**

(a-addr num fn —)

Call an operating system function *fn* with num cell-sized arguments in the array at *a - addr*. Individual functions are terminal dependent and are defined in Annex D. Exception -21 (Unsupported operation) shall be thrown if a particular function is not supported by the virtual machine.

SETCALLBACK **FE 65**

(xp —)

Announce xp as an OTA routine that may be called by the underlying operating system.

NOTE This token is implementation dependent, and is provided so that terminal specific programs (TRS) written using OTA tokens can provide a single call-back routine for the operating system.

8.30 Miscellaneous Tokens

NOOP **2F**

(—)

Take no action.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

DEPTH **FE 42**

(— num)

Return the data stack depth in cells.

PROC **FE 00**

(—)

Mark the start of a high-level definition. Language translators are required to do this. The token has no effect if executed and will not appear in an optimised token program.

END PROC **FE 01**

(—)

Mark the end of a high-level definition. Language translators are required to do this. The token has no effect if executed and will not appear in an optimised token program.

BREAKPNT **FF**

(—)

Cause a breakpoint, which is handled by special debugger code. For production kernels this token shall be implemented as a NO-OP.

HEADER **FE 02 counted-string**

(—)

Begin a header string that is compiled into the token stream. The header token is followed by a counted string, which is stored in the token stream as a count byte followed by that many bytes.

At runtime, interpretation of HEADER should skip the string and then execute the token immediately following the string.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

9 Module Delivery Format

To produce an OTA program, one uses a compiler that generates as its output one or more modules containing tokens representing instructions for the OTA virtual machine (see Section 5). These modules are subsequently processed by the OTA kernel in a terminal to execute the program. This section describes the format for the modules produced by an OTA compiler.

Table 14 — Module delivery format

Label	Size (bytes)	Content
MDF-VER	2	Version number of the module.
MDF-FLAGS	1	VM flags, set to zero by the compiler. This field is used internally by the VM after the module is delivered.
MDF-IDLEN	1	Actual number of bytes taken by the Module ID in the **MDF-ID** field.
MDF- ID	16	Module ID. Shall be blank filled.

MDF-TLEN	4	Length in bytes of the token image **MDF-TI.**
MDF-ILEN	4	Length in bytes of initialised data section **MDF-II.**
MDF-ULEN	4	Length in bytes of uninitialised data section.
MDF-RLEN	2	Length in bytes of the relocation section **MDF-RELOC.**
MDF-PLEN	2	Length in bytes of the procedure list **MDF- PROC.**
MDF-SLEN	2	Length in bytes of the socket list **MDF- SOCK.**
MDF-EXLEN	2	Length in bytes of the export list **MDF-EXL.**
MDF-IMLEN	4	Length in bytes of the import list **MDF-IML.**
MDF-TLVROOT	4	Pointer to the link field of the root of the balanced binary tree of TLV definitions in module initialised data **(MDF-II).** If **MDF-TLVROOT** is -1, the module has no TLV definitions.
MDF-DBROOT	4	Pointer to the head of the database parameter block list in module initialised data **(MDF-II).** If **MDF-DBROOT** contains -1, the module has no database definitions.
MDF-ENTRY	4	Token entry point in **MDF-TI.**
MDF-TI	**MDF-TLEN**	Token image.
MDF-II	**MDF-ILEN**	Initialised data.
MDF-RELOC	**MDF-RLEN**	Relocation map.
MDF- PROC	**MDF-PLEN**	Procedure entry list.
MDF- SOCK	**MDF-SLEN**	Socket list.
MDF-EXL	**MDF-EXLEN**	Module export list (see Table 18).
MDF-IML	**MDF-IMLEN**	Module import list (see Table 17).

Module code is stored as a byte stream. The byte stream is defined in Table 14. All numbers larger than one byte are in big-endian form. A value for **MDF-ENTRY** of -1 specifies that this is a library module that will not be directly executed when loaded (all calls to its procedures are through its Module Export List).

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

9.1 Module ID Format

The actual length of the Module ID held in **MDF- ID** may vary from 5 to 16 bytes (as specified by **MDF-IDLEN).** The initial byte of the Module ID contains its *class.* Module classes in the range F0$_H$ through FF$_H$ are for non EMV application modules, and the OTA system reserves classes F0$_H$ through F7$_H$ for its own use. The Module ID is identical to the Application Identifier (AID) as specified in ISO/IEC Standard 7816-5, and its structure and formatting follows ISO rules. The Module ID F100000000 $_H$ is reserved for the startup module that will be called from the virtual machine.

9.2 Socket List

Each module shall contain a list of the sockets that are to be modified before the module is executed. Socket number zero should not appear in this list; if it does, the entry containing it shall be ignored.

The socket list contained in **MDF- SOCK** consists of a sequence of socket numbers followed by the offset in the token image of the token to be plugged into that socket when the module is loaded.

© ISO/IEC 2010 – All rights reserved

Table 15 — Socket list in Module Delivery Format

Label	Size (bytes)	Content
MDF-SNUM1	1	Socket number
MDF-SADDR1	4	Token address of code for socket service
MDF-SNUM*n*	1	Socket number
MDF-SADDR*n*	4	Token address of code for socket service

9.3 Relocation Section

Relocation is necessary for the initialised data section, since data objects may have different types: strings or single characters, cells, or pointers. The token loader/interpreter needs to know what type each object is so that it can be dealt with appropriately. For example, it may need to byte-swap cells or translate code pointers to physical addresses.

This section describes the data type information in the **MDF-RELOC** field of a module. This information enables the initialised data in the **MDF- II** field to be suitably relocated or other-wise modified for execution.

Cells may contain any of five types of data: pointers into the code image, pointers into the initialised data section, pointers into the uninitialised data section, straight 32-bit values, or a sequence of bytes. All pointers and cells are aligned on a cell boundary.

Four bits are used to encode the type of data the cell holds:

Table 16 — Relocation specification

Relocation Nibble	Relocation Action
0000	Cell contains a sequence of four bytes.
0001	Cell contains a 32-bit value.
0010	Cell contains a 32-bit offset from the start of the code section.
0011	Cell contains a 32-bit offset from the start of initialised data.
0101	Cell contains a 32-bit offset from the start of uninitialised data.

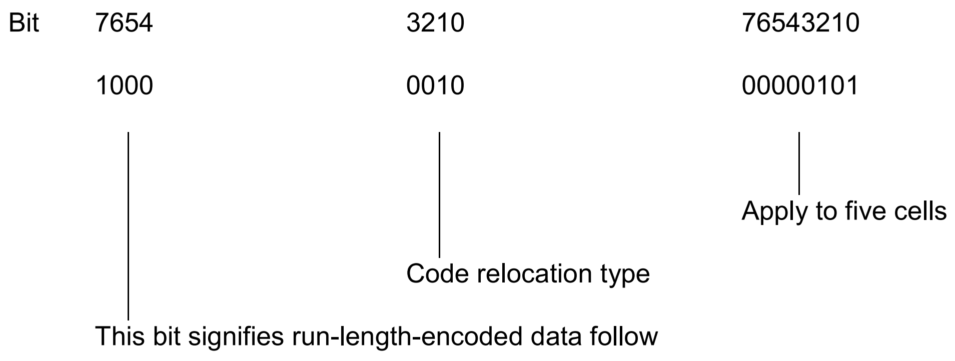
The relocation data is held as a sequence of bytes that shall be processed.

Each byte in the relocation section is processed until the end of the relocation section is reached. The length of the relocation section is contained in the header field **MDF-RLEN.**

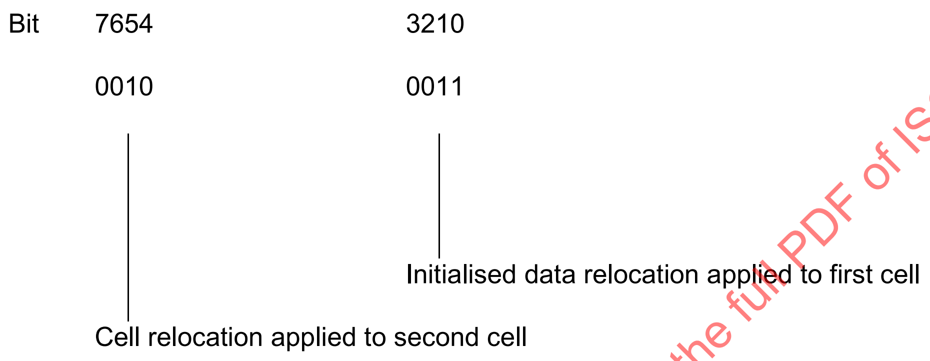
If the relocation byte has bit seven set, the low four bits define the relocation type to apply. The next byte in the relocation section defines the number of cells in the image this relocation applies to.

If the relocation byte does not have bit seven set, the low-order nibble defines the relocation type to apply to the next cell in the image, and the high-order nibble defines the type to apply to the cell after that.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010



If the relocation byte does not have bit seven set, the low-order nibble defines the relocation type to apply to the next cell in the image, and the high-order nibble defines the type to apply to the cell after that.


Bit
7654
3210
0010
0011
Initialised data relocation applied to first cell
Cell relocation applied to second cell

9.4 Module Import List

The module import list **MDF-IML** comprises an array of (**MDF-IMLEN**/17) records of the following format:

Table 17 — Module import list format

Label	Size (bytes)	Content
MDF-IDLEN	1	Actual number of bytes taken by the module ID in the **MDF-ID** field
MDF- ID	16	Module ID of the imported module, blank filled

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

9.5 Module Export List

The module export list **MDF-EXL** comprises an array of (**MDF-EXLEN**/4) records of the following format, where the first array entry represents exported procedure 0, the next procedure 1, and so on to a maximum of 256 entries:

Table 18 — Module export list format

Label	Size (bytes)	Content
MDF-EXPROC	4	Token address of exported procedure in token image **MDF- TI**

© ISO/IEC 2010 – All rights reserved

9.6 Module Procedure List

The module procedure list **MDF- PROC** comprises an array of **(MDF-PLEN**/4) records of the following format, where the first array entry represents procedure 0, the next procedure 1, and so on to a maximum of 40 entries:

Table 19 — Module procedure list format

Label	Size (bytes)	Content
MDF- PROC	4	Token address of procedure in token image **MDF-TI**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Annex A
(normative)

OTA Token Lists

A.1 General

This section presents numeric lists of tokens, with their numeric codes and names. Descriptions of the individual tokens may be found in Section 8 at the pages shown.

A.2 Numeric List of Tokens

Numeric code	Name	Numeric code	Name
00 … 27	**CALL0 … CALL39**	6D *u:8*	**SLIT**
28 *ofset:8*	**SCALL**	6E *u:16*	**LIT**
29 *ofset:16*	**CALL**	6F *num:32*	**ELIT**
2A *u1:8 u2:8*	**IMCALL**	70 *u:8* … 73 *u:8*	**SLITD0 … SLITD3**
2B	**ICALL**	74 *u:8* … 77 *u:8*	**FETCHD0 … FETCHD3**
2C	**RETURN**	78 *u:8* … 7B *u:8*	**STORED0 … STORED3**
2D	**CATCH**	7C *u:16*	**LITD**
2E	**THROW**	7D *ofset:16*	**LITC**
2F	**NOOP**	7E	**LITMINUS1**
30 … 3F	**LIT0 … LIT15**	7F *u:8*	**NLIT**
40 … 43	**PFRFETCH2 … PFRFETCH5**	80 *ofset:8*	**SBZ**
44 … 4F	**TFRFETCH12 … TFRFETCH1**	81 *ofset:16*	**BZ**
50 … 53	**PFRSTORE2 … PFRSTORE5**	82 *ofset:8*	**SBNZ**
54 … 5F	**TFRSTORE12 … TFRSTORE1**	83 *ofset:16*	**BNZ**
60 *u:8* … 63 *u:8*	**SLITU0 … SLITU3**	84 *ofset:8*	**SBRA**
64 *u:8* … 67 *u:8*	**FETCHU0 … FETCHU3**	85 *ofset:16*	**BRA**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Numeric code	Name	Numeric code	Name
68 *u:8* … 6B *u:8*	**STOREU0 … STOREU3**	86 *u:8 ofset:8*	**SROFLIT**
6C *u:16*	**LITU**	87 u:16 ofset:8	**ROFLIT**
88 ofset:16	**RDO**	A0	**TWORFROM**
89 ofset:16	**RQDO**	A1	**TWOOVER**
8A	**RI**	A2	**TWORFETCH**
8B	**RLEAVE**	A3	**FETCH**
8C	**RLOOP**	A4	**STORE**
8D	**RPLUSLOOP**	A5	**CFETCH**
8E ofset:16	**QUOTE**	A6	**CSTORE**
8F	**ENDQUOTE**	A7	**BCDFETCH**
90	**DROP**	A8	**BCDSTORE**
91	**DUP**	A9	**ADD**
92	**SWAP**	AA	**SUB**
93	**OVER**	AB	**MUL**
94	**NIP**	AC	**MOD**
95	**TUCK**	AD	**AND**
96	**ROT**	AE	**OR**
97	**MINUSROT**	AF	**MSLMOD**
98	**QDUP**	B0	**SHRNU**
99	**RFROM**	B1	**SHL**
9A	**TOR**	B2	**SHLN**
9B	**RFETCH**	B3	**CMPEQ**
9C	**TWOSWAP**	Numeric code	**Name**
9D	**TWODROP**	B4	**CMPNE**
9E	**TWODUP**	B5	**CMPLT**
9F	**TWOTOR**	B6	**CMPLTU**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Numeric code	Name	Numeric code	Name
B7	**CMPGT**	CA	**PLUSSTRING**
B8	**CMPGEU**	CB	**COUNT**
B9	**CMPGTU**	CC	**INCR**
BA	**SETEQ**	CD	**BNFETCH**
BB	**SETLT**	CE	**BNSTORE**
BC	**SETNE**	CF 00	**PLUGGABLE**
BD	**WITHIN**	CF u:8	**DOSOCKET**
BE num2:8	**SADDLIT**	D0	**DBMAKECURRENT**
BF u:8	**SMULLIT**	D1	**DBSELECT**
C0	**TLVFIND**	D2	**DBAVAIL**
C1	**TLVFETCH**	D3	**DBSIZE**
C2	**TLVSTORE**	D4 u:8 len:8	**DBSTRFETCHLIT**
C3	**TLVPARSE**	D5 u:8 len2:8	**DBSTRSTORELIT**
C4	**TLVSTATUS**	D6 u:8	**DBCFETCHLIT**
C5	**MOVE**	D7 u:8	**DBCSTORELIT**
C6	**FILL**	D8 u:8	**DBFETCHLIT**
C7	**COMPARE**	D9 u:8	**DBSTORELIT**
C8	**MINUSTRAILING**	DA	**DBSAVE**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Numeric code	Name	Numeric code	Name
DB	**DBRESTORE**	E6 E5 num:16	**BYTE FRSTORE**
DD	**ADDLIT1**	E7	**WIDEN**
DE	**SUBLIT1**	E8 u1:8 u2:8	**SMAKEFRAME**
DF u:16 ofset:16	**DOCLASS**	E9	**RELFRAME**
E0 num:8	**SFRADDR**	EA	**CEXTEND**
E1 num:8	**SFRFETCH**	EB	**RELEASE**
E2 num:8	**SFRSTORE**	EC	**GETTIME**
E3 num:16	**FRADDR**	ED	**GETMS**
E4 num:16	**FRFETCH**	EE	**MS**
E5 num:16	**FRSTORE**	EF	**CARDABSENT**
E6	**BYTE**	F0	**IJMP**
E6 44 … E6 4F	**BYTE TFRFETCH12 …** **BYTE TFRFETCH1**	F1	**NEGATE**
E6 54 … E6 5F	**BYTE TFRSTORE12 …** **BYTE TFRSTORE1**	F2 counted-string	**STRLIT**
E6 64 u:8 … E6 67 u:8	**BYTE FETCHU0 …** **BYTE FETCHU3**	F3	**TLVTRAVERSE**
E6 68 u:8 … E6 6B u:8	**BYTE STOREU0 …** **BYTE STOREU3**	F8	**SETOP**
E6 74 u:8 … E6 77 u:8	**BYTE FETCHD0 …** **BYTE FETCHD3**	F9	**NMBR**
E6 78 u:8 … E6 7B u:8	**BYTE STORED0 …** **BYTE STORED3**	FA	**LTNMBR**
E6 E1 num:8	**BYTE SFRFETCH**	FB	**NMBRGT**
E6 E2 num:8	**BYTE SFRSTORE**	FC	**TONUMBER**
E6 E4 num:16	**BYTE FRFETCH**	FD	**USERVAR**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Numeric code	Name	Numeric code	Name
FE	**SECONDARY**	FE 35	**MINUSZEROS**
FE 00	**PROC**	FE 36	**EXTEND**
FE 01	**ENDPROC**	FE 37	**RJ**
FE 02 counted-string	**HEADER**	FE 38	**SLASHSTRING**
FE 10	**MIN**	FE 40	**SCAN**
FE 11	**MAX**	FE 41	**SKIP**
FE 12	**ABS**	Numeric code	**Name**
FE 13	**CMPLE**	FE 42	**DEPTH**
FE 14	**CMPLEU**	FE 43	**PICK**
FE 15	**SETGE**	FE 44	**TWOROT**
FE 17	**CMPGE**	FE 45	**CNFETCH**
FE 18	**SETGT**	FE 46	**CNSTORE**
FE 19	**SETLE**	FE 50	**XOR**
FE 20	**DADD**	FE 51	**DIV**
FE 21	**DCMPLT**	FE 52	**DIVU**
FE 22	**DNEGATE**	FE 53	**MODU**
FE 30	**TWOFETCH**	FE 54	**MSLMODU**
FE 31	**TWOSTORE**	FE 55	**MMUL**
FE 32	**NMBRS**	FE 56	**MMULU**
FE 33	**HOLD**	FE 57	**SHRN**
FE 34	**SIGN**	FE 60 ofset:32	**EBRA**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Numeric code	Name	Numeric code	Name
FE 61 ofset:32	**ECALL**	FE 87	**LOADPAGE**
FE 62 ofset:32	**EBNZ**	FE 88	**MSGUPDATE**
FE 63 ofset:32	**EBZ**	FE 89	**MSGDELETE**
FE 64 u1:16 u2:16	**MAKEFRAME**	FE 90	**DEVEKEY**
FE 65	**SETCALLBACK**	FE 91	**DEVEKEYQ**
FE 66	**OSCALL**	FE 92	**DEVEMIT**
FE 67 ofset:16	**ROF**	FE 93	**DEVOPEN**
FE 70	**CRYPTO**	FE 94	**DEVREAD**
FE 72	**CARDINIT**	FE 95	**DEVTIMEDREAD**
FE 73	**CARD**	FE 96	**DEVWRITE**
FE 74	**CARDON**	FE 97	**DEVSTATUS**
FE 75	**CARDOFF**	FE 98	**DEVIOCTL**
FE 76	**MAGREAD**	FE 99	**DEVOUTPUT**
FE 77	**MAGWRITE**	FE 9A	**DEVATXY**
FE 79	**SETTIME**	FE 9B	**DEVCONNECT**
FE 80	**MSGFETCH**	FE 9C	**DEVHANGUP**
FE 81	**LANGUAGES**	FE 9D	**DEVBREAK**
FE 83	**MSGLOAD**	FE 9E	**DEVCLOSE**
FE 84	**CHOOSELANG**	FE 9F	**GETOP**
FE 85	**CODEPAGE**	FE B0	**DBADDREC**
FE 86	**MSGINIT**	FE B1	**DBMATCHBYKEY**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Numeric code	Name	Numeric code	Name
FE B3	**DBDELREC**	FE C8	**MODRELEASE**
FE B4	**DBINIT**	FE C9	**MODULES**
FE B5	**DBADDBYKEY**	FE D0	**TLVBITFETCH**
FE B6	**DBDELBYKEY**	FE D1	**TLVBITSTORE**
FE C0	**MODDELETE**	FE D2	**TLVINIT**
FE C1	**MODEXECUTE**	FE D3	**TLVFETCHRAW**
FE C2	**MODINIT**	FE D4	**TLVFETCHVALUE**
FE C3	**MODCARDEXECUTE**	FE D5	**TLVFETCHTAG**
FE C4	**MODAPPEND**	FE D6	**TLVFETCHLENGTH**
FE C6	**MODCHANGED**	FE D7	**TLVFORMAT**
FE C7	**MODREGISTER**	FE D8	**TLVTAG**

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Annex B
(normative)

Exceptions and I/O Return Codes

B.1 General

This section includes all codes used as arguments to the standard exception handling token **THROW** and all I/O related status codes. Status codes are normally used as returned ior (I/O result) values, but may be used as throw codes.

ANS Forth specifies allowable THROW codes as follows: Values {-255.. .- 1 } shall be used only as assigned by ANS Forth (Table 9.2 in ANSI X3.215). The values {-4095...-256} shall be used only as assigned by a system. Positive values are available to programs.

B.2 Exceptions and IOR codes

Table B.1 below shows the ANS Forth codes that may be used in OTA kernels. Table B.2 gives the OTA-specific throw codes and Table B.3 gives the OTA-specific I/O status codes assigned in this specification. Values are given in both hex and decimal; the hex notation emphasises the OTA list organisation by device type and error category. For OTA-specific codes, the least significant byte gives the error code within the category and the next significant byte gives the category.

Table B.1 — ANS Forth THROW codes in OTA kernels

Decimal	Hex	Description
-53	FFFFFFCB	Exception stack overflow.
-24	FFFFFFE8	Invalid numeric argument.
-23	FFFFFFE9	Address alignment exception.
-21	FFFFFFEB	Unsupported operation.
-17	FFFFFFEF	Pictured numeric string overflow.
-12	FFFFFFF4	Argument type mismatch.
-11	FFFFFFF5	Result out of range.
-10	FFFFFFF6	Division by zero.
-9	FFFFFFF7	Invalid memory address.
-7	FFFFFFF9	Do loops nested too deeply during execution.
-6	FFFFFFFA	Return stack underflow.
-5	FFFFFFFB	Return stack overflow.
-4	FFFFFFFC	Data stack underflow.
-3	FFFFFFFD	Data stack overflow.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Table B.2 — OTA `THROW` codes

Decimal	Hex	Description
-32763	FFFF8005	Unsupported device.
-4095	FFFFF001	Invalid record number. Record number outside the range defined for the current structure (zero to AVAIL -1).
-4094	FFFFF002	Invalid function. Function inappropriate for this kind of database (e.g. DBADDBYKEY on a non-ordered structure).
-4093	FFFFF003	Database creation error. An NV structure could not be initialised (e.g. file marked read-only.).
-4092	FFFFF004	Database access error. An NV structure could not be accessed (e.g. error in opening the file).
-4091	FFFFF005	Database close error. An NV structure could not be closed.
-4090	FFFFF006	Database seek error. A selected record could not be found in an NV structure.
-4089	FFFFF007	Database read error. An NV structure could not be read.
-4088	FFFFF008	Database write error. An NV structure could not be written.
-4087	FFFFF009	No database selected. Attempt to use a database function prior to first use of DBMAKECURRENT.
-3839	FFFFF 101	Undefined TLV. Cannot find TLV tag name.
-3838	FFFFF 102	TLV too large. TLV value field is longer than 252 bytes.
-3837	FFFFF 103	Duplicate TLV. A module contains a TLV definition that already exists.
-3583	FFFFF201	Cannot load module.
-3582	FFFFF202	Cannot add module.
-3581	FFFFF203	Cannot open repository.
-3580	FFFFF204	Error creating repository.
-3579	FFFFF205	Error reading repository.
-3578	FFFFF206	Error closing repository.
-3577	FFFFF207	Invalid module length.
-3576	FFFFF208	Deletion of module not allowed.
-3575	FFFFF209	Bad card module.
-3574	FFFFF20A	Invalid token.
-3573	FFFFF20B	Not a valid module.
-3572	FFFFF20C	Relocation error.
-3571	FFFFF20D	Error loading module.
-3570	FFFFF20E	Invalid override.
-3569	FFFFF20F	Module not in repository.
-3567	FFFFF211	Invalid character in MID.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

-3566	FFFFF212	Invalid MID string format.
-3565	FFFFF213	Cannot execute module.
-3564	FFFFF214	Module in use.
-3327	FFFFF301	Invalid socket.
-3071	FFFFF401	Out of memory. Request to allocate memory cannot be satisfied.
-3067	FFFFF405	Attempt to write outside allocated memory.
-3066	FFFFF406	Frame stack error. Frame of the requested size could not be built.
-510	FFFFFE02	Language file full. Insufficient space in terminal language tables.
-509	FFFFFE03	Out of context. Attempt to use a token out of proper sequence.
-508	FFFFFE04	Feature not implemented. Reference to a feature or device not supported in this kernel.
-507	FFFFFE05	String too large. String size is greater than 65535 bytes or given by context.
-506	FFFFFE06	Digit too large. Digit not within number conversion base.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Table B.3 — OTA I/O return codes

Decimal	Hex	Description
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32766	FFFF8002	Time-out.
-32765	FFFF8003	Operation cancelled by user.
-32764	FFFF8004	Device Error.
-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-32239	FFFF8211	Outside border.
-31983	FFFF83 11	Printer Off-line.
-31982	FFFF8312	Printer out of paper.
-31981	FFFF8313	Printer has asserted error signal.
-31980	FFFF8314	Printer does not appear to be connected.
-31727	FFFF8411	No connection (for on-line commands).

-31726	FFFF8412	Invalid serial or modem parameters.
-31725	FFFF8413	No carrier.
-31724	FFFF8414	No answer.
-31723	FFFF8415	Busy.
-31722	FFFF8416	No dial tone.
-31721	FFFF8417	Modem already connected.
-31720	FFFF841 8	Connection in progress.
-31471	FFFF8511	Mute card (no answer).
-31470	FFFF8512	No card in reader.
-31469	FFFF8513	Transmission error.
-31468	FFFF8514	Card buffer overflow.
-31467	FFFF8515	Protocol error.
-31466	FFFF8516	Response has no status bytes.
-31465	FFFF8517	Invalid buffer.
-31464	FFFF8518	Other card error.
-31463	FFFF85 19	Card partially in reader.
-31215	FFFF8611	Transmission error between reader and magstripe.
-31214	FFFF8612	Output buffer overflow.
-31213	FFFF8613	Write operation failed.
-30455	FFFF8909	Vending terminated.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Annex C (normative) Device Control

C.1 General

This section provides reference information for generic device control in OTA, as well as descriptions of some specific devices commonly attached to terminals. Inclusion in this table does not imply that a device is required by a terminal, nor does exclusion imply that a device may not be attached to a terminal. However, if a device described in this section is attached, its control should follow the description herein, as far as the particular hardware configuration permits.

C.2 Device References and Return Codes

Each device, including those whose lower-level operation is hidden by the virtual machine behind device-specific functions, shall be assigned a device number and an 8-bit device type used to categorise result codes. To maintain token density, the commonly used device numbers are kept in a range that can be represented in 4 bits and loaded with a single byte literal token. It is this device number which is used as the dev parameter when specified for an I/O related token.

The standard OTA device number assignments and their mapping onto device types and instances is shown in Table C.1.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Table C.1 — Device code assignments

Device Number	Description	Device Type
-1	Debug device	0
0	Primary keyboard	1
1	Primary display	2
2	Printer 1	3
3	Printer 2	3
4	Modem	4
5	ICC card reader 1	5
6	ICC card reader 2	5
7	Magnetic stripe device	6
8	Secondary keyboard	1
9	Secondary display	2
10	Secondary serial port	4
11	First parallel port	7
12	Second parallel port	7
13	Power management device	8
14	Vending machine device	9

15-31	RFU	-
32-47	Serial Ports	4
48-63	Modems	4
64-79	Printers	3
80-	RFU	-

General I/O control is provided by functions taking one of these device numbers as an input parameter. These functions and corresponding tokens are described in Section 5.18. Lists of valid ior codes for these tokens to return are better sorted by device type than by specific token, and the following sections each contain a specific list of ior codes for a particular device type. Table C.2 below shows a cross-reference from token name to valid device type and number for those tokens that take a device number as input An “X” indicates that this token may be used with this device number, provided the terminal supports the given device. Note the implication that most, if not all, DEVtokens should only appear in TRS programs or libraries, not applications, since an application should not rely on the implementation of a particular device. SETOP does not check device type as that is assumed to be the responsibility of an ensuing DEVtoken. The column labelled “ior?” in Table C.2 indicates whether that token returns an ior. Although tokens that return an ior may not also THROW the same return code, all return codes may be used as THROW codes by higher-level procedures.

Table C.2 — Token — device number cross reference

	ior?	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		db	kb	dis	prn	prn	mo	icc	icc	mg	kb	dis	ser	par	par	pw	ven	rfu
DEVOPEN	yes	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DEVCLOSE	yes	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DEVEKEY	n o	X	X				X				X		X					
DEVEKEYQ	n o	X	X				X				X		X					
DEVEMIT	n o	X		X	X	X	X					X	X					
DEVREAD	yes		X				X	X	X	X	X		X					
DEVTIMEDREAD	yes		X				X	X	X	X	X		X					
DEVWRITE	yes	X		X	X	X	X	X	X	X		X	X					
DEVSTATUS	yes	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DEVIOCTL	yes	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DEVOUTPUT	yes	X			X			X		X			X			X		X
DEVATXY	n o	X		X								X						
DEVCONNECT	yes						X						X					
DEVHANGUP	yes						X						X					
DEVBREAK	yes						X						X					

C.3 Debug Device

A “debug device” is the generic name for a programmer's interactive interface to a terminal under development. The debug device is typically implemented using a host computer's key-board and display, communicating with the terminal through an interactive development link. Table C.3 gives the ior codes appropriate for the debug device.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Table C.3 — Debug device I/O return codes

Status Decimal	StatusHex	Description
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32766	FFFF8002	Time-out.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.
-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.

C.4 Keyboard Handling

The virtual machine returns codes for keystrokes as a set of standard values consistent across all terminals. Each returned keystroke shall be a 16-bit extended character (echar) value, where the more significant byte is an extension code, and the less significant byte is a key value. The key values comply with ISO/IEC 646-1983 for values in the range 20H - 7EH. The standard keyboard encodings are given in Table C.4 (all key values in hexadecimal). A VM implementation is not mandated to return any of these keys, but shall use the values listed for any that it does return.

Table C.4 — Standard key mappings

Key character / function	Extension	Key Value
ASCII printing characters ‘ ’ (space) through ‘~’ (tilde)	00	20 .. 7E
BACKSPACE (destructive backspace)	00	08
ENTER (complete and process entry) (Green key)	00	0D
CANCEL (clear to beginning of entry) (Red key)	00	18
CLEAR (clear entry) (Yellow key)	00	7F
Function Key 1 .. Function Key 10	F0	3B .. 44
Function Key 11 .. Function Key 12	F0	85 .. 86
HOME	F0	47
END	F0	4F
Cursor Up	F0	48
Cursor Down	F0	50
Cursor Left	F0	4B
Cursor Right	F0	4D

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Page Up	F0	49
Page Down	F0	51
INSERT	F0	52
DELETE	F0	53
00 (double zero key)	F0	54

Control of the keyboard devices is provided by the I/O control functions implemented through the generic DEVIOCTL token International Standarded in Section 8.18, where dev is the key-board device code. Table C.5 specifies the DEVIOCTL arguments for the keyboard device. Table C.6 gives the ior codes appropriate for keyboards.

Table C.5 — DEVIOCTL parameters for keyboard device

Function code *(fn)*	**Meaning**
04	Set time-out. *Addr* points to a cell containing the time in milliseconds before **DEVTIMEDREAD** returns. *Num* is set to to 1. The time-out value only needs to be set once and will be retained by the virtual machine until it is explicitly changed or the device is rebooted.

Table C.6 — Keyboard device I/O return codes

Status Decimal	**Status Hex**	**Description**
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32766	FFFF8002	Time-out.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.
-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

C.5 Display and Printer Output

Devices that support text output (display and printer devices) shall provide interpretation of certain control characters appearing as output characters either individually or in output strings. The codes and the effects they shall produce are listed in Table C.7; codes are specified in hexadecimal. Codes in the range 0-1F$_H$ not appearing in this table are implementation-specific.

Table C.7 — Control Code Interpretation

Code	Effect
07	Emit an audible tone (if possible).
08	Destructive backspace (display only). Ior -32239 (Outside border) shall be returned when this control character is written to the display any time the cursor is positioned at the beginning of a line.
0A	Carriage return and line feed. When the cursor is positioned on the last line of the display, the terminal shall behave in one of the following ways: — the cursor does not move and ior -32239 (Outside border) is returned. — the cursor is positioned at the beginning of the first line. — all lines are scrolled one line up and the cursor is positioned at the beginning of the last line
0C	Form feed. Effect a physical or visual break on printer devices; clear the screen on display devices.
0D	Same as code 0A.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Regarding out of bounds behaviour on the display and printer device, the terminal shall not inadvertently write out of bounds or hide away something the application programmer expected to be displayed or printed. Therefore, the following behaviours are correct when a character (that is not a control character listed in Table C.7) is written to the display or printer device when the cursor is positioned at the end of a line:

— do nothing and return ior -32239 (Outside border).

— scroll the line one character to the left and add the character at the end of the line (display only).

— write the character at the beginning of the next line. When the cursor is positioned at the last line of the display, the following four implementations are correct:

  — do nothing and return ior -32239 (Outside border).

  — scroll the line one character to the left and add the character at the end of the line.

  — write the character at the beginning of the first line.

  — scroll all lines one line up and write the character at the beginning of the last line.

Control of the display and printer devices is provided by the I/O control functions implemented through the generic **`DEVIOCTL`** token International Standarded in Section 5.18, where *dev* is the display or printer device code. Table C.8 specifies the **`DEVIOCTL`** arguments for the display device, while Table C.9 specifies the **`DEVIOCTL`** arguments for the printer device. Display ior codes are given in Table C.10, and printer ior codes are given in Table C.11.

Table C.8 — DEVIOCTL parameters for display device

Function code *(fn)*	**Meaning**
01	Specify cursor behaviour. *Num* is set to 1. *Addr* points to a cell containing a bitmap coded as follows (bit 0 is the LSB of Byte 1) : bit 0 : 0=off, 1=on bit 1 : 0=steady, 1=flash bit 2 : 0=block, 1=underscore
02	Set backlight on/off. *Num* is set to 1. *Addr* points to a cell coded as follows: 1 cell = on/off (0=off,1=on)
03	Specify display mode. *Num* is set to 1. *Addr* points to a cell containing a bitmap coded as follows (bit 0 is the LSB of Byte 1): bit 0 : 0=blinking off, 1=blinking on bit 1 : 0=reverse video off, 1=reverse video on bit 2 : 0=underline off, 1=underline on The display mode defined by this function is valid for all characters written to the display after this **`DEVIOCTL`** function has been executed. Other characters already displayed on the display remain unchanged.
05	Return display characteristics. *Num* is set to 2. On return, *addr* points to two cells to be interpreted as follows : 1 cell = width of the display (number of characters) 1 cell = height of the display (number of rows)
06	Specify frequency of beep. *Num* is set to 1. *Addr* points to a cell containing the frequency in Hertz of the beep when control character 07 is written to the display device.
07	Specify duration of beep. *Num* is set to 1. *Addr* points to a cell containing the time in milliseconds for the duration of the beep when control character 07 is written to the display device.
08	Switch between graphic and alphanumeric display mode. *Num* is set to 1. *Addr* points to a cell coded as follows: 0 : switch to graphic display mode 1 : switch to alphanumeric display mode By default the display is in alphanumeric display mode.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Table C.9 — DEVIOCTL parameters for printer device

Function code *(fn)*	Meaning
01	Initialise serial port. *Num* is set to 5. *Addr* points to 5 cells containing the following items : 1 cell = input baud rate 1 cell = output baud rate 1 cell = parity (0=none, 1=odd, 2=even) 1 cell = number of data bits (7 or 8) 1 cell = number of stop bits (1 or 2)
02	Print whatever is already in the buffer. *Num* is set to 0. *Addr* is not used by this function and may be any arbitrary non zero value.

Table C.10 — Display device I/O return codes

Status Decimal	Status Hex	Description
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.
-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-32239	FFFF8211	Outside border.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

Table C.11 — Printer device I/O return codes

Status Decimal	Status Hex	Description
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32766	FFFF8002	Time-out.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.

-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-31983	FFFF8311	Printer off-line.
-31982	FFFF8312	Printer out of paper.
-31981	FFFF8313	Printer has asserted error signal.
-31980	FFFF8314	Printer does not appear to be connected.
-32239	FFFF8211	Outside border.

C.6 Serial Port Management

Control of the serial devices is provided by I/O control functions implemented through the generic **DEVIOCTL** token International Standarded in Section 8.18, where *dev* is the serial device code. Other stack arguments are International Standarded in Table C.12 below. Serial port ior codes are given in Table C.13.

Table C.12 — DEVIOCTL parameters for serial port device

Function code *(fn)*	**Meaning**
01	Initialise serial port. *Addr* contains the following items: 1 cell = input baud rate 1 cell = output baud rate 1 cell = parity (0=none, 1= odd, 2=even) 1 cell = number of data bits (7 or 8) 1 cell = number of stop bits (1 or 2)
04	Set port time-out. *Addr* points to a cell containing the time in milliseconds before **DEVTIMEDREAD** returns. *Num* is set to 1. This value only needs to be set once and will be retained by the virtual machine until it is explicitly changed or the device is rebooted.
05	Return flags indicating line status. *Num* is set to 1. On input, *addr* points to a cell containing the bitmap that indicates which statuses are requested to be returned. A bit set to 1 indicates that the corresponding function has to be executed. On return, *addr* points to a cell containing the result to be interpreted as follows : bit 0 = true if a character is waiting to be read bit 1 = true if a character is waiting to be sent Note that a given bit position is only valid if on input the corresponding bit has been set to 1. If one of the functions requested is not implemented by the underlying virtual machine, -21 (Unsupported operation) shall be thrown and the complete result is invalid.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved

Table C.13 — Serial port device I/O return codes

Status Decimal	Status Hex	Description
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32766	FFFF8002	Time-out.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.
-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-31726	FFFF8412	Invalid serial parameters.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.

C.7 Modem Handling

Besides with the special purpose tokens **DEVCONNECT, DEVHANGUP** and **DEVBREAK,** the modem device is also controlled with I/O control functions implemented through the generic **DEVIOCTL** token International Standarded in Section 8.18, where *dev* is the modem device code. Other stack arguments are International Standarded in Table C.14 below. The modem ior codes are given in Table C.15.

Table C.14 — DEVIOCTL parameters for modem device

Function code *(fn)*	Meaning
01	Initialise serial port. *Addr* contains the following items: 1 cell = input baud rate 1 cell = output baud rate 1 cell = parity (0=none, 1= odd, 2=even) 1 cell = number of data bits (7 or 8) 1 cell = number of stop bits (1 or 2)
04	Set port time-out. *Addr* points to a cell containing the time in milliseconds before **DEVTIMEDREAD** returns. *Num* is set to 1. This value only needs to be set once and will be retained by the virtual machine until it is explicitly changed or the device is rebooted.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

05	Return flags indicating line status. *Num* is set to 1. On input, *addr* points to a cell containing the bitmap that indicates which statuses are requested to be returned. A bit set to 1 indicates that the corresponding function has to be executed. On return, *addr* points to a cell containing the result to be interpreted as follows : bit 0 = true if a character is waiting to be read bit 1 = true if a character is waiting to be sent Note that a given bit position is only valid if on input the corresponding bit has been set to 1. If one of the functions requested is not implemented by the underlying virtual machine, -21 (Unsupported operation) shall be thrown and the complete result is invalid.
06	Initialise modem. *Addr* points to a counted string containing the modem control settings. *Num* is not used by this function and may be any arbitrary value.
07	Ask the kernel if the modem received a request for incoming call. If it did, send the counted string pointed by *addr,* wait for the connection to be established and return with an ior 0 (Success). Otherwise, return immediately with an ior -31727 (No connection). *Num* is not used by this function and may be any arbitrary value.

Table C.15 — Modem device I/O return codes

Status Decimal	Status Hex	Description
0	0	Successful operation.
-32767	FFFF8001	Device valid but currently not responding.
-32766	FFFF8002	Time-out.
-32765	FFFF8003	Operation cancelled by user.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.
-32762	FFFF8006	Device shall be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device shall be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-31727	FFFF8411	No connection (for on-line commands).
-31726	FFFF8412	Invalid serial or modem parameters.
-31725	FFFF8413	No carrier.
-31724	FFFF8414	No answer.
-31723	FFFF8415	Busy.

IECNORM.COM : Click to view the full PDF of ISO/IEC 20060:2010

© ISO/IEC 2010 – All rights reserved