INTERNATIONAL STANDARD

ISO/IEC 20060

First edition
2001-11-01

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

Technologies de l'information — Spécifications pour architecture des terminaux ouverte (OTA) — Spécifications pour machine virtuelle

Reference number
ISO/IEC 20060:2001(E)

© ISO/IEC 2001

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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.

© ISO/IEC 2001

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.ch
Web www.iso.ch

Printed in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2001 – All rights reserved

Not for Resale

Table of Contents

Foreword ix
Introduction x
Purpose x
Subject x
Audience x
How This Standard is Organised xi
Aids in Using This Standard xi
Related Publications xi
References xii

1 Scope **1 –1**

2 Conventions **2–3**
2.1 Abbreviations 2–3
2.2 Glossary 2–4
2.3 Data Types 2–7
2.4 Stack Notation 2–8
2.5 Flags 2–8

3 OTA Virtual Machine **3–9**
3.1 Introduction 3–9
3.2 Virtual Machine CPU 3–10
3.2.1 Registers 3–10
3.2.2 Virtual Machine Size and Cells 3–11
3.2.3 Memory 3–11
3.2.4 Stacks 3–11
3.2.5 Frame Mechanism and Usage 3–12
3.2.6 Extensible Memory 3–14
3.2.7 User Variables 3–14
3.3 Virtual Machine Execution Features 3–15
3.4 Arithmetic 3–15

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3.5 Exception Handling 3–16
3.6 Resources 3–17
3.7 Programs and Tokens 3–18

4 System Services 4–19

4.1 Time Handling 4–19
4.2 Devices and I/O Services 4–20
4.3 Database Services 4–21
4.3.1 The Database Parameter Block 4–22
4.3.2 Database Instantiation 4–24
4.3.3 Database Exception Handling 4–25
4.4 Language and Message Handling 4–25
4.5 TLV Services 4–26
4.5.1 Basic Principles 4–26
4.5.2 TLV Definitions 4–27
4.5.3 TLV References 4–28
4.6 Hot Card List Management 4–28
4.7 Cryptographic Services 4–29
4.7.1 Modulo Multiplication 4–30
4.7.2 Secure Hash Algorithm (SHA-1) 4–30
4.7.3 Modulo Exponentiation 4–30
4.7.4 Long Shift 4–30
4.7.5 Long Subtract 4–31
4.7.6 Incremental Secure Hash Algorithm (SHA-1) 4–31
4.7.7 Cyclic Redundancy Check (CRC) 4–31
4.7.8 DES Key Schedule 4–32
4.7.9 DES encryption/decryption 4–32
4.8 Vectored Execution Sockets 4–32
4.8.1 CSS Functions 4–33
4.8.2 Socket Security 4–33
4.8.3 Socket Organisation 4–33
4.9 Module Handling Services 4–33
*4.9.1 Module Loading by **MODEXECUTE*** 4–34
4.9.2 Module Loading Procedure 4–37
*4.9.3 Module Loading by **MODCARDEXECUTE*** 4–39

5 Token Set Definition 5–43

5.1 Overview 5–43
5.2 Conventions 5–43
5.2.1 Number Formats 5–43
5.2.2 Token Descriptions 5–43
5.2.3 Branch and Code Offsets 5–44
5.2.4 Addresses 5–44
5.3 Data Typing 5–44

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.4 Token Compression 5–44
5.4.1 Optimised Data Access 5–45
5.4.2 Special Procedure Calls 5–45
5.4.3 Quoting 5–45
5.5 Prefix Tokens 5–46
5.6 Stack Manipulation Tokens 5–47
5.7 Data Access Tokens 5–49
5.8 Literal Tokens 5–51
5.9 Address Generation Tokens 5–52
5.10 Arithmetic Tokens 5–53
5.11 Relational Tokens 5–55
5.12 String Tokens 5–57
5.13 Frame Tokens 5–60
5.14 Extensible Memory Tokens 5–62
5.15 Flow of Control Tokens 5–63
5.15.1 Branch Tokens 5–63
5.15.2 Call Tokens 5–64
5.15.3 Loop Tokens 5–65
5.15.4 Hybrid Tokens 5–66
5.15.5 Quoting Tokens 5–66
5.16 Exception Tokens 5–67
5.17 Date, Time, and Timing Tokens 5–67
5.18 Generic Device I/O Tokens 5–68
5.19 Formatted I/O Tokens 5–71
5.20 Integrated Circuit Card Tokens 5–72
5.21 Magnetic Stripe Tokens 5–73
5.22 Socket Tokens 5–75
5.23 Database Services Tokens 5–76
5.24 Language and Message Tokens 5–80
5.25 TLV Tokens 5–81
5.25.1 TLV Buffer Access 5–82
5.25.2 TLV Processing 5–83
5.25.3 TLV Sequence Access 5–85
5.26 Hot Card List Tokens 5–85
5.27 Cryptographic Algorithm Token 5–86
5.28 Module Management Tokens 5–86
5.29 Operating System Interface Tokens 5–88
5.30 Miscellaneous Tokens 5–88

6 Module Delivery Format 6–91

6.1 Module ID Format 6–92
6.2 Socket List 6–92

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.3 Relocation Section 6–92
6.4 Module Import List 6–94
6.5 Module Export List 6–94
6.6 Module Procedure List 6–94

Appendix A: OTA Token Lists A–97

A.1 Numeric List of Tokens A–97
A.2 Alphabetic List of Tokens A–100

Appendix B: Exceptions and I/O Return Codes B–103

B.1 Exceptions and IOR codes B–103

Appendix C: Device Control C–107

C.1 Device References and Return Codes C–107
C.2 Debug Device C–109
C.3 Keyboard Handling C–110
C.4 Display and Printer Output C–111
C.5 Serial Port Management C–115
C.6 Modem Handling C–116
C.7 ICC Card Handling C–117
C.8 Magnetic Stripe Handling C–119
C.9 Power Management C–120
C.10 Vending Machine Control C–121

Appendix D: Operating System Calls D–123

Appendix E: Rules for Using a Data Object List (DOL) E–125

Appendix F: System Overview F–127

How This Appendix is Organised F-127
F.1 Introduction F–128
F.2 System Components F–131
F.3 OTA System Features F–135
F.4 Program Security and Integrity F–142
F.5 OTA Software Development Tools F–145
F.6 Summary of the Open Terminal Architecture F–151

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
© ISO/IEC 2001 – All rights reserved
Not for Resale

Index 153

List of Figures

1. Virtual Machine architecture 3-10
2. Frame management example 3-13
3. Database memory access 4-22
4. Module execution procedure 4-36
5. Module loading procedure 4-38
6. ICC module execution procedure 4-40
7. OTA development environment F-131
8. Tokens in the OTA concept F-135
9. The OTA Virtual Machine F-136
10. Kernel development tools F-146
11. Application development tools F-147

List of Tables

1. Data type designations used in OTA 2–7
2. Virtual Machine registers 3–10
3. Data that the VM may hold on the return stack 3–12
4. User variables in the Virtual Machine 3–14
5. Initial condition of the VM on entry to the TRS 3–15
6. Optional general exceptions from ANS Forth 3–17
7. Virtual Machine resources 3–17
8. DPB Structure 4–22
9. Messages, by number and orig in 4–25
10. Message table format 4–26
11. Cryptographic algorithm codes 4–29
12. Result codes from **COMPARE** 5–58
13. ISO parameter track selection codes 5–74
14. Module delivery format 6–91
15. Socket list in Module Delivery Format 6–92
16. Relocation specification 6–93
17. Module import list format 6–94
18. Module export list format 6–94
19. Module procedure list format 6–95
20. ANS Forth **THROW** codes in OTA kernels B–103
21. OTA **THROW** codes B–104
22. OTA I/O return codes B–105
23. Device code assignments C–108
24. Token — device number cross reference C–109
25. Debug device I/O return codes C–109
26. Standard key mappings C–110
27. DEVIOCTL parameters for keyboard device C–111
28. Keyboard device I/O return codes C–111
30. DEVIOCTL parameters for display device C–112
29. Control Code Interpretation C–112
31. DEVIOCTL parameters for printer device C–113
33. Printer device I/O return codes C–114
32. Display device I/O return codes C–114
34. DEVIOCTL parameters for serial port device C–115
35. Serial port device I/O return codes C–115
36. DEVIOCTL parameters for modem device C–116
37. Modem device I/O return codes C–117

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

38. DEVIOCTL parameters for ICC card reader device C–118
39. ICC card reader device I/O return codes C–118
40. DEVIOCTL parameters for magnetic card reader device C–119
41. Magnetic card reader I/O return codes C–119
42. DEVIOCTL parameters for power management device C–120
43. Power management device I/O return codes C–120
44. DEVIOCTL parameters for vending machine device C–121
45. Vending machine device I/O return codes C–121
46. OSCALL functions D–123
47. Virtual Machine registers F–136

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. 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 International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 20060 was prepared by Europay International and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

Purpose

The Europay Open Terminal Architecture (OTA) consists of technology designed to facilitate implementation of Integrated Circuit Cards (ICCs) and associated terminals.

The purpose of this document is to provide a specification for a standard kernel to be provided in all OTA terminals.

Subject

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.

Additional volumes in this series describe Forth and C language bindings and compiler requirements.

Audience

This document is intended for anyone desiring to evaluate OTA technology, develop OTA kernels, or develop payment programs or libraries designed to run on OTA kernels. General knowledge of computers and programming is assumed.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2001 – All rights reserved

Not for Resale

How This Standard is Organised

This document is divided into the following chapters:

Chapter 1 *Management Summary* provides a summary of the key points that are developed in this book.

Chapter 2 *Conventions* describes notational and syntactic conventions used in this specification, as well as OTA data types supported and other general technical issues applying to subsequent chapters.

Chapter 3 *OTA Virtual Machine* describes the "Virtual Machine" architecture upon which OTA is based.

Chapter 4 *System Services* describes the various services provided by an OTA kernel to client programs.

Chapter 5 *Token Set Definition* provides a detailed specification of the OTA token set (the machine language of the "Virtual Machine").

Chapter 6 *Module Delivery Format* describes the delivery package for tokenised external modules provided to OTA kernels.

Aids in Using This Standard

This document contains the following aids for using the information it presents:

- A list of all of the tables present in this standard, found on page vi.
- A list of abbreviations referred to in this standard, found on page 2–3.
- A glossary of terms used in this standard, found on page 2–4.
- Numeric and alphabetic lists of OTA tokens, with page numbers, in Appendix A.
- A summary of exception codes and I/O result codes in Appendix B.
- A summary of current devices supported, with device numbers and control codes, in Appendix C.
- A list of operating system functions in Appendix D.
- Rules for TLV ‚Data Object List' handling in Appendix E.
- Open Terminal Architecture System Overview in Appendix F.
- An index of topics covered in this standard, found at the end of the document.

Related Publications

The following publications contain material directly related to the content of this standard. Available from original PAS submitter (see above).

- EMV2000, *Integrated Circuit Card Specification for Payment Systems, Book 1 - Applications Independent ICC to Termiual Interface Requirements.* Version 4.0, December 2000.[1]
- EMV2000, *Integrated Circuit Card Specification for Payment Systems, Book 2 - Security*

1. The EMV2000 documents may be obtained free of charge from EMVCo at http://www.emvco.com.

Copyright International Organization for Standardization
Provided by IHS under license with ISO © 2001 – All rights reserved
No reproduction or networking permitted without license from IHS
Not for Resale

and Key Management. Version 4.0, December 2000.

- EMV2000, *Integrated Circuit Card Specification for Payment Systems, Book 3 - Application Specification.* Version 4.0, December 2000.
- EMV2000, *Integrated Circuit Card Specification for Payment Systems, Book 4 - Cardholder, Attendant, and Acquirer Interface Requirements.* Version 4.0, December 2000.
- *Open Terminal Architecture (OTA) Specification Volume 2: Forth Language Programming Interface.* Version 3.0 Draft 2, July 3, 1998.
- *Open Terminal Architecture (OTA) Specification, Volume 3: C Language Programming Interface.* Version 1.3 – July 17, 1997.
- *Open Terminal Architecture (OTA) Specification, Volume 4: Mixed Language Programming*. Version 1.1 – July 1, 1997.
- *OTA Terminal Kernel Test Program (TKTP) Reference Manual. Version 4.5, Oct. 12, 1999.*

References

The following references may be of use to the reader of this document:

ANSI X9.30-2:1997	Public key cryptography using irreversible algorithms for the financial services industry — Part 2: The Secure Hash Algorithm (SHA)
ANSI X9.31-1998	Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)
FIPS PUB 180-1:1994	Secure Hash Standard
ISO 639:1988	Codes for the representation of names and languages
ISO 3166-1:1997	Codes for the representation of names of countries and their subdivisions — Part 1: Country Codes
ISO 4217:2001	Codes for the representation of currencies and funds.
ISO/IEC 8825:1998	Information technology — Open systems interconnection — Specification of basic encoding rules for abstract syntax notation one (ASN.1).
ISO/IEC 7813:2001 (E)	Identification cards — Financial transaction cards.
ISO/IEC 7816-4:1995 (E)	Information technology — Identification cards — Integrated circuit(s) cards with contacts — Part 4: Interindustry commands for interchange
ISO/IEC 9646:1994	Information technology — Open Systems Interconnection — Conformance testing methodology and framework
ISO/IEC 15145:1997	Information technology — Programming Languages — Forth

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

1 Scope

This document is one of several documents containing the Implementation Specification of Europay's Open Terminal Architecture. Other volumes in this series specify the Forth and C language programming interfaces, as well as the EMV application library and the Terminal Kernel Test Program (see related publications).

The overall architecture of the Open Terminal Architecture is described in Appendix F, "System Architecture," 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 Section 3.

This International Standard describes 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.

This International Standard provides the specifications for the standard OTA kernel in several layers:

- definition of the Virtual Machine (VM) (Section 3);
- description of the services provided by the VM to terminal programmers (Section 4);
- specification of a set of tokens representing the native machine language of the VM (Section 5);
- specification of the format in which token modules are delivered to an OTA kernel for processing (Section 6).

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2 Conventions

This section describes technical terms and notational conventions used in this document. Additional notation specific to certain sections is described therein.

In this document the words “shall” and “must” indicate mandatory behaviour. The word “will” indicates predicted or consequential behaviour. The word “may” indicates permitted or desirable, but not mandatory, behaviour. The phrase “may not” indicates prohibited behaviour.

2.1 Abbreviations

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 consortium
H	Hexadecimal (base 16) when used as a subscript
ICC	Integrated Circuit Card
ISO	International Organisation for Standardisation
K	1024
LSB	Least Significant Bit
MDF	Module Delivery Format
MID	Module ID
OTA	Europay Open Terminal Architecture
PAN	Primary Account Number
POS	Point Of Sale (terminal)
RAM	Random Access Memory
RFU	Reserved for Future Use

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

2.2 Glossary

Aligned address	The address of a memory location at which a character or cell can be accessed. The OTA Virtual Machine requires that aligned addresses be exact multiples of 4.
ANS Forth	The Forth programming language as defined by the American National Standard X3.215, 1994. Forth language support for OTA is documented in a separate volume.
Big-endian	Describes a 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). The OTA Virtual Machine uses Big-endian byte order in token modules and card communication.
Binary	In EMV the term *binary* is used for both numbers and bit strings. When a data element is a number, this is to be interpreted as an unsigned integer.
C	The C programming language. C language support for OTA is documented in a separate volume.
Card Selected Services (CSS)	Card-resident code that provides functions supporting terminal transactions, usually service functions that are used as part of a TSS application.
Cell	The primary unit of information storage in the architecture of an OTA system. The standard size of a cell in the OTA Virtual Machine is four bytes.
Compile	Transform higher-level specifications of software and/or data into executable form. The executable form for the OTA Virtual Machine is OTA tokens.
Compressed numeric	In EMV the term *compressed numeric* specifies that a number is represented in BCD format, left justified and padded with trailing hexadecimal F's.
Counted string	A data structure consisting of one character containing the length followed by zero to 255 data characters.

-`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Data space	The logical area of the Virtual Machine that can be accessed by OTA tokens.
Data stack	A stack that may be used for passing parameters between functions. When there is no possibility of confusion, the data stack is referred to as "the stack". Contrast with *return stack.*
EMV	A consortium of Europay International SA, MasterCard International Incorporated, and Visa International Service Association.
EMV 4.0	*Integrated Circuit Card Specification for Payment Systems*, Version 4.0, December 2000, books 1..4.
Exception frame	An exception frame is the implementation-dependent set of information recording the current execution state necessary for the layered exception processing used in the VM.
Exception stack	A stack used for the nesting of exception frames. It may be, but need not be, implemented using the return stack.
Execution pointer	A value that identifies the execution semantics of a function. In ANS Forth this is referred to as an "execution token," however in this specification we have changed the term to avoid confusion with OTA tokens.
Implementation Conformance Statement	A statement made by the supplier of an implementation or system claimed to conform to a given specification, stating which capabilities have been implemented[a].
Instantiate	Register a local instance of a data structure with the Virtual Machine. At power-up in a terminal, initialised data items and kernel databases must be instantiated. Further data, databases and TLV definitions may be instantiated when a module is loaded.
Interpret	For OTA token code, at run-time identify the function associated with a token value in the code and execute it.
Kernel	The standardised set of functions mandated to be present on every terminal to implement the OTA Virtual Machine. The kernel implementation for each CPU type is optimised for that processor.
Library Module	A set of software functions in OTA token code with a published interface, providing general support for Terminal Programs and/or Applications.
LISP	A family of programming languages, developed since the late 1960's, which the American National Standard working group X3J13 standardised as Common Lisp, starting 1986.
Module	A collection of software functions and/or data compiled together and presented in token form as a package whose delivery format is specified in Section 6 of this document.
Module Repository	A non-volatile medium for storing OTA modules within a terminal.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Non-volatile	Guaranteed to be preserved across module loading or terminal power-down and rebooting.
Numeric	In EMV the term *numeric* specifies that a number is represented in BCD format, right justified and padded with leading hexadecimal zeroes. In this document we use it to specify a single-cell binary signed integer.
Return stack	A stack that may be used for program execution nesting, do-loop execution, temporary storage, and other purposes.
Stack	An area in memory containing a Last In, First Out list of items. Stacks in the OTA Virtual Machine are discussed in Section 3.2.4.
Terminal	Any POS (Point of Sale) machine, ATM (Automated Teller Machine), vending machine, etc.
Terminal Program	The overall supervisory program that a Terminal executes. The Terminal Program contains code to select among one or more applications.
Terminal Resident Services (TRS)	Includes terminal-specific functions, program loading functions, and the main Terminal Program loop defining the terminal's behaviour. A TRS is usually provided by a terminal's manufacturer.
Terminal Selected Services (TSS)	Terminal-independent code which implements service functions, also known as applications and libraries. The TRS main program loop selects and calls TSS functions as needed by a particular application.
Token	A one- or two-byte code representing a CPU-independent instruction for the OTA Virtual Machine.
Token Compiler	A compiler that produces OTA token modules.
Token Loader/ Interpreter	A software component on the Terminal that processes downloaded Tokens for execution on that terminal's CPU.
Virtual Machine	A theoretical microprocessor architecture. 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.

a. For a further discussion, see ISO/IEC 9646:1994, Information Technology — Open Systems Interconnection — Conformance Testing Methodology and Framework.

-`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

2.3 Data Types

Table 1: Data type designations used in OTA

Symbol	Data type	Size on stack
a-addr	Aligned address. Must 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 2.5 for conventional interpretations.	1 cell
fmt	TLV parameter, containing its format indicator and status. See Section 4.5 and Section 5.25 for usage.	1 cell
fn	Function code.	1 cell
ior	Result of an I/O operation. See Section 2.5 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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.4 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.

2.5 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 Appendix 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3 OTA Virtual Machine

3.1 Introduction

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 Section 3.2 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.

These services are described in detail in Section 4 of this document.

The fundamental design concepts underlying OTA technology are discussed in *Appendix F, Europay Open Terminal Architecture (OTA) System Architecture*.

OTA provides program portability across heterogeneous terminals by treating terminal programs 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3.2 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.

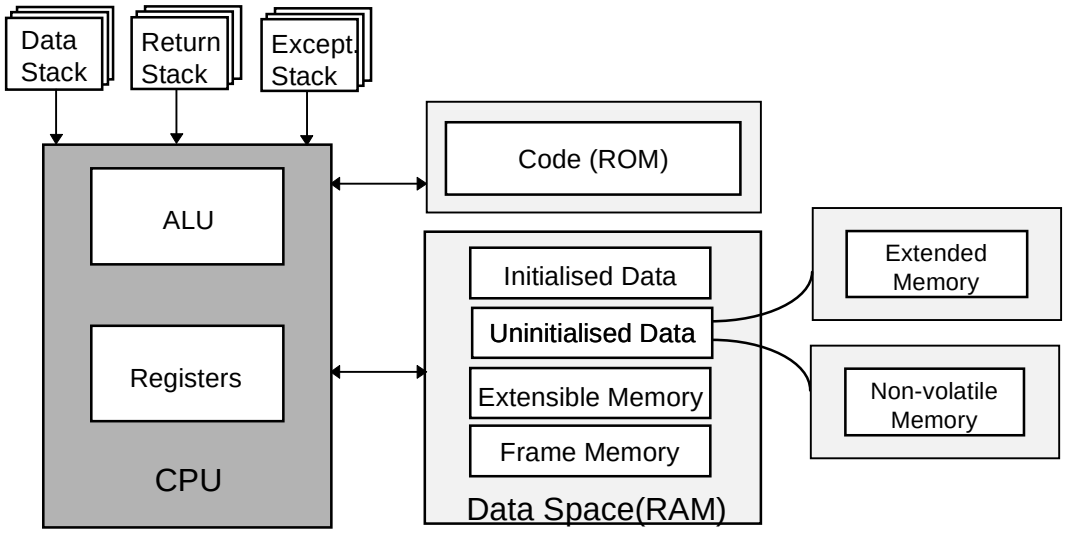


Figure 1: Virtual Machine architecture

The data, return and exception stacks are not in accessible memory space. Code memory is not available to token programs. Extended memory and Non-volatile memory are available only through tokens providing access to buffers in uninitialised data space. See Section 3.2.3.

3.2.1 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

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 5.4.3.

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.

3.2.2 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.

3.2.3 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 program compilation. This data space is initialised to zero by the Virtual Machine;
3. *frame memory*, which is managed by the frame tokens (see Section 3.2.5 and Section 5.13).
4. *extensible memory*, which contains buffers allocated dynamically during program execution.

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 hardware), 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.

3.2.4 Stacks

The *data stack* is used to contain parameters to procedures and temporary results from

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 must 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 3.5, Section 5.16
Frames	Section 3.2.5, Section 5.13
Loop Control	Section 5.15.3
Database context	Section 5.23

All programs must 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 must take appropriate action to recover the data storage from the Virtual Machine before exiting from the procedure in which it is passed.
- Programs must 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 must 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.

3.2.5 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 must conform to the rules given in Section 3.2.4.

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

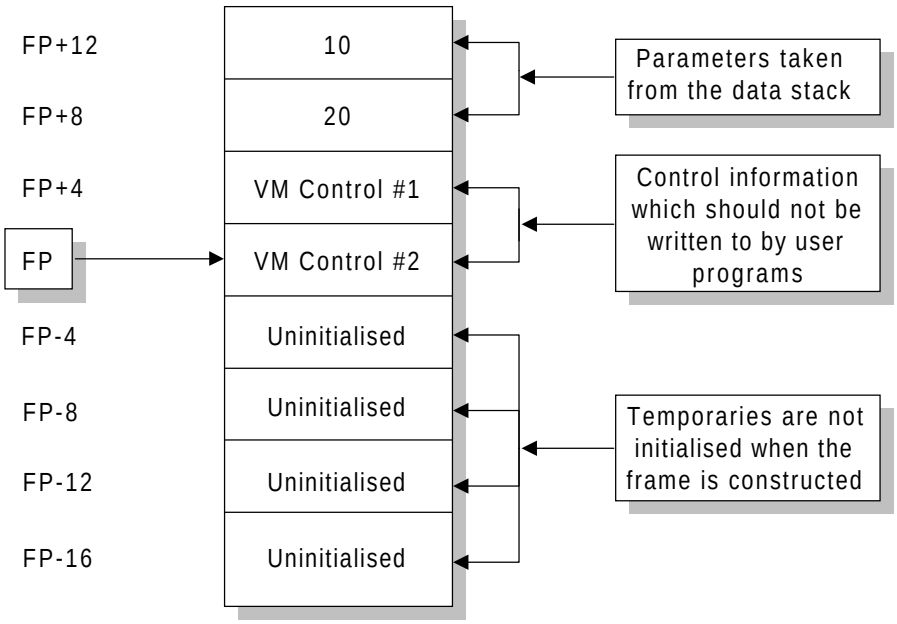


Figure 2: Frame management example

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

memory; the only requirement is that cells within the frame must be contiguous.

3.2.6 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 must 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 3.5 and Section 5.16).

3.2.7 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 must therefore be used only for the purposes described.

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 4.3.1).
4	DBRECNUM	Current record number in the current database.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

3.3 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 4.9. 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 5.22).
Current database	DBCURRENT = 0 (invalid address)
Current database record	DBRECNUM = -1 (no record selected)

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 4.8 and Section 5.28 for details of the module management services provided by the Virtual Machine. See Section 6.1 for the ID of the startup module called from the Virtual Machine.

3.4 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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

3.5 Exception Handling

OTA provides a single exception handling mechanism via the tokens **CATCH**, **THROW** and **QTHROW** (see Section 5.16). These tokens are derived from the Lisp exception handling mechanism, and appear in ANS Forth as **CATCH** and **THROW**. Exception codes are defined in Appendix 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 4 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 5, 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 documented for the individual tokens and which are listed in Table 21 of Appendix B.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.
-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.

3.6 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 5. These resources are identified in Table 7. Each VM implementation shall document the amount of each resource in this table that it will provide to application programs, and each OTA program shall document 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.

Table 7: Virtual Machine resources

Resource	Units
Extensible memory space	Bytes
Data stack	Cells
Return stack	Cells
Exception frames	Frames

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7: Virtual Machine resources *(continued)*

Resource	Units
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	--

3.7 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 5.

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 6 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.

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4 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 outside 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 program 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 6.

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.

4.1 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 documented in Section 5.17.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.2 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 Appendix 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 Appendix 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 5.18. Some of the devices like the ICC and magstripe card readers have specific tokens assigned. To be able to deal with ISO 7816-4 compatible ICC cards in a protocol independent way (T=0 or T=1), the token **CARD** must be used. The format of the input and output buffers is defined as the ISO 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 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 7813 compatible magstripe cards. The **DEVREAD**, **DEVTIMEDREAD** and **DEVWRITE** tokens should only be used for magstripe cards that are not ISO 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 Appendix 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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3 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.

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 5.23.
- 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 database 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 4.9) when replaced, e.g., in the case of a TRS module, the server must delete the module's non-volatile databases when the module is unloaded.

Figure 3 below illustrates this sharing of data space.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

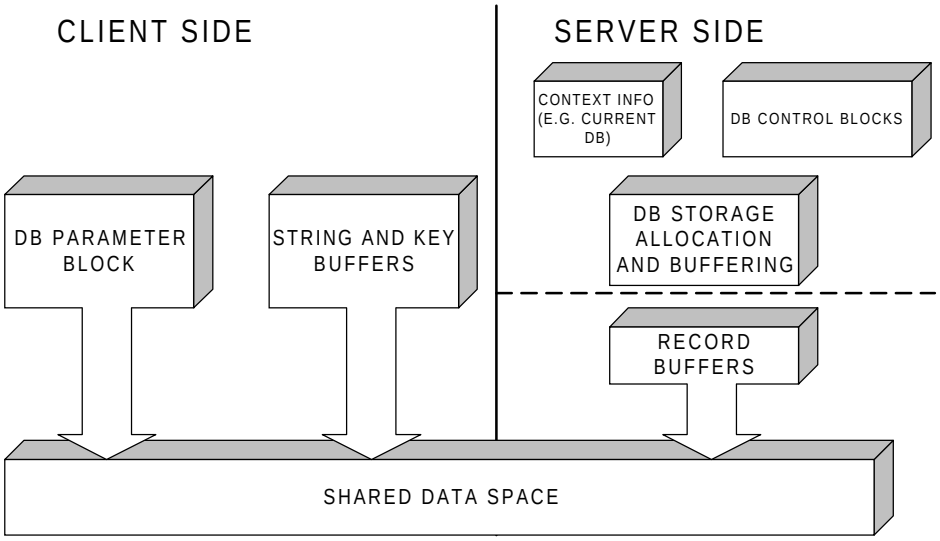


Figure 3: Database memory access

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 must 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.

4.3.1 The Database Parameter Block

The DPB for a database must 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

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

All information to specify a database must be preset in the DPB. Client software may not make any subsequent direct access to the DPB and must make no assumptions about the values directly held in the DPB after the module defining that DPB has been loaded for execution (see Section 5.23). 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 must 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 must be preset to the "origin" address in initialised data. For databases whose storage is controlled by the server, the field must 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 28-31) 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*.

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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1 "Non-volatile" database whose content must 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 supported.

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 must be preset to the size of the record in bytes.

Available For compiled databases, the compiler must set this field to one plus the last allocated record number in the database. For any other database, the compiler must set this field to zero.

Key Offset If ordering is required on the database (**DB type** bit 31=1), this field must 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 must be preset to the size in bytes of the key within the database record.

4.3.2 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3.3 Database Exception Handling

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

4.4 Language and Message Handling

Language selection, message selection and message display are provided by a set of dedicated tokens documented in Section 5.24. 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 [a]

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

a. This organisation reflects the needs given with chapter 1.2, Standard Messages, in part iii of EMV‘96, Integrated Circuit Card Terminal Specification for Payment Systems, respectively the same chapter in EMV2000, Book 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 overlapping **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 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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 8859 code for the character set used by these messages.

4.5 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).

4.5.1 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

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

4.5.2 TLV Definitions

Each TLV that will be used must 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:

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 must 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 must be initialised to zero by the compiler.

The system must also maintain a status byte for each TLV (see Section 5.25.3). 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 ini-

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

tialised 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 subroutine 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 5.25.

4.5.3 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 must 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 must 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 contents of this scratch area. If changes are to be retained, an address and length within the scratch area must 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.

4.6 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. ThePAN entries are stored in compressed numeric (cn) format, 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 iden-

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

tical up to the first F_H in the entry.

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

```
5413278000404808FFFF
5413278000404763FFFF
3625FFFFFFFFFFFFFFF
45066367FFFFFFFFFFF
```

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 documented in Section 5.26. Tokens are provided to initialise the hot card list to an empty state, add entries, delete entries, and find entries.

4.7 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 Section 5.27). 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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.7.1 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 must 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 must 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*$_3$ must be the same. The buffer *c-addr*$_4$ *len*$_3$ must not overlap with any of the input buffers.

4.7.2 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*$_2$ must not overlap the input buffer *c-addr*$_1$ *len*$_1$.

4.7.3 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 must 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 must not overlap with any of the input buffers. There must be enough room at *c-addr*$_3$ to store *len*$_2$ bytes.

4.7.4 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 signifi-

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

cant 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 must 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*.

4.7.5 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 must be the same length. The values must be in big-endian byte order.

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

($c\text{-}addr_1$ $c\text{-}addr_2$ len 5 — $c\text{-}addr_2$ len)

where the value at $c\text{-}addr_2$ *len* is subtracted from the value at $c\text{-}addr_1$ *len*, and the result is stored at $c\text{-}addr_2$ *len*.

4.7.6 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 :

($c\text{-}addr_1$ len_1 $c\text{-}addr_2$ num 6 — $c\text{-}addr_2$)

where $c\text{-}addr_1$ len_1 is the input buffer for computation and $c\text{-}addr_2$ 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.

For the initialisation step $c\text{-}addr_1$ and len_1 are not taken into account. The initialisation step initialises $c\text{-}addr_2$ 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 $c\text{-}addr_2$ must not overlap the input buffer $c\text{-}addr_1$ len_1 and is passed between the different steps.

4.7.7 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 u_1 u_2 7 — u_3)

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. u_1 is the input to the CRC algorithm, which allows chained computation over multiple blocks and also for preconditioning data for CRC-32. u_2 is the polynomial, where the bit represent-

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ing 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). u_3 is the 32 bit value of the new CRC computed over the input data.

4.7.8 DES Key Schedule

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

(c-addr$_1$ c-addr$_2$ 8 — c-addr$_2$)

where *c-addr$_1$* is the 8 byte key buffer including parity bits and *c-addr$_2$* the 128 byte key schedule output buffer.

4.7.9 DES encryption/decryption

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

(c-addr$_1$ c-addr$_2$ c-addr$_3$ num 9 — c-addr$_3$)

where *c-addr$_1$* is the 8 byte input buffer, *c-addr$_2$* the 128 byte key schedule buffer computed with **CRYPTO** function 8 and *c-addr$_3$* the 8 byte result buffer. *Num* is 1 for encryption and 2 for decryption.

4.8 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 software 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 runtime. 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.

Part of the specification for using sockets is that all procedures vectored to use a particular socket must have no stack effect (except for socket zero — see Section 5.22). 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.8.1 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 download tokens that define a new behaviour and then use **PLUGSOCKET** to store it in this same socket, overriding the default behaviour.

4.8.2 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 5.22). The procedure **PLUGGABLE** must 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 **PLUGSOCKET** 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 5.22) 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 4.9 for further details of module loading and execution.

4.8.3 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.

4.9 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:

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Tokenised code
- Initialised data
- Uninitialised data allocation
- 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 documented in Section 6. How this delivery is accomplished is beyond the scope of this document.

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 6.1. 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 document. The VM must, however, provide the following features:

- If modules are not stored in MDF as described in Section 6, then they must behave as though they were.
- The VM must protect modules within the repository from modification by any other module.
- The VM must 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.

4.9.1 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 must 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

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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 4.9.2. 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. Otherwise, the **MDF-ENTRY** field specifies the entry point of the module.

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, must 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 defined 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 must be restored to its state immediately before module execution.
- The contents of the socket list maintained by the VM must 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

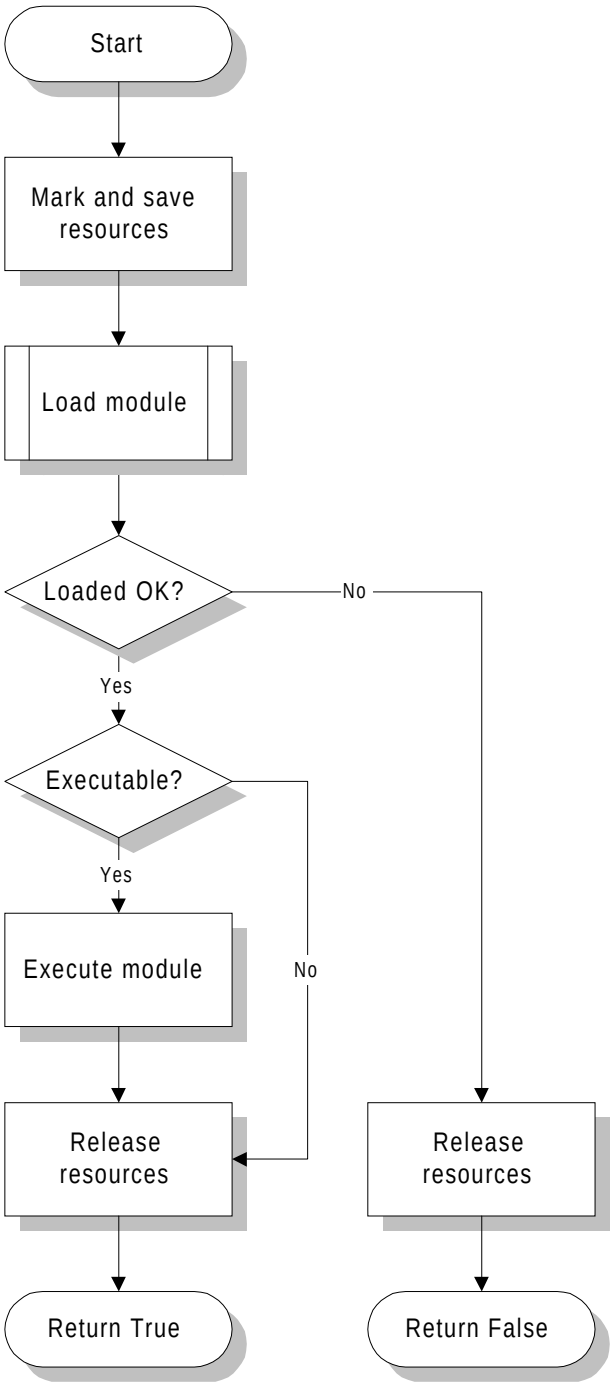


Figure 4: Module execution procedure

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.9.2 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.

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 must 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.*

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

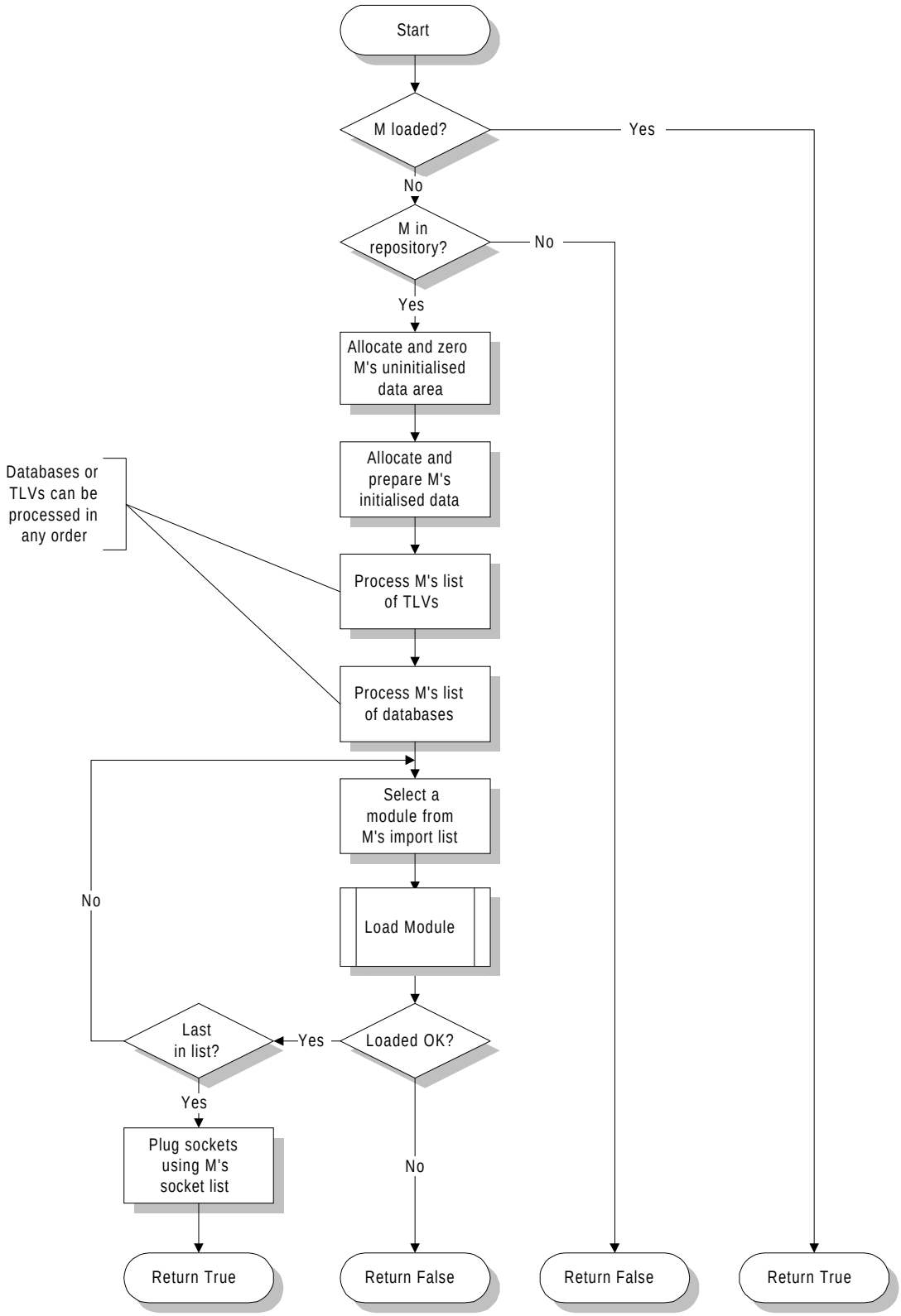


Figure 5: Module loading procedure

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 4.3.2.

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 must be ignored by the Virtual Machine. If socket zero needs to be plugged, it may be accomplished using the **PLUGSOCKET** token.

4.9.3 Module Loading by MODCARDEXECUTE

Modules that are loaded from an ICC by **MODCARDEXECUTE** must 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 must 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

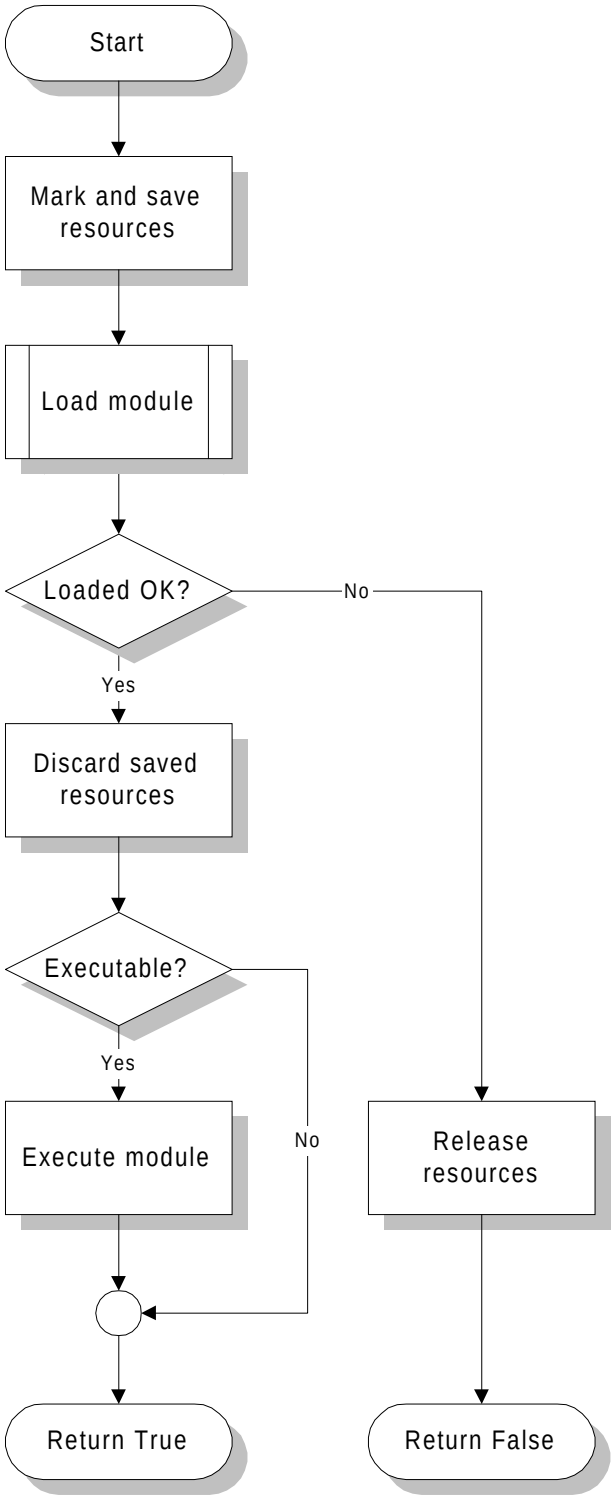


Figure 6: ICC module execution procedure

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Release Resources

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

- All volatile memory required to load the module, and any module that it required to be loaded, must 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 defined 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 must be restored to its state immediately before module execution.
- The contents of the socket list maintained by the VM must 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 must 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 5.28 and Section 6.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5 Token Set Definition

5.1 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.

5.2 Conventions

5.2.1 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.

Double number types are held on the stack with the most significant cell topmost, and programs 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.

5.2.2 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 (page 2–7), 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.

5.2.3 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*.

5.2.4 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.

5.3 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.

5.4 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.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

5.4.1 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 5.7. Similarly, 10-bit address offsets in both initialised and uninitialised data space are provided by tokens of the form **SLIT**xx, described in Section 5.9.

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.

5.4.2 Special Procedure Calls

The group of tokens named **CALL0** … **CALL39** (tokens $\mathbf{0\text{-}27}_{H}$) 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 6.6) 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.

5.4.3 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
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

5.5 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***.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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**.

5.6 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

TWOSWAP **9C**

(x_1 x_2 x_3 x_4 — x_3 x_4 x_1 x_2)

Exchange two pairs of items on the stack.

TWODROP **9D**

(x_1 x_2 —)

Discard two items from the stack.

TWODUP **9E**

(x_1 x_2 — x_1 x_2 x_1 x_2)

Duplicate a pair of items on the stack.

TWOTOR **9F**

(x_1 x_2 —) ; (R: — x_1 x_2)

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

TWORFROM **A0**

(— x_1 x_2) ; (R: x_1 x_2 —)

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

TWOOVER **A1**

(x_1 x_2 x_3 x_4 — x_1 x_2 x_3 x_4 x_1 x_2)

Copy x_1 x_2 to the top of the stack.

TWORFETCH **A2**

(— x_1 x_2) ; (R: x_1 x_2 — x_1 x_2)

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

PICK **FE 43**

(x_u..x_1 x_0 u — x_u..x_1 x_0 x_u)

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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

5.7 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 — x_1 x_2)

Fetch two items stored at *a-addr*, where x_2 is stored at *a-addr*, and x_1 at the next consecutive cell.

TWOSTORE **FE 31**

(x_1 x_2 a-addr —)

Store two items on the stack into memory with x_2 at *a-addr* and x_1 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

5.8 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.

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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.9 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*.

LITD **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.

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.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

5.10 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 $num_3|u_3$.

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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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**

(d_1 d_2|ud_1 ud_2 — d_3|ud_3)

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**

(u_1 u_2 — ud)

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

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. $u_2 = mod(ud,u_1)$, $u3 = ud/u_1$.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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*.

5.11 Relational Tokens

SETNE **BC**

(x — flag)

Return true if $x \neq 0$.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SETEQ **BA**

(x — flag)

Return true if $x = 0$.

SETGE **FE 15**

(num — flag)

Return true if $num \geq 0$

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 $num \leq 0$

CMPNE **B4**

(x_1 x_2 — flag)

Return true if $x_1 \neq x_2$.

CMPEQ **B3**

(x_1 x_2 — flag)

Return true if $x_1 = x_2$.

CMPGEU **B8**

(u_1 u_2 — flag)

Return true if $u_1 \geq u_2$.

CMPLTU **B6**

(u_1 u_2 — flag)

Return true if $u_1 < u_2$.

CMPGTU **B9**

(u_1 u_2 — flag)

Return true if $u_1 > u_2$

CMPLEU **FE 14**

(u_1 u_2 — flag)

Return true if $u_1 \leq u_2$.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

CMPGE **FE 17**

(num_1 num_2 — flag)

Return true if $num_1 \geq num_2$.

CMPLT **B5**

(num_1 num_2 — flag)

Return true if $num_1 < num_2$.

CMPGT **B7**

(num_1 num_2 — flag)

Return true if $num_1 > num_2$.

CMPLE **FE 13**

(num_1 num_2 — flag)

Return true if $num_1 \leq num_2$.

DCMPLT **FE 21**

(d_1 d_2 — flag)

Return true if $d_1 < d_2$.

WITHIN **BD**

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

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

5.12 String Tokens

MOVE **C5**

($addr_1$ $addr_2$ len —)

If *len* is greater than zero, copy *len* bytes from $addr_1$ to $addr_2$, non-destructively if the areas overlap. After the transfer the *len* bytes at $addr_2$ are the same as the *len* bytes at $addr_1$ 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.*

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

COMPARE **C7**

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

Compare two strings in memory *c-addr$_1$* and *c-addr$_2$* byte-by-byte, up to the shorter of *len$_1$* and *len$_2$* 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.

SKIP **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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

Adjust the character string at *c-addr*$_1$ by *num* characters. The resulting character string, specified by *c-addr*$_2$ len$_2$, begins at *c-addr*$_1$ plus *num* characters and is *len*$_1$ minus *num* characters long.

CNFETCH **FE 45**

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

Fetch an ASCII string to the temporary location *c-addr*$_2$ for *len*$_2$ bytes that represents the compressed number in the string at *c-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 encountered. 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 must be moved to a more permanent location immediately.

CNSTORE **FE 46**

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

Store the number represented by the ASCII string at *c-addr*$_1$ for *len*$_1$ bytes as a compressed number into the string at *c-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 in a temporary buffer. The maximum length of this

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

5.13 Frame Tokens

The following tokens provide access to the frame store.

SMAKEFRAME **E8** u_1:8 u_2:8

(x_{u1} . . . 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 3.2.4). Procedure parameters are moved from the data stack into the frame by **SMAKEFRAME** so that they can be accessed by **FRFETCH** and **FRSTORE**.

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

MAKEFRAME **FE 64** u_1:16 u_2:16

(x_{u1} . . . 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 3.2.5 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.

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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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.

FRSTORE **E5** *num:16*

(x —)

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

BYTE FRSTORE **E6 E5** *num:16*

(char —)

Store *char* at the offset *num* in the active frame.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

5.14 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 must 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 insufficient memory available.

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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.15 Flow of Control Tokens

5.15.1 Branch Tokens

IJMP **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*.

SBNZ **82** *offset:8*

(x —)

Take branch if $x \neq 0$ using the supplied branch *offset.*

BNZ **83** *offset:16*

(num —)

Take branch if $x \neq 0$ using the supplied branch *offset.*

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

(x —)

Take branch if $x \neq 0$ using the supplied branch *offset.*

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 $num_1 \neq num_2$, 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 $num \neq u$, 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 $num \neq u$, take branch using *offset*.

5.15.2 Call Tokens

CALL0 … CALL39 **00 … 27**

(—) ; (R: — nest-sys)

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

SCALL **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.

ECALL **FE 61** *offset:32*

(—) ; (R: — nest-sys)

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

IMCALL **2A** u_1*:8* u_2*:8*

(—) ; (R: — nest-sys)

Execute a procedure in another module, where u_1 is the index into the currently executing module's Import List and u_2 is the index into the called module's Export List. Stack effects are dependent on the procedure called. See Section 4.9 for an explanation of modules.

ICALL **2B**

(xp —) ; (R: — nest-sys)

Call the procedure whose execution pointer is *xp*.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

RETURN **2C**

(—) ; (R: nest-sys —)

Return from the procedure.

5.15.3 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 3.2.4).

RDO **88** *offset:16*

(num_1 num_2| u_1 u_2 —) ; (R: — loop-sys)

Set up loop control parameters representing a loop index whose initial value is $num_2|u_2$, an upper limit of $num_1|u_1$ and a termination location specified by the in-line *offset*. An ambiguous condition exists if $num_1|u_1$ and $num_2|u_2$ 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 3.2.4).

RQDO **89** *offset:16*

(num_1 num_2| u_1 u_2 —) ; (R: — loop-sys)

If $num_1|u_1 = num_2|u_2$, 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 $num_2|u_2$, an upper limit of $num_1|u_1$ and a termination location at the given *offset*. An ambiguous condition exists if $num_1|u_1$ and $num_2|u_2$ 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 3.2.4).

RI **8A**

(— $num_1|u_1$) ; (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**

(— $num_1|u_1$) ; (R: $loop\text{-}sys_1$ $loop\text{-}sys_2$ — $loop\text{-}sys_1$ $loop\text{-}sys_2$)

Return the loop index from the outer do-loop structure $loop\text{-}sys_1$. 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.

RLEAVE **8B**

(—) ; (R: loop-sys —)

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

RLOOP **8C**

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

Add one to the loop index represented in *loop-sys*$_1$ 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**).

5.15.4 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**.

DOCLASS **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.*

EDOCLASS **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*.

5.15.5 Quoting Tokens

See Section 5.4.3 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.*

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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**.

5.16 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 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 num_1 num_2 — k*x | i*x num_2)*

If num_1 is non-zero, **THROW** with value num_2. If num_1 is zero, discard num_1 and num_2.

5.17 Date, Time, and Timing Tokens

GETTIME **EC**

(— u_1 u_2 u_3 u_4 u_5 u_6)

Return the current time and date. u_1 is the second {0...59}, u_2 is the minute {0…59}, u_3 is the hour {0...23}, u_4 is the day {1...31}, u_5 is the month {1...12}, and u_6 is the year {0...9999}.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SETTIME **FE 79**

(u_1 u_2 u_3 u_4 u_5 u_6 —)

Set the current time and date. u_1 is the second {0...59}, u_2 is the minute {0...59}, u_3 is the hour {0...23}, u_4 is the day {1...31}, u_5 is the month {1...12}, and u_6 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 2^{32}-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.

5.18 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 Appendix 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 Appendix C.

DEVOPEN **FE 93**

(dev — ior)

Open device *dev*. On buffered devices **DEVOPEN** clears all data from the buffers. *Ior* -32759 (Device must 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 anymore and all DEVtokens applied on a closed device return *ior* -32759 (Device must be opened). Note that closing a modem device with an open connection implicitly performs a hangup of the modem.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.
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.

DEVTIMEDREAD **FE 95**

(c-addr len_1 *dev — c-addr* len_2 *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 len_1 is its maximum length in address units. On return, len_2 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 Appendix 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 Appendix C. If the device is currently occupied because it has been selected by a previous execution of the **DEVOUTPUT** token, **DEVSTATUS** shall return the standard ior code for "device busy" (see Appendix C) until execution of the procedure passed to **DEVOUTPUT** completes.

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 supported devices in Appendix 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 **DEVOUTPUT** 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 Appendix C if it cannot. All exceptions 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**

(num_1 num_2 dev —)

Execute a device-dependent "set absolute position" action on device *dev* using num_1 as the horizontal co-ordinate and num_2 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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).

5.19 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 3.2.7) 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 ud_1 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 ud_2 as the quotient of ud_1 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NMBRS **FE 32**

(ud_1 — ud_2)

Convert at least one digit of ud_1 according to the rule for **NMBR**, continuing to convert digits until ud_2 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.

HOLD **FE 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.

SIGN **FE 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**

(ud_1 c-addr$_1$ len$_1$ — ud_2 c-addr$_2$ len$_2$)

Convert string to number. ud_2 is the unsigned result of converting the characters within the string specified by *c-addr*$_1$ *len*$_1$ into digits, using the number in BASE, and adding each into ud_1 after multiplying ud_1 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-addr*$_2$ is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. *len*$_2$ is the number of unconverted characters in the string. Exception -24 (Invalid numeric argument) may be thrown if ud_2 overflows during the conversion.

5.20 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 Appendix C.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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-addr$_1$ len$_1$ c-addr$_2$ len$_2$ — c-addr$_2$ len$_3$ 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 *len$_3$* 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$_2$* must provide adequate space for the answer from the card plus two status bytes containing SW1 and SW2.

CARDON **FE 74**

(c-addr len$_1$ — c-addr len$_2$ 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.

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.

5.21 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 Appendix C.

MAGREAD **FE 76**

(c-addr len$_1$ u — c-addr len$_2$ 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 Standard 7813 for a description of the format of this data. A particular device is not required to support all of these possibilities.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 len_1 is its maximum length. On return, len_2 gives the actual length of the string read. Refer to ISO Standard 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.

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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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
```

5.22 Socket Tokens

See Section 4.8 for a general account of the use of sockets.

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 sockets, except socket zero, must have no stack effect.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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, must have no stack effect. Exception -3327 (Invalid socket) shall be thrown if *u* is outside the range 0-63.

5.23 Database Services Tokens

The following tokens provide a mechanism for handling databases as described in Section 4.3. 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.

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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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* len_2*:8*

(c-addr len_1 *—)*

Store at most len_2 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 len_1 is less than len_2 then the destination in the database record buffer shall be space filled to len_2. 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 len_1 *u* len_2 *—)*

Store at most len_2 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 len_1 is less than len_2 then the destination in the database record buffer shall be space filled to len_2. 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.

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 token, and it shall cause Exception -3071 (Out of memory) to be thrown.

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DBSELECT **D1**

(num —)

Select record *num* in the currently selected database. This must 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 20_H) 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 20_H) 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 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 20_H) 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: — u_1 u_2)

Save the current database context on the return stack. u_1 is DBCURRENT, and u_2 is DBRECNUM.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DBRESTORE **DB**

(R: u_1 u_2 —)

Restore the most-recently-saved database context information from the return stack. u_1 is DBCURRENT, and u_2 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.

5.24 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 8859 (0 = common character set, 1 = Latin 1, etc.). *flag* is *true* if the code page has been selected.

LOADPAGE **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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

MSGLOAD **FE 83**

(c-addr — flag)

Install a message table at *c-addr*, including message numbers in the range 40-FF_H in the transient messages buffer, over-writing any previous messages with the same message numbers. Messages 40-BF_H may be provided only by a terminal application; messages C0-FF_H may be provided by an ICC. The message table must 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-FF_H.

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 must return string parameters for a zero-length message.

LANGUAGES **FE 81**

(—)

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

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 4.4.

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.

5.25 TLV Tokens

The tokens described in this section provide TLV management and access functions as described in Section 4.5.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.25.1 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 must 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 must be the same as the value field format of the corresponding TLV.

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 **TLVSTORE** 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

TLVBITSTORE **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.

TLVBITSET **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.

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

TLVBITCLEAR **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.

5.25.2 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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**

($c\text{-}addr_1$ len_1 $c\text{-}addr_2$ len_2 — $c\text{-}addr_2$ len_3)

Process len_1 bytes at $c\text{-}addr_1$ for a sequence of tag and length fields. For all tag fields encountered, the corresponding value fields are concatenated into the buffer at $c\text{-}addr_2$ up to its maximum size, len_2.

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 Appendix E.

Exception -507 (String too large) shall be thrown if len_2 is exceeded. The token returns the beginning of the destination string ($c\text{-}addr_2$) and the resulting output length (len_3, not greater than len_2).

TLVPLUSSTRING **FE D9**

(c-addr len_1 a-addr — c-addr len_2)

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* len_1. 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 (len_2). 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 4.5.2).

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

5.25.3 TLV Sequence Access

TLVFETCHTAG **FE D5**

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

Parse the input string at $c\text{-}addr_1$ for a tag field. Return the string $c\text{-}addr_2$ len_2 and the decoded tag value *u*. $c\text{-}addr_2$ points to the byte immediately following the tag, and len_2 is len_1 minus the length of the tag. The *flag* is *true* if the tag is entirely contained within the input string.

TLVFETCHLENGTH **FE D6**

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

Parse the input string at $c\text{-}addr_1$ for a length field. Return the string $c\text{-}addr_2$ len_2 and the decoded length value *u*. $c\text{-}addr_2$ points to the byte immediately following the length, and len_2 is len_1 minus the size of the length field. The *flag* is *true* if the length is entirely contained within the input string.

TLVFETCHVALUE **FE D4**

($c\text{-}addr_1$ len_1 — $c\text{-}addr_2$ len_2 $c\text{-}addr_3$ len_3 flag)

Split the input string at $c\text{-}addr_1$, which is expected to contain the length and value fields of a TLV. Decode the length field. Return the trailing string $c\text{-}addr_2$ len_2, the value field $c\text{-}addr_3$ len_3 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.

5.26 Hot Card List Tokens

Tokens in this group provide a mechanism for handling the hot card list file. See the discussion in Section 4.6 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 F_H's. This token must 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 F_H in the entry; see Section 4.6).

5.27 Cryptographic Algorithm Token

This token provides support for using cryptographic services. The supported cryptographic algorithms are discussed in more detail in Section 4.7.

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 (page 4–29). Specific stack parameters for each algorithm are given in Section 4.7.1 and following.

5.28 Module Management Tokens

The following tokens provide for the storage and execution of OTA modules in the Virtual Machine. See Section 4.9 for a general account of module handling.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 (page 4–36)). 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 Appendix B.

MODRELEASE **FE C8**

(—)

Release the resources used by the internal module buffer. This is required if premature loading of a module must be terminated by the application without registering the module in the module repository.

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 Appendix B.

MODCARDEXECUTE **FE C3**

*(i*x a-addr — j*x flag)*

Load the module at *a-addr*. *a-addr* 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 (page 4–40)). 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 database.

5.29 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 Appendix 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 that this token is implementation dependent, and is provided so that terminal specific programs (TRS) written using OTA tokens can provide a single callback routine for the operating system.

5.30 Miscellaneous Tokens

NOOP **2F**

(—)

Take no action.

DEPTH **FE 42**

(— num)

Return the data stack depth in cells.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

ENDPROC **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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6 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 3). 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. Must 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).

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 14: Module delivery format *(continued)*

Label	Size (bytes)	Content
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).

6.1 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 not registered 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.

6.2 Socket List

Each module must 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.

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

6.3 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

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 otherwise 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:

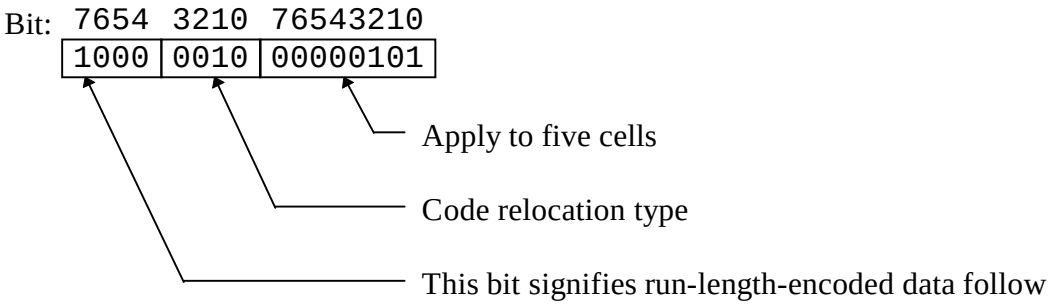
Table 16: Relocation specification

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 must 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.

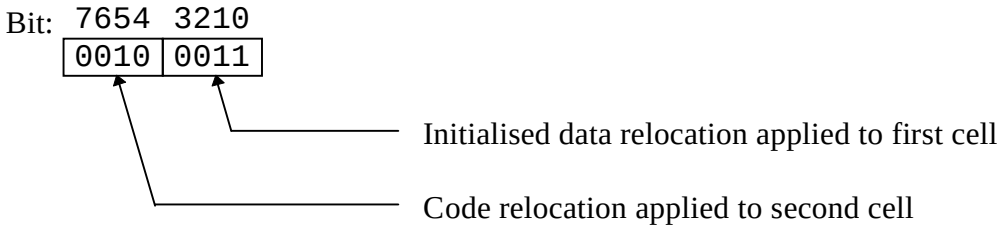


If the relocation byte does not have bit seven set, the low-order nibble defines the relocation type to apply to the next cell in the image, and the high-order nibble defines the type to apply

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

to the cell after that.



6.4 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

6.5 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**

6.6 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

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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**

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Appendix A: OTA Token Lists

This section presents numeric and alphabetic lists of tokens, with their numeric codes and names. Descriptions of the individual tokens may be found in Section 5 at the pages shown.

A.1 Numeric List of Tokens

Numeric code	Name	Page
00 … 27	**CALL0 … CALL39**	5-64
28 *offset:8*	**SCALL**	5-64
29 *offset:16*	**CALL**	5-64
2A *u1:8 u2:8*	**IMCALL**	5-64
2B	**ICALL**	5-64
2C	**RETURN**	5-65
2D	**CATCH**	5-67
2E	**THROW**	5-67
2F	**NOOP**	5-88
30 … 3F	**LIT0 … LIT15**	5-51
40 … 43	**PFRFETCH2 … PFRFETCH5**	5-60
44 … 4F	**TFRFETCH12 … TFRFETCH1**	5-60
50 … 53	**PFRSTORE2 … PFRSTORE5**	5-61
54 … 5F	**TFRSTORE12 … TFRSTORE1**	5-61
60 *u:8* … 63 *u:8*	**SLITU0 … SLITU3**	5-52
64 *u:8* … 67 *u:8*	**FETCHU0 … FETCHU3**	5-50
68 *u:8* … 6B *u:8*	**STOREU0 … STOREU3**	5-50
6C *u:16*	**LITU**	5-52
6D *u:8*	**SLIT**	5-51
6E *u:16*	**LIT**	5-51
6F *num:32*	**ELIT**	5-51
70 *u:8* … 73 *u:8*	**SLITD0 … SLITD3**	5-52
74 *u:8* … 77 *u:8*	**FETCHD0 … FETCHD3**	5-50
78 *u:8* … 7B *u:8*	**STORED0 … STORED3**	5-51
7C *u:16*	**LITD**	5-52
7D *offset:16*	**LITC**	5-52
7E	**LITMINUS1**	5-51
7F *u:8*	**NLIT**	5-51
80 *offset:8*	**SBZ**	5-63
81 *offset:16*	**BZ**	5-63
82 *offset:8*	**SBNZ**	5-63
83 *offset:16*	**BNZ**	5-63
84 *offset:8*	**SBRA**	5-63
85 *offset:16*	**BRA**	5-63
86 *u:8 offset:8*	**SROFLIT**	5-64
87 *u:16 offset:8*	**ROFLIT**	5-64
88 *offset:16*	**RDO**	5-65
89 *offset:16*	**RQDO**	5-65
8A	**RI**	5-65
8B	**RLEAVE**	5-65
8C	**RLOOP**	5-66
8D	**RPLUSLOOP**	5-66
8E *offset:16*	**QUOTE**	5-66
8F	**ENDQUOTE**	5-67
90	**DROP**	5-47
91	**DUP**	5-47
92	**SWAP**	5-47
93	**OVER**	5-47
94	**NIP**	5-47
95	**TUCK**	5-47
96	**ROT**	5-47
97	**MINUSROT**	5-47
98	**QDUP**	5-47
99	**RFROM**	5-48
9A	**TOR**	5-48
9B	**RFETCH**	5-48
9C	**TWOSWAP**	5-48
9D	**TWODROP**	5-48
9E	**TWODUP**	5-48
9F	**TWOTOR**	5-48
A0	**TWORFROM**	5-48
A1	**TWOOVER**	5-48
A2	**TWORFETCH**	5-48
A3	**FETCH**	5-49
A4	**STORE**	5-49
A5	**CFETCH**	5-49
A6	**CSTORE**	5-49
A7	**BCDFETCH**	5-49
A8	**BCDSTORE**	5-50
A9	**ADD**	5-53
AA	**SUB**	5-53
AB	**MUL**	5-53
AC	**MOD**	5-53
AD	**AND**	5-54
AE	**OR**	5-54
AF	**MSLMOD**	5-54
B0	**SHRNU**	5-55
B1	**SHL**	5-55
B2	**SHLN**	5-55
B3	**CMPEQ**	5-56

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Numeric code	Name	Page
B4	**CMPNE**	5-56
B5	**CMPLT**	5-57
B6	**CMPLTU**	5-56
B7	**CMPGT**	5-57
B8	**CMPGEU**	5-56
B9	**CMPGTU**	5-56
BA	**SETEQ**	5-56
BB	**SETLT**	5-56
BC	**SETNE**	5-55
BD	**WITHIN**	5-57
BE *num2:8*	**SADDLIT**	5-53
BF *u:8*	**SMULLIT**	5-53
C0	**TLVFIND**	5-82
C1	**TLVFETCH**	5-82
C2	**TLVSTORE**	5-82
C3	**TLVPARSE**	5-83
C4	**TLVSTATUS**	5-85
C5	**MOVE**	5-57
C6	**FILL**	5-57
C7	**COMPARE**	5-58
C8	**MINUSTRAILING**	5-59
CA	**PLUSSTRING**	5-58
CB	**COUNT**	5-58
CC	**INCR**	5-55
CD	**BNFETCH**	5-50
CE	**BNSTORE**	5-50
CF 00	**PLUGGABLE**	5-75
CF *u:8*	**DOSOCKET**	5-75
D0	**DBMAKECURRENT**	5-76
D1	**DBSELECT**	5-79
D2	**DBAVAIL**	5-78
D3	**DBSIZE**	5-76
D4 *u:8 len:8*	**DBSTRFETCHLIT**	5-77
D5 *u:8 len2:8*	**DBSTRSTORELIT**	5-78
D6 *u:8*	**DBCFETCHLIT**	5-77
D7 *u:8*	**DBCSTORELIT**	5-77
D8 *u:8*	**DBFETCHLIT**	5-76
D9 *u:8*	**DBSTORELIT**	5-76
DA	**DBSAVE**	5-79
DB	**DBRESTORE**	5-80
DD	**ADDLIT1**	5-53
DE	**SUBLIT1**	5-53
DF *u:16 offset:16*	**DOCLASS**	5-66
E0 *num:8*	**SFRADDR**	5-62
E1 *num:8*	**SFRFETCH**	5-61
E2 *num:8*	**SFRSTORE**	5-61
E3 *num:16*	**FRADDR**	5-62
E4 *num:16*	**FRFETCH**	5-61
E5 *num:16*	**FRSTORE**	5-61
E6	**BYTE**	5-46
E6 44 … E6 4F	**BYTE TFRFETCH12 …** **BYTE TFRFETCH1**	5-60
E6 54 … E6 5F	**BYTE TFRSTORE12 …** **BYTE TFRSTORE1**	5-61
E6 64 *u:8* … E6 67 *u:8*	**BYTE FETCHU0 …** **BYTE FETCHU3**	5-50

Numeric code	Name	Page
E6 68 *u:8* … E6 6B *u:8*	**BYTE STOREU0 …** **BYTE STOREU3**	5-50
E6 74 *u:8* … E6 77 *u:8*	**BYTE FETCHD0 …** **BYTE FETCHD3**	5-50
E6 78 *u:8* … E6 7B *u:8*	**BYTE STORED0 …** **BYTE STORED3**	5-51
E6 E1 *num:8*	**BYTE SFRFETCH**	5-61
E6 E2 *num:8*	**BYTE SFRSTORE**	5-61
E6 E4 *num:16*	**BYTE FRFETCH**	5-61
E6 E5 *num:16*	**BYTE FRSTORE**	5-61
E7	**WIDEN**	5-55
E8 *u1:8 u2:8*	**SMAKEFRAME**	5-60
E9	**RELFRAME**	5-60
EA	**CEXTEND**	5-62
EB	**RELEASE**	5-62
EC	**GETTIME**	5-67
ED	**GETMS**	5-68
EE	**MS**	5-68
EF	**CARDABSENT**	5-73
F0	**IJMP**	5-63
F1	**NEGATE**	5-54
F2 *counted-string*	**STRLIT**	5-59
F3	**TLVTRAVERSE**	5-84
F8	**SETOP**	5-71
F9	**NMBR**	5-71
FA	**LTNMBR**	5-71
FB	**NMBRGT**	5-72
FC	**TONUMBER**	5-72
FD	**USERVAR**	5-52
FE	**SECONDARY**	5-47
FE 00	**PROC**	5-89
FE 01	**ENDPROC**	5-89
FE 02 *counted-string*	**HEADER**	5-89
FE 10	**MIN**	5-55
FE 11	**MAX**	5-55
FE 12	**ABS**	5-55
FE 13	**CMPLE**	5-57
FE 14	**CMPLEU**	5-56
FE 15	**SETGE**	5-56
FE 17	**CMPGE**	5-57
FE 18	**SETGT**	5-56
FE 19	**SETLE**	5-56
FE 20	**DADD**	5-54
FE 21	**DCMPLT**	5-57
FE 22	**DNEGATE**	5-54
FE 30	**TWOFETCH**	5-49
FE 31	**TWOSTORE**	5-49
FE 32	**NMBRS**	5-72
FE 33	**HOLD**	5-72
FE 34	**SIGN**	5-72
FE 35	**MINUSZEROS**	5-59
FE 36	**EXTEND**	5-62
FE 37	**RJ**	5-65
FE 38	**SLASHSTRING**	5-59
FE 40	**SCAN**	5-58
FE 41	**SKIP**	5-58

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Numeric code	Name	Page
FE 42	**DEPTH**	5-88
FE 43	**PICK**	5-48
FE 44	**TWOROT**	5-49
FE 45	**CNFETCH**	5-59
FE 46	**CNSTORE**	5-59
FE 50	**XOR**	5-54
FE 51	**DIV**	5-53
FE 52	**DIVU**	5-53
FE 53	**MODU**	5-53
FE 54	**MSLMODU**	5-54
FE 55	**MMUL**	5-54
FE 56	**MMULU**	5-54
FE 57	**SHRN**	5-55
FE 60 *offset:32*	**EBRA**	5-63
FE 61 *offset:32*	**ECALL**	5-64
FE 62 *offset:32*	**EBNZ**	5-63
FE 63 *offset:32*	**EBZ**	5-63
FE 64 *u1:16 u2:16*	**MAKEFRAME**	5-60
FE 65	**SETCALLBACK**	5-88
FE 66	**OSCALL**	5-88
FE 67 *offset:16*	**ROF**	5-64
FE 70	**CRYPTO**	5-86
FE 72	**CARDINIT**	5-73
FE 73	**CARD**	5-73
FE 74	**CARDON**	5-73
FE 75	**CARDOFF**	5-73
FE 76	**MAGREAD**	5-73
FE 77	**MAGWRITE**	5-75
FE 79	**SETTIME**	5-68
FE 80	**MSGFETCH**	5-81
FE 81	**LANGUAGES**	5-81
FE 83	**MSGLOAD**	5-81
FE 84	**CHOOSELANG**	5-80
FE 85	**CODEPAGE**	5-80
FE 86	**MSGINIT**	5-80
FE 87	**LOADPAGE**	5-80
FE 88	**MSGUPDATE**	5-81
FE 89	**MSGDELETE**	5-81
FE 90	**DEVEKEY**	5-69
FE 91	**DEVEKEYQ**	5-69
FE 92	**DEVEMIT**	5-69
FE 93	**DEVOPEN**	5-68
FE 94	**DEVREAD**	5-69
FE 95	**DEVTIMEDREAD**	5-69
FE 96	**DEVWRITE**	5-70
FE 97	**DEVSTATUS**	5-70
FE 98	**DEVIOCTL**	5-70
FE 99	**DEVOUTPUT**	5-70
FE 9A	**DEVATXY**	5-70
FE 9B	**DEVCONNECT**	5-71
FE 9C	**DEVHANGUP**	5-71
FE 9D	**DEVBREAK**	5-71
FE 9E	**DEVCLOSE**	5-68
FE 9F	**GETOP**	5-71
FE B0	**DBADDREC**	5-78
FE B1	**DBMATCHBYKEY**	5-79

Numeric code	Name	Page
FE B3	**DBDELREC**	5-80
FE B4	**DBINIT**	5-78
FE B5	**DBADDBYKEY**	5-79
FE B6	**DBDELBYKEY**	5-79
FE C0	**MODDELETE**	5-87
FE C1	**MODEXECUTE**	5-87
FE C2	**MODINIT**	5-87
FE C3	**MODCARDEXECUTE**	5-87
FE C4	**MODAPPEND**	5-87
FE C6	**MODCHANGED**	5-88
FE C7	**MODREGISTER**	5-87
FE C8	**MODRELEASE**	5-87
FE C9	**MODULES**	5-88
FE D0	**TLVBITFETCH**	5-83
FE D1	**TLVBITSTORE**	5-83
FE D2	**TLVINIT**	5-82
FE D3	**TLVFETCHRAW**	5-82
FE D4	**TLVFETCHVALUE**	5-85
FE D5	**TLVFETCHTAG**	5-85
FE D6	**TLVFETCHLENGTH**	5-85
FE D7	**TLVFORMAT**	5-84
FE D8	**TLVTAG**	5-84
FE D9	**TLVPLUSSTRING**	5-84
FE DA	**TLVPLUSDOL**	5-84
FE DB	**TLVCLEAR**	5-85
FE DC	**TLVBITSET**	5-83
FE DD	**TLVBITCLEAR**	5-83
FE DE	**TLVSTORERAW**	5-82
FE E0	**HOTINIT**	5-86
FE E1	**HOTADD**	5-86
FE E2	**HOTDELETE**	5-86
FE E3	**HOTFIND**	5-86
FE E4	**DBFETCH**	5-76
FE E5	**DBCFETCH**	5-77
FE E6	**DBSTRFETCH**	5-78
FE E7	**DBSTORE**	5-77
FE E8	**DBCSTORE**	5-77
FE E9	**DBSTRSTORE**	5-78
FE F0	**QTHROW**	5-67
FE F1 *u:16*	**DOCREATE**	5-66
FE F2	**IDOSOCKET**	5-76
FE F3	**PLUGSOCKET**	5-75
FE F4 *num:32 offset:32*	**EDOCLASS**	5-66
FE F5 *num:32*	**EDOCREATE**	5-66
FE F6 *offset:32*	**ELITC**	5-52
FE F7 *num:32*	**ELITD**	5-52
FE F8 *num:32*	**ELITU**	5-52
FF	**BREAKPNT**	5-89

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.2 Alphabetic List of Tokens

Numeric value	Name	Page
FE 12	**ABS**	5-55
A9	**ADD**	5-53
DD	**ADDLIT1**	5-53
AD	**AND**	5-54
A7	**BCDFETCH**	5-49
A8	**BCDSTORE**	5-50
CD	**BNFETCH**	5-50
CE	**BNSTORE**	5-50
83 *offset:16*	**BNZ**	5-63
85 *offset:16*	**BRA**	5-63
FF	**BREAKPNT**	5-89
E6	**BYTE**	5-46
E6 74 *u:8* … E6 77 *u:8*	**BYTE FETCHD0 … BYTE FETCHD3**	5-50
E6 64 *u:8* … E6 67 *u:8*	**BYTE FETCHU0 … BYTE FETCHU3**	5-50
E6 E4 *num:16*	**BYTE FRFETCH**	5-61
E6 E5 *num:16*	**BYTE FRSTORE**	5-61
E6 E1 *num:8*	**BYTE SFRFETCH**	5-61
E6 E2 *num:8*	**BYTE SFRSTORE**	5-61
E6 78 *u:8* … E6 7B *u:8*	**BYTE STORED0 … BYTE STORED3**	5-51
E6 68 *u:8* … E6 6B *u:8*	**BYTE STOREU0 … BYTE STOREU3**	5-50
E6 44 … E6 4F	**BYTE TFRFETCH12 … BYTE TFRFETCH1**	5-60
E6 54 … E6 5F	**BYTE TFRSTORE12 … BYTE TFRSTORE1**	5-61
81 *offset:16*	**BZ**	5-63
29 *offset:16*	**CALL**	5-64
00 … 27	**CALL0 … CALL39**	5-64
FE 73	**CARD**	5-73
EF	**CARDABSENT**	5-73
FE 72	**CARDINIT**	5-73
FE 75	**CARDOFF**	5-73
FE 74	**CARDON**	5-73
2D	**CATCH**	5-67
EA	**CEXTEND**	5-62
A5	**CFETCH**	5-49
FE 84	**CHOOSELANG**	5-80
B3	**CMPEQ**	5-56
FE 17	**CMPGE**	5-57
B8	**CMPGEU**	5-56
B7	**CMPGT**	5-57
B9	**CMPGTU**	5-56
FE 13	**CMPLE**	5-57
FE 14	**CMPLEU**	5-56
B5	**CMPLT**	5-57
B6	**CMPLTU**	5-56
B4	**CMPNE**	5-56
FE 45	**CNFETCH**	5-59
FE 46	**CNSTORE**	5-59
FE 85	**CODEPAGE**	5-80
C7	**COMPARE**	5-58
CB	**COUNT**	5-58
FE 70	**CRYPTO**	5-86
A6	**CSTORE**	5-49
FE 20	**DADD**	5-54
FE B5	**DBADDBYKEY**	5-79
FE B0	**DBADDREC**	5-78
D2	**DBAVAIL**	5-78
FE E5	**DBCFETCH**	5-77
D6 *u:8*	**DBCFETCHLIT**	5-77
FE E8	**DBCSTORE**	5-77
D7 *u:8*	**DBCSTORELIT**	5-77
FE B6	**DBDELBYKEY**	5-79
FE B3	**DBDELREC**	5-80
FE E4	**DBFETCH**	5-76
D8 *u:8*	**DBFETCHLIT**	5-76
FE B4	**DBINIT**	5-78
D0	**DBMAKECURRENT**	5-76
FE B1	**DBMATCHBYKEY**	5-79
DB	**DBRESTORE**	5-80
DA	**DBSAVE**	5-79
D1	**DBSELECT**	5-79
D3	**DBSIZE**	5-76
FE E7	**DBSTORE**	5-77
D9 *u:8*	**DBSTORELIT**	5-76
FE E6	**DBSTRFETCH**	5-78
D4 *u:8 len:8*	**DBSTRFETCHLIT**	5-77
FE E9	**DBSTRSTORE**	5-78
D5 *u:8* len2:8	**DBSTRSTORELIT**	5-78
FE 21	**DCMPLT**	5-57
FE 42	**DEPTH**	5-88
FE 9A	**DEVATXY**	5-70
FE 9D	**DEVBREAK**	5-71
FE 9E	**DEVCLOSE**	5-68
FE 9B	**DEVCONNECT**	5-71
FE 90	**DEVEKEY**	5-69
FE 91	**DEVEKEYQ**	5-69
FE 92	**DEVEMIT**	5-69
FE 9C	**DEVHANGUP**	5-71
FE 98	**DEVIOCTL**	5-70
FE 93	**DEVOPEN**	5-68
FE 99	**DEVOUTPUT**	5-70
FE 94	**DEVREAD**	5-69
FE 97	**DEVSTATUS**	5-70
FE 95	**DEVTIMEDREAD**	5-69
FE 96	**DEVWRITE**	5-70
FE 51	**DIV**	5-53
FE 52	**DIVU**	5-53
FE 22	**DNEGATE**	5-54
DF *u:16 offset:16*	**DOCLASS**	5-66
FE F1 *u:16*	**DOCREATE**	5-66
CF *u:8*	**DOSOCKET**	5-75
90	**DROP**	5-47

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Numeric value	Name	Page
91	**DUP**	5-47
FE 62 *offset:32*	**EBNZ**	5-63
FE 60 *offset:32*	**EBRA**	5-63
FE 63 *offset:32*	**EBZ**	5-63
FE 61 *offset:32*	**ECALL**	5-64
FE F4 *num:32 offset:32*	**EDOCLASS**	5-66
FE F5 *num:32*	**EDOCREATE**	5-66
6F *num:32*	**ELIT**	5-51
FE F6 *offset:32*	**ELITC**	5-52
FE F7 *num:32*	**ELITD**	5-52
FE F8 *num:32*	**ELITU**	5-52
FE 01	**ENDPROC**	5-89
8F	**ENDQUOTE**	5-67
FE 36	**EXTEND**	5-62
A3	**FETCH**	5-49
74 *u:8* … 77 *u:8*	**FETCHD0 … FETCHD3**	5-50
64 *u:8* … 67 *u:8*	**FETCHU0 … FETCHU3**	5-50
C6	**FILL**	5-57
E3 *num:16*	**FRADDR**	5-62
E4 *num:16*	**FRFETCH**	5-61
E5 *num:16*	**FRSTORE**	5-61
ED	**GETMS**	5-68
FE 9F	**GETOP**	5-71
EC	**GETTIME**	5-67
FE 02 *counted-string*	**HEADER**	5-89
FE 33	**HOLD**	5-72
FE E1	**HOTADD**	5-86
FE E2	**HOTDELETE**	5-86
FE E3	**HOTFIND**	5-86
FE E0	**HOTINIT**	5-86
2B	**ICALL**	5-64
FE F2	**IDOSOCKET**	5-76
F0	**IJMP**	5-63
2A *u1:8 u2:8*	**IMCALL**	5-64
CC	**INCR**	5-55
FE 81	**LANGUAGES**	5-81
6E *u:16*	**LIT**	5-51
30 … 3F	**LIT0 … LIT15**	5-51
7D *offset:16*	**LITC**	5-52
7C *u:16*	**LITD**	5-52
7E	**LITMINUS1**	5-51
6C *u:16*	**LITU**	5-52
FE 87	**LOADPAGE**	5-80
FA	**LTNMBR**	5-71
FE 76	**MAGREAD**	5-73
FE 77	**MAGWRITE**	5-75
FE 64 *u1:16 u2:16*	**MAKEFRAME**	5-60
FE 11	**MAX**	5-55
FE 10	**MIN**	5-55
97	**MINUSROT**	5-47
C8	**MINUSTRAILING**	5-59
FE 35	**MINUSZEROS**	5-59
FE 55	**MMUL**	5-54

Numeric value	Name	Page
FE 56	**MMULU**	5-54
AC	**MOD**	5-53
FE C4	**MODAPPEND**	5-87
FE C3	**MODCARDEXECUTE**	5-87
FE C6	**MODCHANGED**	5-88
FE C0	**MODDELETE**	5-87
FE C1	**MODEXECUTE**	5-87
FE C2	**MODINIT**	5-87
FE C7	**MODREGISTER**	5-87
FE C8	**MODRELEASE**	5-87
FE 53	**MODU**	5-53
FE C9	**MODULES**	5-88
C5	**MOVE**	5-57
EE	**MS**	5-68
FE 89	**MSGDELETE**	5-81
FE 80	**MSGFETCH**	5-81
FE 86	**MSGINIT**	5-80
FE 83	**MSGLOAD**	5-81
FE 88	**MSGUPDATE**	5-81
AF	**MSLMOD**	5-54
FE 54	**MSLMODU**	5-54
AB	**MUL**	5-53
F1	**NEGATE**	5-54
94	**NIP**	5-47
7F *u:8*	**NLIT**	5-51
F9	**NMBR**	5-71
FB	**NMBRGT**	5-72
FE 32	**NMBRS**	5-72
2F	**NOOP**	5-88
AE	**OR**	5-54
FE 66	**OSCALL**	5-88
93	**OVER**	5-47
40 … 43	**PFRFETCH2 … PFRFETCH5**	5-60
50 … 53	**PFRSTORE2 … PFRSTORE5**	5-61
FE 43	**PICK**	5-48
CF 00	**PLUGGABLE**	5-75
FE F3	**PLUGSOCKET**	5-75
CA	**PLUSSTRING**	5-58
FE 00	**PROC**	5-89
98	**QDUP**	5-47
FE F0	**QTHROW**	5-67
8E *offset:16*	**QUOTE**	5-66
88 *offset:16*	**RDO**	5-65
EB	**RELEASE**	5-62
E9	**RELFRAME**	5-60
2C	**RETURN**	5-65
9B	**RFETCH**	5-48
99	**RFROM**	5-48
8A	**RI**	5-65
FE 37	**RJ**	5-65
8B	**RLEAVE**	5-65
8C	**RLOOP**	5-66
FE 67 *offset:16*	**ROF**	5-64
87 *u:16 offset:8*	**ROFLIT**	5-64

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Numeric value	Name	Page
96	**ROT**	5-47
8D	**RPLUSLOOP**	5-66
89 *offset:16*	**RQDO**	5-65
BE num2:8	**SADDLIT**	5-53
82 *offset:8*	**SBNZ**	5-63
84 *offset:8*	**SBRA**	5-63
80 *offset:8*	**SBZ**	5-63
28 *offset:8*	**SCALL**	5-64
FE 40	**SCAN**	5-58
FE	**SECONDARY**	5-47
FE 65	**SETCALLBACK**	5-88
BA	**SETEQ**	5-56
FE 15	**SETGE**	5-56
FE 18	**SETGT**	5-56
FE 19	**SETLE**	5-56
BB	**SETLT**	5-56
BC	**SETNE**	5-55
F8	**SETOP**	5-71
FE 79	**SETTIME**	5-68
E0 *num:8*	**SFRADDR**	5-62
E1 *num:8*	**SFRFETCH**	5-61
E2 *num:8*	**SFRSTORE**	5-61
B1	**SHL**	5-55
B2	**SHLN**	5-55
FE 57	**SHRN**	5-55
B0	**SHRNU**	5-55
FE 34	**SIGN**	5-72
FE 41	**SKIP**	5-58
FE 38	**SLASHSTRING**	5-59
6D *u:8*	**SLIT**	5-51
70 *u:8* … 73 *u:8*	**SLITD0 … SLITD3**	5-52
60 *u:8* … 63 *u:8*	**SLITU0 … SLITU3**	5-52
E8 *u1:8 u2:8*	**SMAKEFRAME**	5-60
BF *u:8*	**SMULLIT**	5-53
86 *u:8 offset:8*	**SROFLIT**	5-64
A4	**STORE**	5-49
78 *u:8* … 7B *u:8*	**STORED0 … STORED3**	5-51
68 *u:8* … 6B *u:8*	**STOREU0 … STOREU3**	5-50
F2 *counted-string*	**STRLIT**	5-59
AA	**SUB**	5-53
DE	**SUBLIT1**	5-53
92	**SWAP**	5-47
44 … 4F	**TFRFETCH12 … TFRFETCH1**	5-60
54 … 5F	**TFRSTORE12 … TFRSTORE1**	5-61
2E	**THROW**	5-67
FE DD	**TLVBITCLEAR**	5-83
FE D0	**TLVBITFETCH**	5-83
FE DC	**TLVBITSET**	5-83
FE D1	**TLVBITSTORE**	5-83
FE DB	**TLVCLEAR**	5-85
C1	**TLVFETCH**	5-82
FE D6	**TLVFETCHLENGTH**	5-85

Numeric value	Name	Page
FE D3	**TLVFETCHRAW**	5-82
FE D5	**TLVFETCHTAG**	5-85
FE D4	**TLVFETCHVALUE**	5-85
C0	**TLVFIND**	5-82
FE D7	**TLVFORMAT**	5-84
FE D2	**TLVINIT**	5-82
C3	**TLVPARSE**	5-83
FE DA	**TLVPLUSDOL**	5-84
FE D9	**TLVPLUSSTRING**	5-84
C4	**TLVSTATUS**	5-85
C2	**TLVSTORE**	5-82
FE DE	**TLVSTORERAW**	5-82
FE D8	**TLVTAG**	5-84
F3	**TLVTRAVERSE**	5-84
FC	**TONUMBER**	5-72
9A	**TOR**	5-48
95	**TUCK**	5-47
9D	**TWODROP**	5-48
9E	**TWODUP**	5-48
FE 30	**TWOFETCH**	5-49
A1	**TWOOVER**	5-48
A2	**TWORFETCH**	5-48
A0	**TWORFROM**	5-48
FE 44	**TWOROT**	5-49
FE 31	**TWOSTORE**	5-49
9C	**TWOSWAP**	5-48
9F	**TWOTOR**	5-48
FD	**USERVAR**	5-52
E7	**WIDEN**	5-55
BD	**WITHIN**	5-57
FE 50	**XOR**	5-54

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Appendix B: Exceptions and I/O Return Codes

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.1 Exceptions and IOR codes

Table 20 below shows the ANS Forth codes that may be used in OTA kernels. Table 21 gives the OTA-specific throw codes and Table 22 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 20: 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 21: 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	FFFFF101	Undefined TLV. Cannot find TLV tag name.
-3838	FFFFF102	TLV too large. TLV value field is longer than 252 bytes.
-3837	FFFFF103	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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 21: OTA `THROW` codes *(continued)*

Decimal	Hex	Description
-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.

Table 22: 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-32239	FFFF8211	Outside border.
-31983	FFFF8311	Printer Off-line.
-31982	FFFF8312	Printer out of paper.
-31981	FFFF8313	Printer has asserted error signal.

© ISO/IEC 2001 – All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 22: OTA I/O return codes *(continued)*

Decimal	Hex	Description
-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	FFFF8418	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	FFFF8519	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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Appendix C: Device Control

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.1 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 23.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 23: 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 24 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, **DEV**`tokens` 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 **DEV**`token`. The column labelled “ior?” in Table 24 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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 24: Token — device number cross reference

	ior?	-1 db	0 kb	1 dis	2 prn	3 prn	4 mo	5 icc	6 icc	7 mg	8 kb	9 dis	10 ser	11 par	12 par	13 pw	14 ven	15 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	no	X	X				X				X		X					
DEVEKEYQ	no	X	X				X				X		X					
DEVEMIT	no	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	X		X	X	X	X	X	X	X
DEVATXY	no	X		X								X						
DEVCONNECT	yes						X						X					
DEVHANGUP	yes						X						X					
DEVBREAK	yes						X						X					

C.2 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 keyboard and display, communicating with the terminal through an interactive development link. Table 25 gives the ior codes appropriate for the debug device.

Table 25: Debug device I/O return code s

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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 25: Debug device I/O return codes *(continued)*

Status Decimal	Status Hex	Description
-32757	FFFF800B	Device cannot be opened.

C.3 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 646-1983 for values in the range 20_H - $7E_H$. The standard keyboard encodings are given in Table 26 (all key values in hexadecimal). A VM implementation is not mandated to return any of these keys, but must use the values listed for any that it does return.

Table 26: 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
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 documented in Section 5.18, where *dev* is the keyboard device code. Table 27 specifies the **`DEVIOCTL`** arguments for the keyboard device.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 28 gives the ior codes appropriate for keyboards.

Table 27: 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 28: 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.

C.4 Display and Printer Output

Devices that support text output (display and printer devices) must provide interpretation of certain control characters appearing as output characters either individually or in output strings. The codes and the effects they must produce are listed in Table 29; codes are specified in hexadecimal. Codes in the range $0\text{-}1F_H$ not appearing in this table are implementation-specific.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 29: 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.

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 29) 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 documented in Section 5.18, where *dev* is the display or printer device code. Table 30 specifies the **`DEVIOCTL`** arguments for the display device, while Table 31 specifies the **`DEVIOCTL`** arguments for the printer device. Display ior codes are given in Table 32, and printer ior codes are given in Table 33.

Table 30: 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

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 30: DEVIOCTL parameters for display device

Function code *(fn)*	Meaning
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.

Table 31: 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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 32: 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-32239	FFFF8211	Outside border.

Table 33: 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must 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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

C.5 Serial Port Management

Control of the serial devices is provided by I/O control functions implemented through the generic **DEVIOCTL** token documented in Section 5.18, where *dev* is the serial device code. Other stack arguments are documented in Table 34 below. Serial port ior codes are given in Table 35.

Table 34: 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.

Table 35: 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 35: Serial port device I/O return codes *(continued)*

Status Decimal	Status Hex	Description
-31726	FFFF8412	Invalid serial parameters.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.

C.6 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 documented in Section 5.18, where *dev* is the modem device code. Other stack arguments are documented in Table 36 below. The modem ior codes are given in Table 37.

Table 36: DEVIOCTL parameters for modem devic e

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.
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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 36: DEVIOCTL parameters for modem device *(continued)*

Function code *(fn)*	Meaning
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 37: 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must 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.
-31722	FFFF8416	No dial tone.
-31721	FFFF8417	Modem already connected.
-31720	FFFF8418	Connection in progress.

C.7 ICC Card Handling

Control of the ICC card devices is provided by a set of dedicated functions documented in Section 5.20 and by the I/O control functions implemented through the generic **`DEVIOCTL`**

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

token documented in Section 5.18, where *dev* is the ICC card reader 1 or the ICC card reader 2 device code. Table 38 specifies the **`DEVIOCTL`** arguments for the ICC card reader device. ICC card reader device ior codes are given in Table 39 below.

Table 38: DEVIOCTL parameters for ICC card reader device

Function code *(fn)*	Meaning
01	Assign *dev* (ICC card reader 1 or ICC card reader 2) to the requested card slot. *Addr* points to a cell containing the slot number to select. *Num* is set to 1. Slots count from 0. Exception -21 (Unsupported operation) shall be thrown if the requested slot is not supported by the reader in question. All single slot readers return 0 (Success) for slot 0 and throw -21 (Unsupported operation) for anything else.
04	Set 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.

Table 39: ICC card reader 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 39: ICC card reader device I/O return codes *(continued)*

Status Decimal	Status Hex	Description
-31463	FFFF8519	Card partially in reader.

C.8 Magnetic Stripe Handling

Control of the magnetic stripe device is provided by I/O control functions implemented through the generic **`DEVIOCTL`** token documented in Section 5.18, where *dev* is the magnetic stripe code. Other stack arguments are documented in Table 40 below. Magnetic card reader device ior codes are given in Table 41.

Table 40: DEVIOCTL parameters for magnetic card reader device

Function code *(fn)*	Meaning
04	Set time-out. *Addr* points to a cell containing the time in milliseconds before **`DEVTIMEDREAD`** and **`MAGREAD`** return. *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.

Table 41: Magnetic card reader 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-31215	FFFF8611	Transmission error between reader and magstripe.
-31214	FFFF8612	Output buffer overflow.
-31213	FFFF8613	Write operation failed.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

C.9 Power Management

Control of the power management device is provided by I/O control functions implemented through the generic **`DEVIOCTL`** token documented in Section 5.18, where *dev* is the power management device code. Other stack arguments are documented in Table 42 below, and ior codes in Table 43.

Table 42: DEVIOCTL parameters for power management device

Function code *(fn)*	Meaning
01	Return charge state of battery. *Num* is set to 1. On return, *addr* points to a cell coded as follows : Battery charge state : 0-100, 0=uncharged, 100=fully charged.
02	Return current power source. *Num* is set to 1. On return, *addr* points to a cell coded as follows : 1 = powered by battery 2 = non depleting power source
03	Power down the terminal. *Num* is set to 0. *Addr* is not used by this function and may be any arbitrary non zero value.
04	Disallow terminal power-off initiated by the "off" button. *Num* is set to 0. *Addr* is not used by this function and may be any arbitrary non zero value.
05	Allow terminal power-off initiated by the "off" button. *Num* is set to 0. *Addr* is not used by this function and may be any arbitrary non zero value.
06	Inquire why power has been supplied to the terminal. *Num* is set to 1. On return, *addr* points to a cell coded as follows : 1 = Normal Power-on 2 = Periodic wake-up 3 = Response to call
07	Put the terminal in sleep mode. *Num* is set to 1. *Addr* points to a cell coded as follows : 0 = put terminal in sleep mode 1 = wake-up terminal

Table 43: Power management device I/O return codes

Status Decimal	Status Hex	Description
-32767	FFFF8001	Device valid but currently not responding.
-32764	FFFF8004	Device Error.
-32763	FFFF8005	Unsupported device.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 43: Power management device I/O return codes *(continued)*

Status Decimal	Status Hex	Description
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.

C.10 Vending Machine Control

Control of the vending machine device is provided by I/O control functions implemented through the generic **`DEVIOCTL`** token documented in Section 5.18, where *dev* is the vending machine device code, *num* is always zero, and *addr* is used to pass vending amount on certain calls. Other stack arguments are documented in Table 44 below, and valid ior codes are in Table 45 below.

Table 44: DEVIOCTL parameters for vending machine device

Function code *(fn)*	Meaning
01	Begin vending session. Vending amount returned in *addr.*
02	Continue vending session. Next vending amount returned in *addr.*
03	Get vending session status.
04	Get vending controller transaction approval.
05	Get vending controller transaction denial.

Table 45: Vending machine 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 must be initialised.
-32761	FFFF8007	Device busy.
-32760	FFFF8008	Insufficient resources.
-32759	FFFF8009	Device must be opened.
-32758	FFFF800A	Device already open.
-32757	FFFF800B	Device cannot be opened.
-30455	FFFF8909	Vending terminated.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Appendix D: Operating System Calls

This section provides reference information for generic Operating System functions (**OSCALL**) in OTA. Note that not all functions have to be implemented by the Virtual Machine. However, if a function is implemented, its interface should follow the description herein, as far as the particular hardware configuration permits.

Table 46: OSCALL functions

Function code *(fn)*	Meaning
01	Initiate terminal dependent procedure to replace the kernel. *Num* is set to 0. *Addr* is not used by this function and may be any arbitrary non zero value.
02	Return serial number of the terminal. On return,*addr* points to a counted string containing the serial number of the terminal. *Num* is not used by this function and may be any arbitrary value.
03	Commit all the non-volatile data to non-volatile memory. *Addr* and *num* are not used by this function. For terminals that have not to deal with flash this may be a NO-OP.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Appendix E: Rules for Using a Data Object List (DOL)[1]

In several instances, the terminal is asked to build a flexible list of data elements to be passed to the card under the card's direction. To minimise processing within the ICC, such a list is generally not TLV encoded but is a single constructed field built by concatenating several data elements together. Since the elements of the constructed field are not TLV encoded, it is imperative that the ICC knows the format of this field when the data is received. This is achieved by including a Data Object List (DOL) in the ICC, specifying the format of the data to be included in the constructed field. This appendix describes the rules for constructing a field using a DOL supplied by the card.

A DOL is a concatenated list of entries, with each entry representing a single data element to be included in the constructed field. The format of each entry is a one- or two-byte tag identifying the desired data object, followed by a one-byte length, representing the number of bytes the field shall occupy in the command data. Only tags representing primitive data objects defined in Annex B shall be used in DOLs.

The terminal shall complete the following steps to build the constructed field:

- Read the DOL from the ICC.
- Concatenate together all data elements listed in the DOL. The following rules apply to the creation of this concatenation:

1. If the tag of any data object identified in the DOL is unknown to the terminal or represents a constructed data object, the terminal shall provide a data element with the length specified and a value of all hexadecimal zeroes.
2. If a data object is in the list and is meaningful to the terminal but represents optional static data that is absent from the ICC, the portion of the command field representing the data object shall be filled with hexadecimal zeroes.
3. If the length specified in the DOL entry is less than the length of the actual data object, the leftmost bytes of the data element shall be truncated if the data object has numeric (n) format, or the rightmost bytes of the data shall be truncated for any other format. If the length specified is greater than the actual data, the actual data shall be padded with leading hexadecimal zeroes if the data has numeric format, with hexadecimal FF's if the data has compressed numeric (cn) format, or with trailing hexadecimal zeroes for any other format.
4. If a data object is in the list and is meaningful to the terminal but represents data that is not applicable to the current transaction, the portion of the command field representing the data object shall be filled with hexadecimal zeroes.

The completed list of data elements shall be concatenated in the sequence in which the corresponding data objects appear in the DOL.

1. These rules apply to the Token TLVPLUSDOL FE DA and are intended as application specific enhancement, in order to facilitate implementation of TLV data object list handling conforming to EMV'96 and EMV2000, where applicable.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Appendix F: System Overview

How This Appendix is Organised

This appendix is divided into the following chapters:

Chapter 1 provides a brief overview of the Open Terminal Architecture.

Chapter 2 describes the elements of the Open Terminal Architecture, and how they relate to each other and to the external environment.

Chapter 3 describes the "Virtual Machine" architecture upon which OTA is based.

Chapter 4 describes ways in which the Open Terminal Architecture protects the terminal software and data.

Chapter 5 describes the software development environments provided to support OTA terminals and applications.

Chapter 6 summarises the principal points covered in this appendix.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.1 Introduction

The purpose of this appendix is to describe the general architecture underlying the Europay Open Terminal Architecture (OTA) and to give a brief overview of how it appears in a terminal environment. A complete description of the Open Terminal Architecture is given in several volumes comprising Open Terminal Architecture Technical Specifications (Virtual Machine Specification, Programming Interface Specifications, Library Specification, and Terminal-Resident Services Specification).

F.1.1 Purpose of the Open Terminal Architecture

Modern payment applications are moving to "Integrated Circuit Card" (ICC) technology. ICCs can significantly improve security of payment transactions by being able to manage encrypted account data off-line, participate actively in the transaction validation process, and being intrinsically extremely difficult to violate or reproduce.

Use of this new technology, however, will necessitate altering the firmware in all terminals that will use the ICCs. In order to facilitate this transition, Europay is designing a standardised software system that will be compact, efficient and easy to maintain and enhance for terminal owners and operators. This system is called the Europay "Open Terminal Architecture."

OTA defines a single standard software representation of a "virtual machine," whose behaviour is common across all terminal types, described in detail in the OTA Virtual Machine Specification. This virtual machine is implemented on particular terminal, and 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 using standard kernel functions may be developed and certified once; thereafter, they will run on any conforming terminal of the appropriate type (for example, ATM or POS) without change. Therefore, a significant consequence of the OTA technology is a simplified and uniform set of test and certification procedures for all terminal functions.

To provide a common means of distributing programs in a compact, standard, machine independent form, the OTA uses "tokens" — one- or two-byte sequences that represent virtual machine instructions. A token compiler converts source code to an intermediate string of tokens that is extremely compact (and therefore easy to transmit on a network or read from a merchant or maintenance ICC) and also easy for even simple processors to translate into fast-running programs.

To summarise, the Open Terminal Architecture provides the following major benefits:

- **A standard Virtual Machine**, which may be easily implemented on different CPU and terminal types and which allows applications to be developed that will run on any terminal with an OTA compliant kernel. The VM implementation's lifetime is expected to match that of the terminal (7-10 years).
- **One-time development and certification of applications**, which may then be loaded into any terminal containing a compliant OTA VM (no more multiple software developments to implement the same change).
- **Terminal VM certification independent of applications**, so certification only needs to be done once for each terminal type. A terminal type is defined as a specific configuration of terminal CPU and I/O functions.
- **Compact modules** for libraries and applications, to minimise download time and applica-

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

tion storage requirements.

- **CPU-independent program code** which can be interpreted during a transaction to enhance a terminal's behaviour. CPU independence means that only one certification and validation is needed for this code.

Merchant and maintenance cards may also contain code to enhance the transaction processing, thereby providing new opportunities for payment products and services. As far as cardholder cards are concerned, this feature is only intended to be used as a selection mechanism for functions already existing in the terminal. This is controlled both by the acquirer and the scheme rules.

F.1.2 Architectural Foundations

The basic architecture of the Open Terminal Architecture is drawn from several major software technologies developed in Europe and America to address the goals of efficient, machine-independent software for embedded systems. Two of these are:

1. SENDIT (ESPRIT Project EP9152, May 1996) was developed by a consortium of five European companies to produce a set of debugging and systems integration tools for heterogeneous hard real-time networks. SENDIT features a CPU-independent token set representing the instruction set of a two-stack Virtual Machine. This architecture yields very high code density to minimise downloading across networks as well as providing hardware independence without excessive performance costs in real-time applications.
2. Open Firmware (IEEE 1275) was developed by Sun Microsystems and adopted by IBM, Motorola and Apple for the "plug-and-play" firmware on PCI-bus systems. It also features a very dense CPU-independent token set, so that peripheral controller boards can offer diagnostics and drivers in on-board PROMs that can be used by any host regardless of its native CPU.

Both of these technologies are based on the Forth programming language, which was designed expressly for embedded and control applications.

The token set and underlying logic of OTA is also based on Forth. Forth is an "extensible" language, meaning that it is designed so that its instruction set may be extended with application-specific commands. For OTA it has been extended with an extensive vocabulary of commands to facilitate development of payment applications.

Readers who are not familiar with Forth or stack machines are encouraged to read some introductory Forth material. Two recommended books are Koopman, *Stack Computers, The New Wave* (Ellis Horwood, Ltd. 1989) which provides a theoretical discussion of stack machines, and Brodie, *Starting Forth* (Prentice Hall, 1987, ISBN 0138430799), which is a practical introduction to the Forth language.

Forth was chosen because, of all standard interpretive-type languages, it provides the most compact and efficient means of representing both terminal programs and the code that may reside on an ICC, e.g. merchant or maintenance card. Compactness in terminal programs translates directly into reduced transmission time and cost for terminal updates.

Although the architecture of the OTA VM has been optimised for Forth, VM implementations have been developed in both Forth and C. The C implementations are appropriate for terminals without severe memory restrictions; the Forth implementations are compact enough for terminals with 8-bit CPUs and relatively small memories.

Token compilers are available in C or Forth for development of application programs, termi-

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

nal control programs, and associated libraries.

F.1.3 Software Integrity and Validation

Since terminal software fully utilising ICC capabilities is inevitably more complex and more powerful than current terminal software, software integrity and validation must be carefully considered. Validation of program compilation is done during development and before certification. Validation of program execution is done primarily during development but also to some extent during transaction processing in a terminal. Extensive checking is done before certification to ensure that applications do not execute illegal memory accesses and that applications will not interact with each other. Applications and libraries must explicitly declare all functions that they intend to import or will provide for export, and only those inter-program dependencies are allowed when an application is compiled.

F.1.4 Security Mechanisms in OTA

Since ICCs have significantly more capability than conventional magnetic stripe cards, and can even be programmed to enhance the transaction processing, the security of systems using ICCs is of extreme importance. There are three different types of terminal security applicable to OTA. They are independent of each other.

1. **Communications Security** is involved with communications to and from a terminal. It consists of message encryption and decryption and message authorisation, to assure that code or data downloaded into a terminal comes from an authorised source and is valid (does not contain transmission errors).
2. **Access Security** is involved with a cardholder's access rights to an application. It ensures that a terminal user only executes functions that are valid for both the card and the terminal. Code on an ICC may be able to enhance an application's behaviour, *but only* if access rights are satisfied, i.e. both acquirer and issuer authorisations are successful.
3. **Program Security** concerns attempts to tamper with the terminal software directly. The previous categories address the issue of maintaining security in a running terminal, whereas this category is concerned with attempts to modify or completely replace the software.

Mechanisms for implementing these terminal security functions are discussed in more detail in Section F.4 (page 142). Program validation and certification, which also affects security, is discussed in Section F.5.5 (page 148).

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.2 System Components

The purpose of an OTA system is to provide software to run in terminals used in payment applications. Conceptually, there are two hardware environments, and several classes of software. The hardware environments include the *development system*, which is based on a simple PC, and a *target* which is some form of payment terminal. The software includes *development software*, which runs on the PC; *Virtual Machine implementation*, which include all platform-specific software in a terminal and other mandatory standard functions; *libraries*, which provide general functions to support terminal programs and payment applications; *applications*, which are the functions specific to a particular payment product, and *terminal programs*, which perform general non-payment terminal functions and include high-level mechanisms for selecting and executing transactions and associated applications.

This section briefly discusses these components and how they are used to develop and run payment programs. They are considered in more detail in later sections.

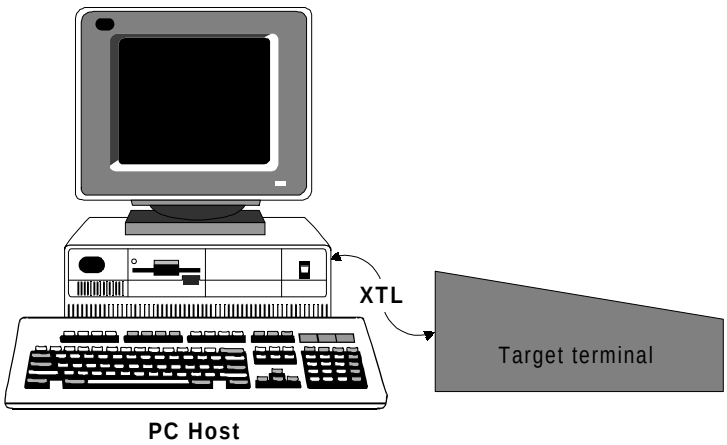


Figure 7: OTA development environment

A typical OTA development environment for a terminal. A small program to support the terminal end of the Cross-Target Link (XTL) protocol is usually placed in the terminal's kernel.

F.2.1 Development Environment

An OTA development system is used during the production of terminal software, either CPU-dependent kernel software or CPU-independent library or application software. OTA kernel development requires a target terminal to be connected, as the kernel is cross-compiled on the PC host and downloaded to the terminal across the Cross-Target Link (or XTL). OTA libraries, terminal programs, and applications are also developed on the PC host. Because they are high-level code, they may also be executed on the host for preliminary testing, using the host's version of the standard kernel, and thus a target is not needed. For final testing using the terminal's own kernel and I/O, these programs are tokenised (see discussion below) and sent across the XTL to the target terminal.

As an example, one prototype development system consists of a PC running Windows 3.11, Windows NT or Windows 95 and an optional XTL to a target terminal (see Figure 1). Development software on the PC is based on ProForth for Windows, a product of MicroProcessor Engineering Ltd., and direct links may be established with any Windows programmer's editor. Together, these products provide a convenient, seamless programming environment.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.2.2 Terminal Target Environments

The target system may be any one of a large variety of payment terminals. Actual products range from small, hand-held devices with simple 8-bit microprocessors such as the 8031/51 family to 32-bit computers running operating systems such as Windows NT or UNIX. In order to simplify the production, certification and maintenance of software on such a wide variety of targets, OTA terminal code is based on a single *virtual machine* which is emulated on the actual devices. Prototype coding and testing has shown that this approach is feasible and provides good run-time performance, even on an 8051 CPU.

Each target contains a *Virtual Machine implementation* consisting of a standardised set of CPU-specific functions optimised for that specific platform. This kernel supports standard *libraries* and *terminal programs* and *applications* written in high-level code for the Virtual Machine and will therefore run on any terminal with a standard OTA kernel.

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

1. *Terminal Resident Services* (TRS) include the terminal's kernel and I/O support, all non-payment functions, libraries supporting these functions, program loading functions, and the main loop defining the terminal's behaviour. TRS are usually provided by a terminal vendor. The terminal vendor software may include features in addition to the TRS which are outside the scope of the OTA specification .
2. *Terminal Selected Services* (TSS) include all payment service functions, also known as applications, and libraries supporting these functions. TSS are provided by the individual payment systems, either directly or through the acquirer who owns the terminal, and are resident on the terminal. The TRS main program loop selects and calls TSS functions as needed by a particular transaction.
3. *Card Selected Services* (CSS) include functions supporting terminal transactions, usually payment service options that are used as part of a TSS application. CSS are resident on an ICC and downloaded to a terminal as requested, within acquirer and payment system rules. For terminals with two ICC readers (one for the cardholder and one for the merchant card) there may be two independent sets of CSS. As mentioned earlier, code on an ICC is limited to providing access to pre-existing library functions on the terminal.

F.2.3 Virtual Machine

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 Section F.3 for details). The OTA kernel for each particular CPU type is 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 implementation for a particular terminal contains all functions whose implementation depends upon that particular platform (CPU and OS). It includes a selected subset of ANS Forth commands, plus a number of specialised OTA functions, such as terminal I/O support and token loader/interpreter support. As on some terminals the VM may reside in PROM, the OTA kernel is intended to be installed once, and subsequently changed infrequently during the lifetime of the terminal. Therefore its functions are carefully designed to be very general in nature and as complete as possible, in order to support a wide range of present and future terminal programs and applications.

A VM implementation is normally provided by the manufacturer of the terminal in which it

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

resides, although EPI and its partners can develop the kernel and applications on behalf of the terminal manufacturer or member. It is developed and certified according to the OTA Virtual Machine Specification. Standard functions not appropriate to a particular terminal type (e.g. cash dispenser function on a POS terminal) may be coded for that terminal as null functions, so that every kernel has an identical set of functions and the testing and certification process is simplified. These null functions add very little to the system overhead and complexity, and their advantage far outweighs their cost.

F.2.4 Libraries

OTA Libraries contain higher-level functions that support common features of terminal programs, such as language selection, and common features of applications, such as PIN verification. A terminal may contain several libraries, some accessible to all applications, and some restricted to particular applications or payment systems. Libraries are written and tokenised for the virtual machine, using functions provided in the kernel, and therefore can be run on any terminal.

F.2.5 Terminal Program

A terminal program is part of the Terminal Resident Services and consists of the high-level "personality" characteristic of this terminal type (POS, ATM, etc.). This includes the functions common to all transactions (e.g., card initialisation and language selection) as well as the user interface required to select an application and process a transaction. The terminal program at the highest level is typically triggered by a card insertion. Like libraries, a terminal program is written for the virtual machine, based on both kernel and library functions, and supplied in token form. It can therefore be run on any terminal of the appropriate type (e.g., ATM), and is easily changed by downloading over a network at any time.

F.2.6 Applications

A terminal transaction will select an application as part of its processing flow. Applications fall into three general areas: cashless "purses" (Clip), debit cards (Pay Now) and credit cards (Pay Later); applications will generally vary in their method of processing a given transaction. Versions of these applications may be provided by different payment systems and further customised by individual issuers or acquirers. Applications are supplied in token form via the communications path, and (if security considerations permit) may be enhanced by token programs on an ICC.

F.2.7 The Token Compiler and Token Loader/Interpreter

Libraries, applications, and terminal programs are written in high-level code for the virtual machine. The OTA development system includes a special compiler for this virtual machine, whose output consists of *tokens*. Tokens may be thought of as machine language instructions for the virtual machine. Tokens are either one or two bytes in length, and therefore represent the program in a form that is both CPU-independent and extremely compact (far more so, for example, than compressed source text).

Each kernel contains a *Token Loader/Interpreter* (TLI), which processes a stream of tokens into an executable form. Once the kernel is installed in a terminal, the libraries, applications, and terminal programs can be downloaded into the terminal in a variety of ways (direct connection to an OTA development host, acquirer hard-wired network, modem and dial-up telephone line, ICC, etc.). Program modifications and entire new applications are down-loadable in the same manner whenever needed. The kernel is designed to be so general purpose in

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

nature that a wide range of present and future terminal programs and applications can be accommodated without kernel modifications.

F.2.8 ICC Functions

One function of ICCs is to improve transaction security by incorporating and managing encrypted data and participating actively in the transaction validation process. It is a natural feature of the Open Terminal Architecture to go beyond these functions, and provide for ICCs that also contain restricted program code to enhance a terminal's transaction processing, thereby providing new opportunities for payment products and services. A cardholder card is entitled to request the execution of predefined library functions resident on the terminal during the execution of the application; a merchant or maintenance card could bring more extensive upgrades of applications under the usual security restrictions.

Under this concept, the payment systems (who specify the applications) may define up to 64 modifiable procedures (referred to as "sockets") which may be inserted by the application programmer (and hence under acquirer control and under payment system supervision) to act as placeholders. Sockets hold execution vectors, also known as function pointers, which are initialised to default behaviours. If no further action is taken by the terminal program the default behaviour of these procedures will be to do nothing when they are executed. A terminal application may also by itself define one or more functions to "plug" into the appropriate socket(s) at run-time to provide a set of initial non-zero behaviours. For example, the EMV function Cardholder Verification Method (CVM) could be defined as a socket, with the terminal application also plugging a default of the EMV standard behaviour into it. Code downloaded from the ICC can request enhanced behaviours and then plug these behaviours into any or all of the existing application sockets, but only if the acquirer application program permits it. For security, these enhanced behaviours must be predefined in a terminal library accessible during the transaction. To continue this example, an ICC might plug an EMV-compliant superset into the CVM socket that provides a more elaborate verification method than in the standard EMV Cardholder Verification Rules (CVR). At the conclusion of a cardholder session the sockets are restored to their original application default behaviours. Socket access security is discussed further in Section F.4.1.2.

Sockets are intended to be placed at various points in existing terminal applications, or even in the terminal program itself. Sockets may be assigned new plug behaviours either by terminal-resident software or by software on an ICC. The basic restriction for ICC plugs is that ICC code which modifies a socket's behaviour cannot be placed in the program flow before ICC authentication, because only after this point is the card data (and code) known to be valid.

However, it is possible to define a skeleton terminal program which after the data authentication step consists only of sockets and basic program flow from socket to socket, and therefore is very flexible as to what the ICC may command it to do.

As far as security is concerned, the presumption is that if an ICC passes the decryption and data authentication tests, then whatever functions and plug behaviours are on the card have been certified and are syntactically valid. The terminal decides to allow or disallow the card's proposed actions only as controlled by the terminal access security functions (see Section F.4.1.1)

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

F.3 OTA System Features

The OTA token set provides program portability in systems with multiple CPU types by passing source code programs of various types through a compiler whose output is a string of one- or two-byte tokens. Target terminals then interpret this token stream. Figure 8 illustrates this process.

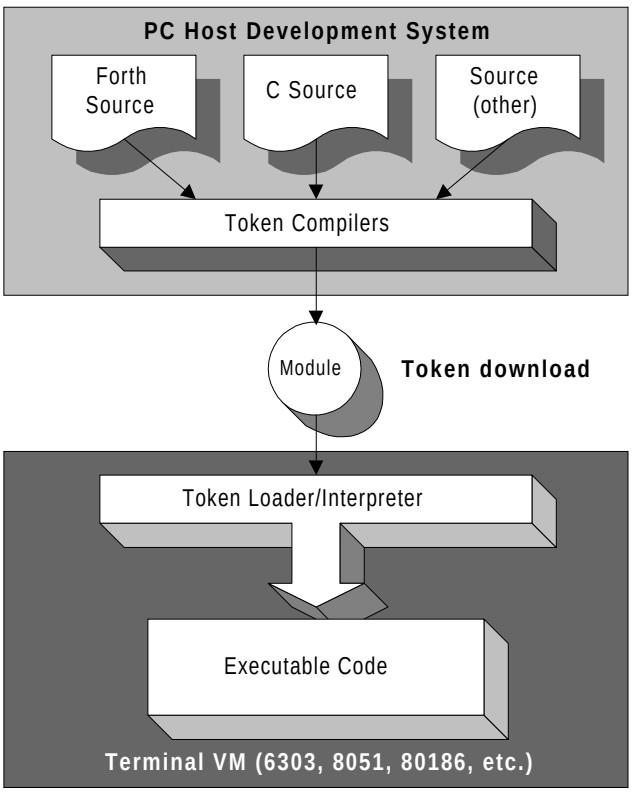


Figure 8: Tokens in the OTA concept

Tokens may be generated from a variety of source code formats, downloaded to a terminal, and then converted into executable code for that terminal by any of several different methods.

The OTA token set covers two main areas. The first is the instruction set of a theoretical processor (virtual machine), and thus provides the instructions necessary for the efficient execution of programs. The second allows what are normally called "operating system functions" to be defined. In the embedded systems for which OTA is targeted, system functions cover not only application-specific functions such as communications, TLVs, and EMV library functions, but also access between program modules and the associated access control mechanism.

In order to transport programs across multiple CPUs and operating systems there are two common industry approaches:

- Define a *common source language*, and recompile for each target CPU and Operating System. This approach is widely used in UNIX systems, but requires full source code to be provided, which has obvious commercial problems, as well as leading to version control problems. Also, terminals using simple microprocessors may not have the resources (RAM, processing speed) to support this approach.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Define a *common binary language*, or set of tokens, which is interpreted or converted to an executable form by the target itself. This approach has a long history, mostly in language specific forms such as UCSD PCode, Open Firmware FCode, ParcPlace Smalltalk, ISE Eiffel, Microsoft Quickbasic and Visual C.

OTA uses the second approach of defining a virtual machine (a theoretical processor) for which all programs are compiled. Programs for this processor consist of a list of tokens that are transported by modem or ICC to the target terminal and are processed by the terminal into an executable form.

The OTA token set has been optimised for use on small terminals with ease of compilation, ease of interpretation, and good code density.

F.3.1 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 ANS standard X3.215-1994 for the ANS Forth standard). This architecture derived from Forth has been further modified for portability, code density, ease of compilation, and for use with other programming languages. For example, it contains provisions for local variables which are used in C. Thus, OTA token compilers can be written not only for Forth but also C and other languages, allowing for application and library development in various languages.

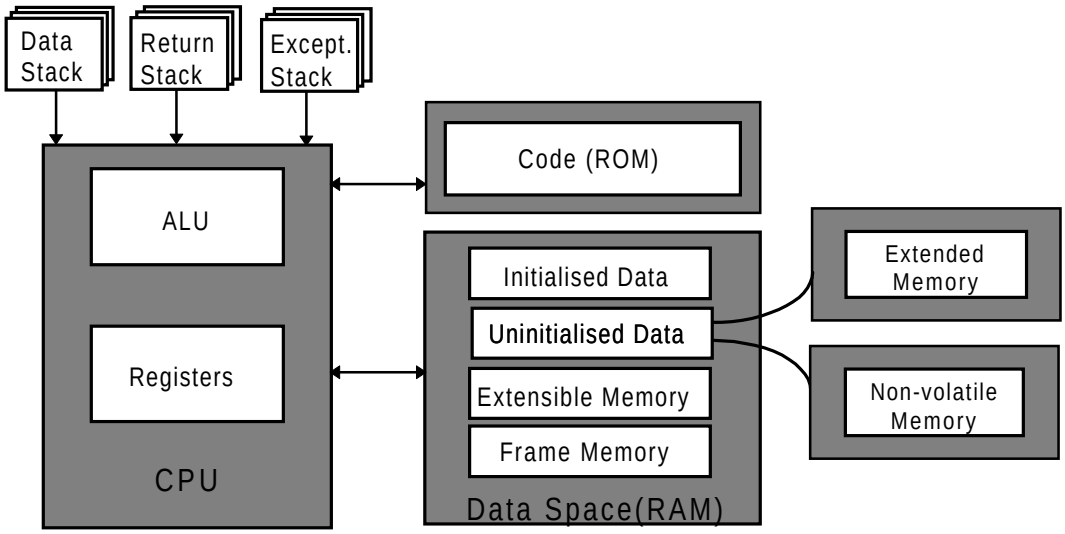


Figure 9: The OTA Virtual Machine

The data, return and exception stacks are not in accessible memory space. Code memory is not available to token programs and so can be considered to be in ROM. Extended memory and non-volatile memory are available only through tokens.

F.3.1.1 Registers

The Virtual Machine's registers are defined in Table 47.

Register	Name	Description
TP	Token Pointer	Points to the next token to be executed

Table 47: Virtual Machine registers

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Register	Name	Description
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 main memory.
FEP	Frame End Pointer	Points to the frame end in main memory.
QRR	Quote Return Register	Contains the return address for a “quote” optimisation.

Table 47: Virtual Machine registers

The data and return stacks are not in the VM address space and cannot be addressed directly for security reasons; they are only accessible via stack operations.

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 quote return register is used by an optimiser that processes compiled token modules to further reduce their size by eliminating repeated sequences. Quoting resembles compression algorithms.

How VM registers are mapped onto the target CPU architecture is up to the implementers for that CPU. Although these registers are conceptually present, for security very little access is provided to them by the token set.

F.3.1.2 Virtual Machine Size and Cells

The OTA virtual machine is defined as a byte-addressed, 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.

The entire OTA Virtual Machine (including I/O drivers, token loader/interpreter, etc.) occupies approximately 32 Kbytes for 8- and 16-bit processors, and under 64K on 32-bit processors.

F.3.1.3 Memory

OTA defines a single address space for programs. This address space is accessible for data storage only. Programs may not assume that executable code is in this address space. Depending on the actual processor and VM implementation strategy, the executable code may be in a different address space, or may be under the control of a memory management unit. In any case, programs are not permitted to access their own program memory directly, and any attempt to do so will be flagged during the program certification procedure.

In addition to statically allocated memory for initialised and uninitialised variables and buffers, the VM also provides a dynamically allocated pool of memory from which programs may request space for temporary storage. This storage is automatically released when the program finishes executing.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.3.1.4 Exception handling

Because it is so tightly bound to register access, OTA provides only one exception handling mechanism via the tokens **`CATCH`** and **`THROW`**. These tokens are derived from the Lisp exception handling mechanism, and appear in the ANS Forth Standard as **`CATCH`** and **`THROW`**, and in the ANS C Standard as **`try`** and **`except`** (see those documents for complete descriptions).

F.3.1.5 Stacks

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

The *return stack* contains return addresses, and is also used for temporary storage (often of structure pointers). Because of the variety of implementation techniques and processor architectures, application programs may *not* assume that accessing the top of the return stack will give the return address of the current procedure. Application programs may only retrieve from the return stack what they have put on it inside the current procedure.

F.3.1.6 Frame Pointer usage

The *frame pointer* mechanism allows OTA to satisfy the requirements of languages such as C that permit local variables to be defined at run time. It also satisfies the requirements of languages like Pascal that permit procedures to be defined within procedures.

F.3.2 I/O and System Functions

Each operating system function or I/O device, including 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 (keypads, displays, printers, ICC readers, etc.).

General I/O facilities are provided by functions taking the device number as an input parameter. These functions return specified I/O result codes indicating success or specific failure modes. This generic method of I/O access permits kernel implementers to hide implementation details and proprietary technology behind a defined interface. Kernel implementers can thus make their core technology available to application developers in a controlled way without exposing any implementation details.

F.3.3 OTA-Specific Features

The Virtual Machine contains a number of powerful extensions that greatly simplify the task of writing application programs and libraries. The most commonly used of these functions are accessed by primary (one-byte) tokens, and all are highly optimised for maximum performance.

F.3.3.1 Database Services

The Virtual Machine provides a mechanism for handling databases that conceals implementation details (handled by the VM as a “server”) from external programs (“clients”). A database contains a variable number of records which, in turn, contain named fields of various types (numeric, strings, etc.). Several different database types are defined, with custom

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

access methods implemented using object-oriented techniques.

Client programs may access databases only through special tokens that provide a high degree of security; databases are only "visible" to programs that are explicitly authorised to use them under defined circumstances. And at any one time, only one "current" record in each database is accessible.

Databases are used for most data storage, including user messages, transaction logs, temporary transaction values, etc.

F.3.3.2 Language and Message Handling

A special group of tokens are provided for language selection, message selection and message display. EMV requires space for up to 255 message in each terminal language table, and provides for up to 64 "Issuer" messages to be supplied on a temporary basis by an ICC.

Standard mechanisms are provided for downloading and storing messages, as well as using them during the course of a transaction.

As with all data managed by the OTA VM, message tables are not directly accessible to programs.

F.3.3.3 TLV Services

EMV specifies that the standard structure for data interchange is the TLV data formats described in ISO/IEC 8825 (1990). EMV also defines the specific TLV data structures defined in OTA.

The VM instruction set includes a rich set of functions for managing TLV data: reading and writing it, parsing sequences, managing TLV formats, etc. Support for Data Object Lists (DOLs) is also provided.

Each TLV that will be used must be defined by the progam using it. Internally, the VM uses a linked structure (a balanced binary tree) to allow rapid addition and removal of tag names from the global tag name list.

F.3.3.4 Hot Card List Management

Hot Card Lists present some special technical challenges, since a list may contain on the order of 10,000 entries of up to 10 bytes each. Hot Card List management is provided through a specialised group of tokens that enables a VM to implement this potentially very large special database with excellent performance. Due to the special security requirements inherent in Hot Card Lists, even the standard database tokens may not be used to access the Hot Card List; access is provided only through the special Hot Card tokens.

F.3.3.5 Cryptographic Services

High-performance data authentication algorithms are essential for good user acceptance of ICC terminals. A small library of specialised algorithms, including modulo multiplication and exponentiation, a secure hashing algorithm, and other operations are performed by highly optimised code functions within the VM.

F.3.4 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 are primarily used for primitive instructions

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

such as are commonly found in any instruction set. Multibyte tokens are referred to as *secondary* tokens, and are used for less frequently needed functions. In some cases, data type overrides allow one token to be used for several data types.

The *token compiler* compiles source code as a string of tokens formed into a "module," which may be downloaded to the target terminal. A module is an extremely compact representation of a program, because the token set has been highly optimised such that the vast majority of functions are performed with single-byte tokens. Thus, they are far more compact than binary object code (in addition to being CPU-independent) or even compressed interpretable text.

An optional optimiser performs various post-processing functions on compiled modules to eliminate repeated code strings and unused code, resulting in further module size reductions.

The extremely compact size of OTA modules offers two benefits:

1. It can significantly reduce the transmission time required for updating modules.
2. It means that a terminal can easily store modules for a number of different applications, thereby greatly increasing the flexibility of terminals.

After downloading, the terminal uses a *token loader/interpreter* (TLI) to execute the module. The TLI loads the token image into memory and adjusts it as necessary for byte order and address relocation. The result is then interpreted directly. The target image is the same size as the program image. This technique, which has been used by MS Quickbasic and UCSD PCode, is simple to implement, is suitable for small processors (e.g. 8051), and has been implemented with very low performance overhead.

F.3.5 Program and Library Management

All software on an OTA terminal above the kernel is organised as a set of separate *modules*. The fundamental characteristic of a module is that it is a collection of definitions or program functions that are passed through the token compiler as a single unit. The main Terminal Program (TRS), each application, each library, and each CSS download are examples of modules. There are two basic types of modules: an *executable* module has an entry point that is called when it is loaded; and a *library* module acts only as resources to other modules. An executable module may also provide specific resources to other modules.

Separately compiled modules need to communicate to perform overall terminal functions. Four kinds of inter-module interactions are provided:

1. An executable module (TRS or application) may call another executable module.
2. A module (either executable or library) may export functions for use by other modules. The only external access to a module's code is through its entry point (if it is executable), or through one or more specifically declared export functions.
3. A module may import pointers to functions in other modules (such as library modules). Only functions that have been declared as exported by the providing module may be imported by the requesting module.b
4. The plug and socket mechanism described in Section F.2.8 in connection with ICCs may be used for communication between modules as well. A module may provide alternative functions to be plugged into existing sockets, but only if the previously loaded module allows it. The module defining the sockets includes a application-specific routine, **PLUGGABLE**, which contains logic to determine under what conditions a subsequent module may plug a given socket.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A library or application module may also include the definition of new sockets, for later **PLUG** commands coming from, for example, an ICC. In this case the module would define a socket and then use **PLUG** to assign a default behaviour to it (often a null behaviour). If access control (**PLUGGABLE**) allows it, an ICC could later download tokens that define a new behaviour and then **PLUG** it into this same socket, overriding the default behaviour.

F.3.6 Performance

In order to present an acceptable user interface, it is necessary for even low-end terminals to perform complete transactions in only a few seconds. The EMV specifications require a terminal to be able to perform dynamic data authentication using the RSA algorithm of fairly long keys. Optimised code in the OTA kernels developed to date have demonstrated the ability to do this in roughly one second even on slow 8-bit processors. Indeed, present kernels have demonstrated sufficiently fast performance that transaction times are dominated by user interaction and ICC communication.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.4 Program Security and Integrity

Terminals fully utilising ICC capabilities necessarily include complex and powerful software, and protecting this software from tampering or fraudulent usage is extremely important. Communications security and access control, discussed in this section, ensure that programs in the terminal are protected against unauthorised access and contamination. Program security is concerned with ensuring that a previously authorised and certified program reaches the terminal and executes only in an authorised way; program integrity is concerned with ensuring that certified terminal programs are not tampered with or modified.

This section discusses program security and integrity in the terminal environment. Production of the certified programs is a separate issue, discussed in Section F.5.5.

F.4.1 Protecting Program Security

OTA is designed as a secure system, and includes many features (such as strict segregation of code and data and hiding of low-level VM functions from application programs) to enhance security. But the actual implementation of high-level security functions is the responsibility of the TRS and high-level libraries, as they must comply with acquirer and merchant requirements and protocols as well as those of the payment system and those of other bodies. The objective of the OTA VM design is to provide sufficient tools that the high-level programs may ensure a high level of security of the terminal, its programs and data.

F.4.1.1 Communications and Access Security

Communications security is invoked when data or programs are being loaded into a terminal, either by modem from an acquirer or from an ICC (maintenance for the entire application or cardholder card for predefined features), and consists of message encryption and decryption and message authentication.

The OTA kernel specification includes generalised commands in the Core Extension to initialise and perform various security algorithms such as DES (Data Encryption Standard), RSA (Rivest Shamir and Adleman algorithm), SHA (Secure Hash Algorithm), etc.

After successful decryption, methods such as authentication protocols and checksums validate the data and/or program as coming from an authorised acquirer or issuer. This ensures that the data and/or program have not been corrupted and are suitable for use in the selected terminal application.

Access Security ensures that a terminal user only executes applications that are considered valid by both the card and the terminal. By comparing the terminal applications against the AIDs (Application Identifiers) on the card, the terminal will only permit applications covered by a current contract between the merchant and the acquirer, the issuer and the cardholder, and within the payment scheme.

F.4.1.2 Commercial Card Services Security

In addition to general security for access to applications, there is the possibility of issuer- or acquirer-specific branches within a transaction, and the need to restrict access to these branches based on issuer ID or other transaction-specific data. This is done in OTA by providing access rights control to socketed functions.

Transaction enhancements by code on an ICC is possible in the Interpreter Concept, and is

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

done through special functions called “sockets,” as described in Section F.3.5. Sockets allow the creation of a definition whose behaviour can be changed at execution time, using the words **PLUG** and **INTO**. These are implemented as “deferred execution” words. Each socket has an associated permission flag which may be set by the acquirer’s terminal program, and which permits or denies later use of a **PLUG** command. For security, the array of socket permission flags is in a separate program module and is only settable through an acquirer-defined logic function (the **PLUGGABLE** routine); thus the execution of code downloaded from an ICC is strictly controlled by the acquirer.

As an example of an acquirer-specific function, suppose the basic transaction code includes the phrase **27 SOCKET LOYALTY** which defines **LOYALTY** and makes it available for later execution. The acquirer’s transaction program code further defines code that sets the permission flag for this socket only if the issuer is the same as the acquirer and the transaction amount exceeds a certain minimum.

During the transaction there would be a command which reads in arbitrary code from the ICC. Part of the ICC code could define a **REWARD** routine which updates the user’s frequent flier miles, and then attempt to execute the phrase **PLUG REWARD INTO LOYALTY**. This phrase connects the execution of **REWARD** with the execution of **LOYALTY**. If the **LOYALTY** socket permission flag is set according to the above logic, the **PLUG** will take place; otherwise **LOYALTY** will retain its default behaviour, likely a no-op. Then when the application code executes its **LOYALTY** function later, it will allow the ICC-defined **REWARD** only according to the acquirer-defined rules.

Part of the specification for using sockets is that all functions vectored to use a particular one must have exactly the same stack effect. This ensures program continuity no matter which vectored version of the function is executed.

F.4.2 Protecting Program Integrity

Another security issue is protection of the overall terminal integrity. Some aspects of this involve mechanical or electronic issues which are outside the scope of the OTA specifications. Here we will briefly consider the possibility of unauthorised modification or complete replacement of the terminal program. It seems unlikely that this could be done because of the high level of communications security and encryption, but if one assumes that the communications security could be breached, then the integrity of the terminal software itself becomes of interest. The Interpreter Concept for terminal software has additional benefits in this area.

In order to replace terminal software with a fraudulent version, one would presumably proceed by disassembling or decompiling a running authorised version. In OTA, depending on the VM implementation strategy used, terminals can be running programs which are structured as “threaded code.” In threaded code, all routines except the most elementary ones consist basically of strings of addresses or calls that thread a mix of other high-level and elementary functions together to perform complex actions. Threaded code, unlike conventionally compiled programs, is extremely difficult to disassemble and commercial disassemblers are not available for it. Further, because of the way that cross-compiling is done in OTA the names of the routines do not even exist in the terminal, as discussed below in Section F.5.4. This means that in following a thread, one has no clues as to the function of a particular sequence. Analysing threaded code calls for the far more difficult task of program tracing (dynamic analysis) rather than code inspection (static analysis).

Additional program security is available in areas such as data hiding and data integrity. An example of data hiding is simple encryption (such as XOR against a sequence available at run

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

time), which is sufficient to keep data from being evident after a disassembly. Data integrity can be checked periodically with checksums or more complex techniques.

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.5 OTA Software Development Tools

This section discusses the process of developing two major components of OTA software: terminal kernels and terminal programs. It is important to note that the following discussion is concentrated on terminals using embedded microprocessors; more powerful devices, such as those based on Windows NT or other operating systems, can support an entire development environment in the device itself. Procedures will thus be even simpler.

F.5.1 Hardware and Software Used

Two basic development kits are offered for the two main aspects of OTA software development:

1. The Kernel Development Kit (KDK) is designed for terminal vendors or others building the platform-dependent implementation of the OTA Virtual Machine for a particular terminal platform. It contains a cross-compiler and assembler, source for a generic kernel for a particular CPU type, and many programming aids supported in an Integrated Development Environment (IDE). It may also include simplified “terminal” hardware to provide a running example for developers.
2. The Application Development Kit (ADK) is designed for anyone desiring to write and test token programs, whether a TRS for a new terminal, or libraries and even payment applications. It includes a compiler whose object is OTA tokens and a test environment that simulates a payment terminal.

Both of these development systems are based on a PC running Windows. They are discussed in more detail in the following sections.

F.5.2 OTA Kernel Development

OTA kernels have been developed in both Forth and C for a range of processors and terminal types. Adapting such a kernel for a particular terminal requires only providing it access to the terminal’s I/O, and adding OTA-specific Core Extension words. C development uses standard C programming tools.

Figure 4 shows a Forth development set-up, using the OTA development system running on a PC compatible computer linked by a communications port to the microcontroller. Special software, usually only a few hundred bytes in size, resides on the target to support the Cross-Target Link (XTL). Using the XTL to provide communication to the host, the chip is completely under the control of the PC. The Link Control routine in the host communicates with the XTL on the target. It moves compiled code into the target, and also obtains information from the target for the programmer.

Forth kernels may include native I/O drivers, for top performance, or may call pre-existing drivers written in C, assembly language, or other languages.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

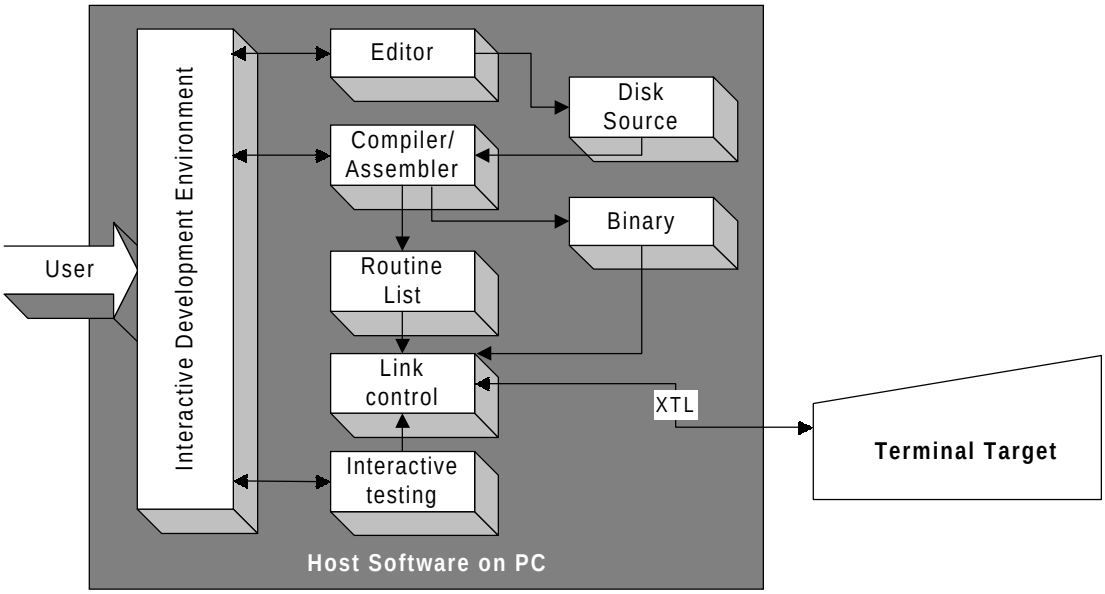


Figure 10: Kernel development tools

The process of compiling and down-loading code to the chip takes only a fraction of a second; even a full kernel takes only a few seconds, and the download is automatic and concurrent. This development approach is optimised for interactive coding and testing, since there is virtually no time or procedure involved.

Source is normally compiled from files, although it is possible to simply type new definitions on the keyboard, and they will be automatically downloaded to the target where they are available for immediate testing. To test a routine, you just type its parameters and name; the routine will be immediately executed on the target and the results (if any) passed back to the host.

As increasing amounts of code are developed and tested, the time to re-compile the program will increase. At any time, it is possible to save a binary module for down-loading to save this time.

F.5.3 Terminal Program Development

OTA library routines and terminal programs are written in high-level code, and will therefore run equivalently on any fully compliant OTA VM implementation. The development environment designed for this purpose includes two main components: aToken Compiler and a PC-based Token Interactive Development Environment, or TIDE. TIDE includes a VM implementation under Windows, coupled with tools interactively compiling and debugging token programs. TIDE is capable of emulating most terminal parameters (screen size, etc.) in order to provide a realistic test platform for token programs.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

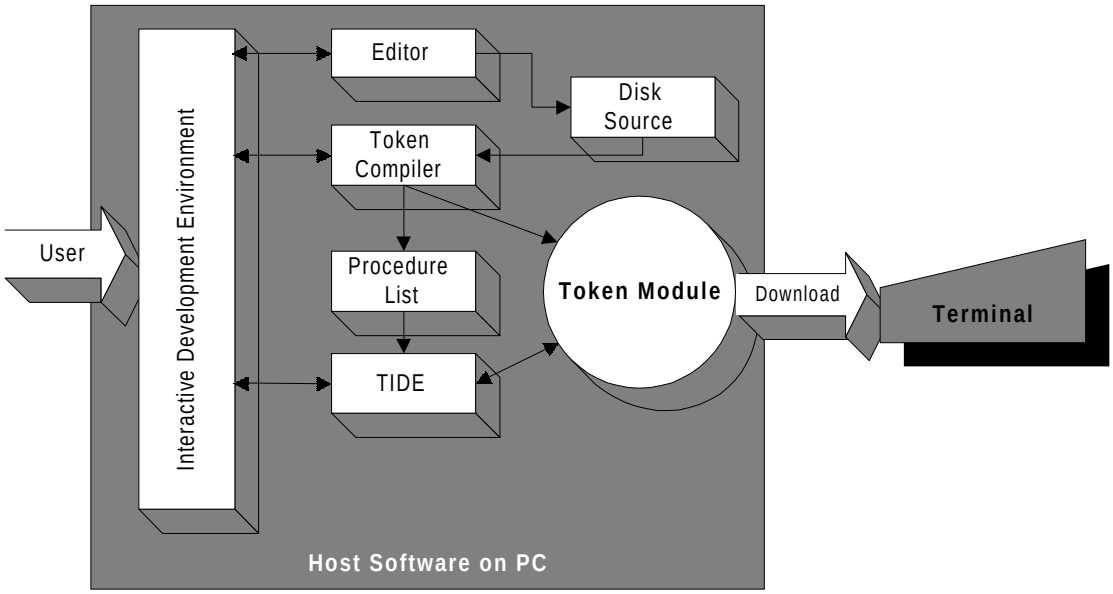


Figure 11: Application development tools

Here is a typical Forth definition for an EMV function:

```
: ACCOUNT_BALANCE ( -- )
BALANCES INITIALIZE
ADD_RECORD
TLV 81 @TLV
CONVERT_CURRENCY
TO BALANCE
;
```

This code occupies 100 or more bytes of ASCII text depending on the way it is formatted. When converted to tokens it takes only 21 bytes.

A Token Optimiser is also available. It provides post-processing on previously compiled token modules, regardless of whether the original source was in Forth or C. Several optimisation strategies are employeed, including stripping unused code paths, gathering frequently-used token sequences into “quotations” that can be referenced in just a couple of tokens, and other strategies. The Optimiser can often reduce module size by as much as 50%.

F.5.4 Development System and Target Memory Usage

True symbolic access to code exists only in a development system host. The OTA development environments are optimised for cross-compiling to the Virtual Machine, and provide interactive test capabilities oriented towards cross-development. A consequence of this is that there is a strict separation between symbol space, code space, and data space. Symbol space exists only on the host. Code space exists on the target, and may contain either target machine code or tokens, depending upon whether one is developing a kernel or terminal program. Data space also exists only on the target, and is available only in a virtual sense in the development system.

A compiled program consists of the code space prepared during the compilation process plus a table for initialising that portion of data space which requires it. It may be downloaded into a target to run. The Kernel Development Kit contains the Cross-Target Link to perform the

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

downloading and testing; target support for the XTL can be removed from final programs.

The 32-bit Virtual Machine accesses memory in two areas, code space and data space All code accesses are internal to the VM only, and are not available to programs; the memory fetch and store operators access data space only. Translated program code only exists in code space. No terminal software (libraries or other functions external to the kernel) can make any assumptions regarding the nature or content of code space, or attempt to modify code space in any way. This restriction, plus the complete absence of a symbol table, adds significantly to program security, as discussed in Section F.4.2.

F.5.5 Program Validation

Program Integrity and Validation has two separate phases: program validation at development and certification time, and protection of program integrity in the terminal. Program integrity in the terminal environment is discussed in Section F.4. With proper certification procedures, much of the program validation can and should occur in the development system environment where many more resources are available.

Program validation is a function internal to the kernel compiler and token compiler. Program validation is invoked both when functions are being compiled and when they are being executed during program validation and testing, and is concerned with verifying that the selected program operates correctly. It does not address the question of user authorisation, which is a separate area of security. Since compiling and execution can be intermixed during development, these two steps are not separate in time, but are different in what they can accomplish.

There are two categories of conditions: “errors” (detectable at compile time, typically syntax errors); and “exceptions” (detectable only at execution time, typically data-dependent errors). A list of recognised exceptions is provided in Appendix C of the *OTA Specification*, Vol. 1.

F.5.5.1 Compile-Time Integrity

An OTA compiler for a particular language will take source code in that particular language and translate it into an OTA module.

The amount of compile-time integrity checking performed will depend on both the compiler and the language. There is a choice of two compilers provided in the initial OTA application development kit, one for Forth and one for C. Both these compilers support specialised versions of their respective languages for OTA development as specified in Open Terminal Architecture (OTA) Volume 2: Forth Programming Language Interface' for Forth and 'Open Terminal Architecture (OTA) Volume 3: C Programming Language Interface' for C.

Each compiler will ensure that the source is correct with respect to the rules of the particular language before translating it to an OTA module. The language may also support additional checking, for example the C language is typed and so the compiler may perform type checking, whereas Forth is untyped and may not do so.

An alternative approach to OTA application development is to use a compiler for a real target and substitute where necessary real target libraries for OTA module libraries. In this scenario the same compiler integrity checks described above would be performed.

An OTA module can be further checked to ensure that is correctly formed using the OTA module checker. This utilty ensures that the structure of the OTA module is correct and that no range errors exist, for example there are no invalid relocation types in the relocation section or that a control flow token with the code section does not target a location outside of the code section. The format of an OTA module and a description of the OTA virtual machine

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

can be found in *Open Terminal Architecture Specification*, Volume 1: Virtual Machine Specification.

In order to properly address the issues involved in compile-time integrity, it is important to consider the entire program development process as described above. Here we briefly indicate the important steps for a payment system application:

1. An application program is written on an OTA development system.
2. The program is compiled, tested and debugged as far as possible on the development system, using null definitions or other simulations for hardware not present. The extreme modularity of a Forth application and the requirement that words be defined before they are used (there is no **`go to`** construct in Forth) promotes thorough testing at this level.
3. When the program has passed formal, documented test procedures and appears to be executing properly, then (and *only* then) is the OTA token compiler called to create a "module," consisting of the tokenised version of the program plus associated data and header information.
4. The module is sent to a terminal in the development system laboratory, where it is interpreted and the terminal checked for proper operation of the application.
5. When the application program is operating in accordance with formal, documented test procedures on the terminal in the development system laboratory (and *only* then), is the module containing the program certified and an authentication certificate attached to it.
6. The certified module is released for downloading and operation on terminals in the field.

The steps to create certifiable code in an ICC are quite similar. The same coding, testing, and token creation occur on a development system. The ICC with the new tokens is tested with terminals in the development laboratory and certified only when it is operating properly.

During the initial debugging process (Step 2) there should be extensive compile-time and execution-time error checking. In any case during module creation (Step 3) the token compiler itself will check further for compiling errors, which it must do in order to create the binary file containing the module. By the time certification (Step 5) is reached there cannot be any compile-time errors left and if testing is thorough there should not be any execution-time exceptions as well. Since the certified module to be downloaded or the certified contents of the ICC include an authentication certificate which prevents tampering, it appears that the terminal only needs to perform some very rudimentary compile-time checking on code downloaded to it either by modem or from an ICC. This simplifies the terminal program and its specifications.

The remainder of this section discusses compile-time errors and security as they would exist in the development system laboratory.

While compiling some limited security functions can be performed, but it is not possible to verify at this time that the compiled word will execute correctly; more extensive program security functions are performed at execution or run time.

Program validity (error checking) is addressed while compiling. Syntax checking of control structures is done at this time (e.g. matching **`IF`**s and **`THEN`**s, **`DO`**s and **`LOOP`**s, etc.) If the program has structure problems or uses unauthorised words (see word list discussion below) such that it will not compile without errors it is not added to the word list (or dictionary) and an error event is logged, possibly involving a return error message to the downloader.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Individual payment programs have access to kernel and library functions by name at the source level, and by tokens (for the kernel) or import lists (for library functions) within the terminals. Support functions at any level are not visible , and code space is not directly accessible by any means.

When a terminal application program is compiled, the compiler provides a standard global word list for all words in the kernel and extensions for which token equivalents have been defined, one word list for all library functions that have been imported, and one word list for the definitions internal to the application. A certified program would, when compiling, search first its own application word list, next the list of import functions, and last the kernel/extension word list. The word lists provide access to the tokens (for the application and kernel) or imported functions (for libraries or other high-level modules) as appropriate.

When code for an ICC is compiled, it resides in its own module, corresponding to a special token page that is reset at the end of the session.

F.5.5.2 Run-Time Integrity

Program run-time exception checking can be extensive. However, run-time checking is also costly in performance. Therefore, the run-time error checking is switchable, so that it may be performed during testing, and disabled or removed in a fully tested and certified program.

Most run-time security is concerned with the program's access of memory. The limits on memory may be a function of the program's security level. The following items could be checked during execution:

- Memory bounds checking of data space access for reads and writes, and absolute prohibition of writes to code space (this is part of the ANS Forth Standard).
- Check that addresses passed to indirect calls or branches are valid procedure entry addresses. This can only be done during the development process when the full source code on the host is available. This check is important because the address containing the execution token is in data space where writes are allowed.
- Data and return stack checks for underflow and overflow.
- Return stack checking for balancing and access type.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.6 Summary of the Open Terminal Architecture

The Europay Open Terminal Architecture is a complete system for maximising the efficiency of the terminal and ensuring the support of multiple applications across a terminal independent platform. The benefits of standardisation mean that the same application can be developed once and loaded into different manufacturers' terminals. Certification is made easier. Security and independence of applications within the terminal is ensured within the architecture.

The architecture has already been developed and implemented across a range of terminals using different CPU types. In order to ensure ease of implementation by terminal vendors, EPI has worked closely with its development partners to provide a range of tools to allow implementation and profitable development of the OTA.

These tools include:

1. **OTA Kernel Development Kit:** PC-based cross compiler for a specific target CPU, kernel source, and tools for interactive testing and code verification. This kit supports development of an OTA compliant kernel for a particular type of terminal.
2. **OTA Application Development Kit:** PC based Forth and C Language compilers that produce terminal independent programs in token form, as well as programming and debugging utilities. This set of programmer tools allows you to write your own applications such as a Customer Loyalty application which would run on the OTA kernel software.
3. **Software Development Services:** EPI, through its software partners, who are experts in the Open Terminal Architecture, can assist in developing custom kernels and application programs on your behalf.
4. **Training Courses:** Courses on all aspects of OTA development are available for software managers, developers and maintenance engineers.
5. **Design and Consultancy Services:** Europay and its software partners are available to advise you on how the Open Terminal Architecture Concept can benefit your business.

Europay International have already developed applications for Clip (electronic purse), as well as Credit and Debit which run on the Open Terminal Architecture and have been demonstrated on equipment from the leading card and terminal vendors in Europe. The EPI Open Terminal Architecture is available today. For more information contact your local Europay regional office or alternatively contact:

Europay International
Telephone +32 2 352 5111
Telefax +32 2 352 5807

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(Blank page)

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Index

A
a-addr data type 2–7
address space 3–11
aligned address 2–4, 2–7
B
BASE 3–14, 3–15, 5–71
big-endian 2–4, 5–43
binary, in EMV v3.0 2–4
C
C programming language 2–4
c-addr data type 2–7
card selected services 2–4, 4–19, 4–33
CATCH 3–16, 5–67
cell 2–4
character-aligned address 2–7
compile 2–4
compressed numeric 2–4
continuation location (of loops) 5–65
control characters, interpretation of C–111
control structures 5–44
counted string 2–4
cryptographic algorithms 5–86
current input device 3–14, 3–15
current output device 3–14, 3–15
D
data space 2–5, 3–11
    initialised 3–11
    uninitialised 3–11
data stack 2–8, 3–11
    in exception handling 3–16
database
    current 3–14, 3–15, 4–21
    current record 3–14, 3–15, 4–21
    deletion of 4–21
    instantiation 4–21
    non-volatile 3–11, 4–21, 4–22
    parameter block 3–14, 4–21, 5–76
    read-only 4–22
    services 4–21
    tokens 5–76
    types 4–23
    volatile 3–11, 4–22
date and time functions 4–19, 5–67, 5–68
DBCURRENT 3–14, 3–15, 4–21, 5–76
DBRECNUM 3–14, 3–15, 4–21, 5–76
debug device C–109
dev data type 2–7
development system 2–5, 2–6
device access 4–20, 5–68, C–107
device numbers C–108
DEVIP 3–14, 3–15
DEVOP 3–14, 3–15, 5–71
DPB *See database parameter block*
E
echar data type 2–7
EMV v3.0 2–5
exception frame 2–5, 3–16, 5–67
exception handling 3–16, 5–67
exception stack 2–5, 3–12, 3–16, 3–17, 5–67
execution pointer 2–5, 2–7
    in sockets 4–32
extended memory 3–11
extensible memory 3–14, 3–15
    in exception handling 3–16
    tokens 5–62
F
flag data type 2–7
flags 2–8
fmt data type 2–7
fn data type 2–7
Forth programming language 2–4
frame memory 3–11
frame pointers 3–11, 3–12, 3–13
frame stack
    in exception handling 3–16
frame tokens 5–60
H
hot card list management 4–28, 5–85
I
ICC I/O functions 5–72
initialised data 6–92
initialised data space 3–11
instantiate 2–5
interpret 2–5
ior data type 2–7, 2–8
ior usage C–108
K
kernel 2–5, 4–19

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

keyboard handling C–110
L
language selection 5–80
LANGUAGES database 4–26
len data type 2–7
library 2–5
loop control parameters 2–7, 5–65
loop-sys data type 2–7
M
magnetic stripe I/O functions 5–73, C–119
MDF See module delivery format
messages 4–25, 5–80
module 2–5, 4–19, 5–86
    delivery format 4–34
    executable 4–34
    execution process 4–34
    export list 6–94, 6–95
    format of 6–91
    handling services 4–33
    ID (MID) 4–34, 6–91
    import list 6–94
    library 4–34
    loading from ICC 4–39
    loading procedure 4–37
    management 3–15
    repository 2–5, 3–11, 3–18
modulo
    exponentiation 4–30
    multiplication 4–30
N
nest-sys data type 2–7
non-volatile databases 3–11, 4–21
non-volatile memory 3–11
num data type 2–7
number formatting functions 5–71
numeric formats 2–6
O
ordered databases 4–22
OTA tokens 5–43
P
power management I/O functions C–120
primary tokens 3–18, 5–43
Q
quote return register 3–10, 5–46, 5–66
quoting technique 5–45, 5–66
R
relocation data 6–92
return stack 2–8, 3–12, 3–15
    in exception handling 3–16
    rules of usage 3–12
S
secondary tokens 3–18, 5–43, 5–46
Secure Hash Algorithm (SHA) 4–30
serial port I/O functions C–115
socket
    list in module 6–92
sockets 4–32
    tokens 5–75
stack 2–6, 2–8, 3–10
T
terminal program 2–6
Terminal Resident Services 2–6, 3–15, 4–19
Terminal Selected Services 2–6, 4–19
terminal, defined 2–6
termination location (of loops) 2–7, 5–65
THROW 2–5, 3–16, 5–67
TLV
    constructed 4–26
    data format indicators 4–27
    Data Object List (DOL) 4–27
    length field 4–26
    primitive 4–26
    services 4–26
    status bit 4–27
    tag field 4–26
    tokens 5–81
    value field 4–26
token compiler 2–6, 3–18
token loader/interpreter 2–6, 3–18, 5–44
tokens 2–6, 3–18, 5–43
U
u data type 2–7
uninitialised data space 3–11
user variables 3–14, 5–52
V
vending machine I/O functions C–121
virtual machine 2–6, 3–9
    minimum resources 3–17
volatile databases 4–22
X
x data type 2–7
xp data type 2–7

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.204.15

Price based on 146 pages

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale