

BS ISO/IEC 20060:2010



BSI Standards Publication

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

bsi.

...making excellence a habit.
AVA Documents™
Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 20060:2010. It supersedes BS ISO/IEC 20060:2001 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/17, Cards and personal identification.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© BSI 2010

ISBN 978 0 580 69843 9

ICS 35.080; 35.240.15; 35.240.20

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 July 2010

Amendments issued since publication

Date	Text affected
------	---------------

INTERNATIONAL STANDARD

BS ISO/IEC 20060:2010
ISO/IEC
20060

Second edition
2010-07-01

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

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



Reference number
ISO/IEC 20060:2010(E)

AVA **Documents**
Document Sharing Hub
© ISO/IEC 2010

PDF disclaimer

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

Adobe is a trademark of Adobe Systems Incorporated.

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



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2010

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

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

Published in Switzerland

Contents

Page

Foreword	vi
Introduction.....	vii
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Symbols and abbreviated terms	5
5 Data types, stack notation and flags	6
5.1 Data Types	6
5.2 Stack Notation	7
5.3 Flags	7
6 OTA virtual machine.....	7
6.1 General principles	7
6.2 Virtual Machine CPU	8
6.2.1 Registers	9
6.2.2 Virtual Machine Size and Cells.....	9
6.2.3 Memory.....	9
6.2.4 Stacks	10
6.2.5 Frame Mechanism and Usage	11
6.2.6 Extensible Memory.....	12
6.2.7 User Variables.....	12
6.3 Virtual Machine Execution Features.....	13
6.4 Arithmetic	13
6.5 Exception Handling	14
6.6 Resources	15
6.7 Programs and Tokens.....	15
7 System Services	16
7.1 Time Handling.....	16
7.2 Devices and I/O Services	17
7.3 Database Services.....	17
7.3.1 The Database Parameter Block.....	19
7.3.2 Database Instantiation	21
7.3.3 Database Exception Handling.....	22
7.4 Language and Message Handling	22
7.5 TLV Services	23
7.5.1 Basic Principles.....	23
7.5.2 TLV Definitions	23
7.5.3 TLV References	24
7.6 Hot Card List Management.....	25
7.7 Cryptographic Services	25
7.7.1 Modulo Multiplication.....	26
7.7.2 Secure Hash Algorithm (SHA-1)	26
7.7.3 Modulo Exponentiation.....	27
7.7.4 Long Shift.....	27
7.7.5 Long Subtract	27
7.7.6 Incremental Secure Hash Algorithm (SHA-1)	27
7.7.7 Cyclic Redundancy Check (CRC)	28
7.7.8 DES Key Schedule.....	28
7.7.9 DES encryption/decryption	28

7.8	Vectored Execution Sockets	28
7.8.1	CSS Functions	29
7.8.2	Socket Security	29
7.8.3	Socket Organisation	29
7.9	Module Handling Services	29
7.9.1	Module Loading by MODEXECUTE	30
7.9.2	Module Loading Procedure	32
7.9.3	Module Loading by MODCARDEXECUTE	35
8	Token Set Definition	37
8.1	Overview	37
8.2	Conventions	37
8.2.1	Number Formats	37
8.2.2	Token Descriptions	38
8.2.3	Branch and Code Offsets	38
8.2.4	Addresses	38
8.3	Data Typing	39
8.4	Token Compression	39
8.4.1	Optimised Data Access	39
8.4.2	Special Procedure Calls	39
8.4.3	Quoting	39
8.5	Prefix Tokens	40
8.6	Stack Manipulation Tokens	41
8.7	Data Access Tokens	43
8.8	Literal Tokens	45
8.9	Address Generation Tokens	46
8.10	Arithmetic Tokens	47
8.11	Relational Tokens	51
8.12	String Tokens	53
8.13	Frame Tokens	56
8.14	Extensible Memory Tokens	58
8.15	Flow of Control Tokens	59
8.15.1	Branch Tokens	59
8.15.2	Call Tokens	60
8.15.3	Loop Tokens	61
8.15.4	Hybrid Tokens	62
8.15.5	Quoting Tokens	63
8.16	Exception Tokens	63
8.17	Date, Time, and Timing Tokens	64
8.18	Generic Device I/O Tokens	64
8.19	Formatted I/O Tokens	68
8.20	Integrated Circuit Card Tokens	69
8.21	Magnetic Stripe Tokens	70
8.22	Socket Tokens	71
8.23	Database Services Tokens	72
8.24	Language and Message Tokens	77
8.25	TLV Tokens	78
8.25.1	TLV Buffer Access	78
8.25.2	TLV Processing	80
8.25.3	TLV Sequence Access	81
8.26	Hot Card List Tokens	82
8.27	Cryptographic Algorithm Token	83
8.28	Module Management Tokens	83
8.29	Operating System Interface Tokens	84
8.30	Miscellaneous Tokens	84
9	Module Delivery Format	85
9.1	Module ID Format	86
9.2	Socket List	86
9.3	Relocation Section	87
9.4	Module Import List	88

9.5	Module Export List	88
9.6	Module Procedure List.....	89
Annex A	(normative) OTA Token Lists	90
Annex B	(normative) Exceptions and I/O Return Codes	97
Annex C	(normative) Device Control	101
Annex D	(normative) Operating System Calls	116
Annex E	(normative) Rules for Using a Data Object List (DOL).....	117
Annex F	(informative) System Overview	118
Bibliography		137

Foreword

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

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

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

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

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

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

Introduction

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

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

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

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

- core compactness,
- execution speed,
- security requirements.

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

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

1 Scope

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

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

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

2 Normative references

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

None.

3 Terms and definitions

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

3.1

aligned address

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

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

3.2

ANS Forth

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

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

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

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

3.5
binary
bit string

3.6
C
C programming language

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

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

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

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

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

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

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

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

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

cf. **return stack** (3.28)

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

3.14

EMV

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

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

3.15

exception frame

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

3.16

exception stack

stack used for the nesting of exception frames

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

3.17

execution pointer

value that identifies the execution semantics of a function

3.18

implementation conformance statement¹⁾

statement made by the supplier of an implementation or system claimed to conform to a given specification, stating which capabilities have been implemented

3.19

instantiate

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

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

3.20

interpret

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

3.21

kernel

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

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

3.22

library module

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

3.23

LISP

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

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

3.24
module

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

3.25
module repository

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

3.26
non-volatile

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

3.27
numeric

single-cell binary signed integer

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

3.28
return stack

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

3.29
stack

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

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

3.30
terminal

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

3.31
terminal program

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

3.32
terminal resident services
TRS

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

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

3.33
terminal selected services
TSS

terminal-independent code which implements service functions, also known as applications and libraries

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

3.34
token

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

3.35

token compiler

compiler that produces Open Terminal Architecture token modules

3.36

token loader/interpreter

software component on the terminal that processes downloaded tokens for execution on that terminal's CPU

3.37

virtual machine

theoretical microprocessor architecture

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

4 Symbols and abbreviated terms

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

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

5 Data types, stack notation and flags

5.1 Data Types

Table 1 — Data type designations used in OTA

Symbol	Data type	Size on stack
<i>a-addr</i>	Aligned address. Shall be a non-zero value exactly divisible by four.	1 cell
<i>addr</i>	Address. May not be zero.	1 cell
<i>c-addr</i>	Character-aligned address. May not be zero.	1 cell
<i>char</i>	Character (occupying the low-order byte of a stack item).	1 cell
<i>d</i>	Double-cell integer.	2 cells
<i>dev</i>	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
<i>echar</i>	Extended character (occupying the 2 low-order bytes of a stack item).	1 cell
<i>flag</i>	Flag (true or false). See Section 5.3 for conventional interpretations.	1 cell
<i>fmt</i>	TLV parameter, containing its format indicator and status. See Section 7 and Section 8.25 for usage.	1 cell
<i>fn</i>	Function code.	1 cell
<i>ior</i>	Result of an I/O operation. See Section 5.3 for conventional interpretations.	1 cell
<i>len</i>	Length of a string (0-65535).	1 cell
<i>loop -sys</i>	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
<i>nest -sys</i>	Implementation-dependent information kept on the return stack by function calls.	Implementation dependent
<i>num</i>	Signed integer (when used to represent an in-line value in a token sequence, may occupy less than one cell).	1 cell
<i>u</i>	Unsigned integer (when used to represent an in-line value in a token sequence, may occupy less than one cell).	1 cell
<i>ud</i>	Unsigned double-cell integer.	2 cells
<i>x</i>	Unspecified cell.	1 cell
<i>xp</i>	Execution pointer.	1 cell

5.2 Stack Notation

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

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

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

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

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

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

5.3 Flags

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

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

6 OTA virtual machine

6.1 General principles

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

The virtual machine provides the following services:

- standard CPU and instruction set, represented by OTA tokens;
- standard I/O support for common devices, with provisions for generic I/O to support additional devices that may be added;
- database management functions;
- TLV management, including format conversions and other functions;
- management of token modules, including maintaining them in storage (updating as necessary) and executing them on demand.

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

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

OTA provides program portability across heterogeneous terminals by treating terminal 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.

6.2 Virtual Machine CPU

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

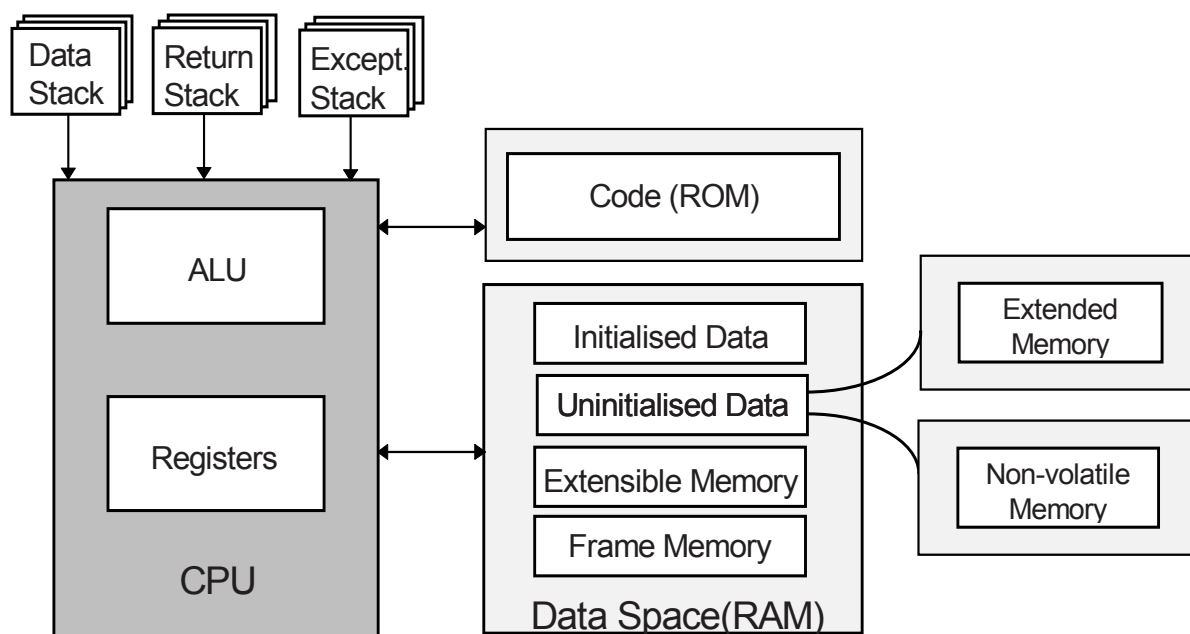


Figure 6-1 — Virtual machine architecture

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 6.2.3 Memory.

6.2.1 Registers

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

Table 2 — Virtual machine registers

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

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

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

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

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

6.2.2 Virtual Machine Size and Cells

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

6.2.3 Memory

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

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

- 1) *initialised data space*, which contains initial values specified at compile time and set when the kernel is activated and subsequently when a module containing initialised data is loaded;
- 2) *uninitialised data space*, which contains variables and static buffers allocated during pro-gram compilation. This data space is initialised to zero by the virtual machine;
- 3) *frame memory*, which is managed by the frame tokens (see Section 6.2.5 Frame Mechanism and Usage and Section 8.13 Frame Tokens).
- 4) *extensible memory*, which contains buffers allocated dynamically during program execution.

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.

6.2.4 Stacks

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

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

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

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

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

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

All programs shall observe the following rules:

- Programs may not assume that the data in any of these categories is guaranteed to be held on the return stack, and so may use only the retrieval mechanism specified explicitly for a given category in the sections referenced in Table 3.
- Programs that pass data in one of these categories to the virtual machine shall take appropriate action to recover the data storage from the virtual machine before exiting from the procedure in which it is passed.
- Programs shall assume that any data previously placed on the return stack is rendered inaccessible by passing data in one of these categories to the virtual machine.
- Programs shall assume that any data in one of these categories passed to the virtual machine is rendered inaccessible by executing code that places values on the return stack, until such time as those values are removed.

6.2.5 Frame Mechanism and Usage

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

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

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

As an example, consider the token code sequence:

LIT 5

LIT 10

LIT 20

SMAKEFRAME 2 4

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

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

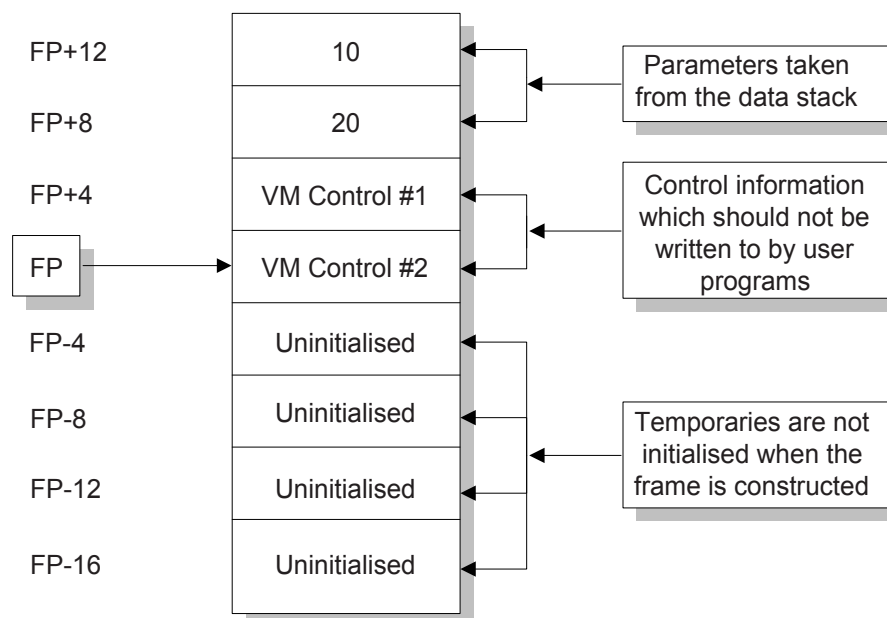


Figure 6-2 — Frame management example

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

Parameters can be fetched from the frame using the frame access tokens (see Section 5.13). For instance the cell at **FP+8**, which is **FP+2*CELLS**, can be loaded onto the data stack using the single-byte token **PPRFETCH2**; the cell at **FP-12**, which is **FP-3*CELLS**, can be stored to using **TFRSTORE3**.

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

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

6.2.6 Extensible Memory

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

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

Exception Handling and Section 8.16 Exception Tokens).

6.2.7 User Variables

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

Table 4 — User variables in the virtual machine

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

6.3 Virtual Machine Execution Features

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

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

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

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

6.4 Arithmetic

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

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

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

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

6.5 Exception Handling

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

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

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

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

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

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

Table 6 — Optional general exceptions from ANS Forth

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

-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 CATCHES used too much exception stack space.

6.6 Resources

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

Table 7 — Virtual machine resources

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

6.7 Programs and Tokens

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

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

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

7 System Services

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

- *The Kernel*, which includes machine-dependent implementations of I/O drivers and all functions required in this specification for supporting applications.
- *Terminal Resident Services* (TRS) include terminal-specific functions, libraries supporting these functions, program loading functions, and the main loop defining the terminal's behaviour.
- *Terminal Selected Services* (TSS) include service functions, also known as applications, and libraries supporting these functions. TSS contain only terminal independent code resident on the terminal. The TRS main program loop will select and call TSS functions as needed by a particular application.
- *Card Selected Services* (CSS) include functions supporting terminal applications, e.g. payment service functions that are used as part of a TSS application. CSS are resident on an ICC and downloaded to a terminal as requested. For terminals with two ICC readers (e.g., one for normal transactions and one for maintenance) there may be two independent sets of CSS (CSS1 and CSS2).

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

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

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

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

7.1 Time Handling

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

7.2 Devices and I/O Services

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

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

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

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

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

The guideline for usage of ior codes and **THROWS** 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 **THROWS** 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 **THROWS** may also be appropriate in many cases:

- If a procedure takes memory address(es) as parameter(s) and an invalid address is detected, exception -9 (Invalid memory address) should be thrown.
- If a procedure takes numerical value(s) as parameter(s) and an invalid value is detected, exception -24 (Invalid numeric argument) should be thrown.

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

7.3 Database Services

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

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

- At any given time the client has access to one currently selected database (**DBCURRENT**) and one currently-selected record number (**DBRECNUM**) that is shared across all defined databases.
- Information about each database is shared between client and server through a Database Parameter Block (DPB), which the server can access for reading and writing. The client “owns” the DPB, in the sense that it is in the client's data space; but the client is not allowed to access it directly. Instead, specific methods are used as described in Section 8.23 Database Services Tokens.
- The address of a window onto the current record (a record buffer) is provided by the server to the client for each client database; for certain database operations the client may pass the addresses of strings and key buffers to the server. For each database that has been made known to the server by a client module, a record buffer shall be provided by the VM. This buffer starts at an aligned address. The content of the record buffer associated with a particular database after a record selection remains available until the client selects another record from that database and should be considered to be read-only. If changes are made to the contents of the record buffer without using a database store token (**DBSTORE**, **DBSTRSTORE**, etc), the terminal may throw a -9 (Invalid memory address) or those changes may be lost when any database access token is executed. Any fetch from or store to a database record will cause the selected database record to be read from the database and placed into the record buffer. In this manner, any changes previously made to the data-base buffer will be lost and replaced with the contents of the record as held in database. Therefore, the database record always reflects the actual contents of the record as held in the database after a database fetch or store is performed. Except for these record buffers, compiled databases, and parameters passed on the stack by specific database functions, no other data space is shared between client and server. Programs may not assume that records in a database are contiguous in memory.
- A database is instantiated by the loading process for the module in which its DPB is defined. Volatile databases installed by application modules are uninstantiated automatically and transparently by the server when the application is terminated by Terminal Resident Services, when all server-allocated data storage relating to those databases is released.
- The server shall delete non-volatile databases when the module that defined them is replaced. If the module is loaded (as described in Section 7.9 Module Handling Services) when replaced, e.g., in the case of a TRS module, the server shall delete the module's non-volatile databases when the module is unloaded.

Figure 3 below illustrates this sharing of data space.

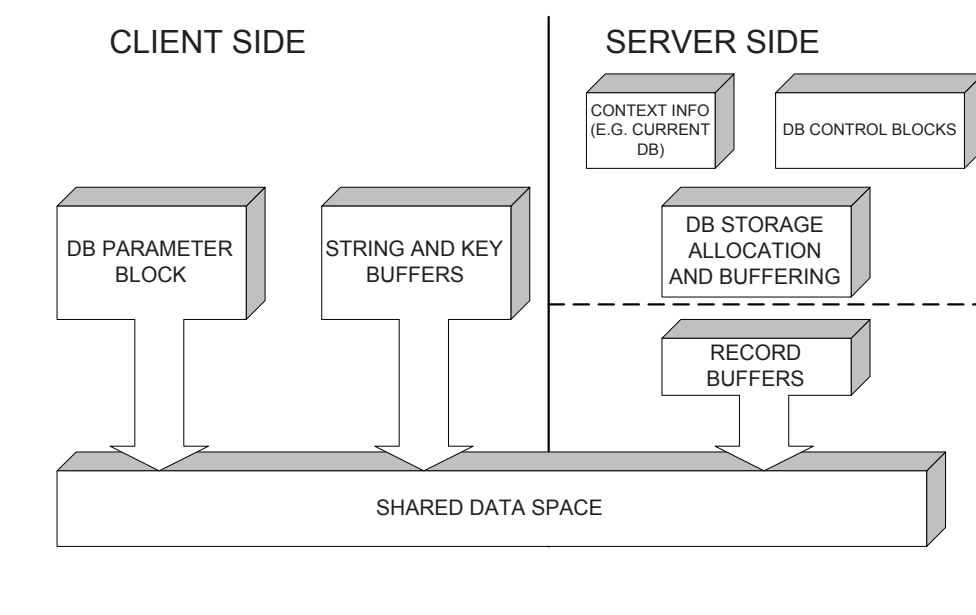


Figure 7-1 — Database memory access

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 shall be preserved across module loads or a power-down condition);
- *non-ordered* and *ordered* databases (ordered by a client-defined key);
- *read-only* and *read/write* databases.

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

7.3.1 The Database Parameter Block

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

Table 8 — DPB Structure

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

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

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

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

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

Bits 24-31 Bits 0-23

DB attributes DB kind

At present four **DB attributes** (bits 28-31) 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.
- 1 “Non-volatile” database whose content shall be preserved between module loads or across a power-down. If the module defining a non-volatile database is replaced, the database is destroyed when the old module is unloaded.
- 2 (Reserved for future use.)
- 3 “Compiled database” constructed by the compiler in a contiguous area of initialised data as fixed-length records. Records may not be added to or deleted from a compiled database, and the database may not be initialised at run-time, but otherwise full read-write capability shall be sup-ported.
- 4 Reserved for optional internal use by kernel implementations (e.g. in TLV management functions). Token programs may not refer to or make any assumptions about this kind.

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

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

7.3.2 Database Instantiation

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

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

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

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

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

7.3.3 Database Exception Handling

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

7.4 Language and Message Handling

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

Table 9 — Messages, by number and origin

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

NOTE This organisation reflects the needs given in [4]

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

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

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

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

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

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

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

Number 1 byte message number.

Message Counted string which holds the characters for the message.

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

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

Table 10 — Message table format

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

7.5 TLV Services

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

7.5.1 Basic Principles

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

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

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

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

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

7.5.2 TLV Definitions

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

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

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

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

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

7.5.3 TLV References

References to TLV definitions in the source code are compiled as either direct references to the definition structures defined above, or numerical tag values. Within certain binary TLV definitions, individual bits or groups of bits are defined to have certain meanings. These are referred to in OTA as “TLV bits”. References to TLV bits may be compiled with a literal pointing to a bit within the value field of the TLV. Bit 0 is the least significant bit of

the first byte, bit 7 is the most significant bit of that same byte, bit 8 is the least significant bit of the second byte and so forth.

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

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

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

7.6 Hot Card List Management

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

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

```
5413278000404808FFFF
5413278000404763FFFF
3625FFFFFFFFFFFFFFFF
45066367FFFFFFFFFFFF
```

The following input PANs would be found in the list:

```
5413278000404808
4506636700422497
362567810001
3625
```

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

```
541327800041234F
5413
45066F
```

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

7.7 Cryptographic Services

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

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

Table 11 — Cryptographic algorithm codes

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

7.7.1 Modulo Multiplication

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

$$\text{result} = \text{mod}(x*y,z)$$

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

The stack parameters for the **CRYPTO** token in this case are as follows: $(c\text{-}addr_1\ len_1\ c\text{-}addr_2\ len_2\ c\text{-}addr_3\ len_3\ c\text{-}addr_4\ 1 \text{ — } c\text{-}addr_4\ len_3)$

where $c\text{-}addr_1\ len_1$ and $c\text{-}addr_2\ len_2$ are the values x and y for the multiplication, and $c\text{-}addr_3$ and len_3 is the modulus z . The result is stored in $c\text{-}addr_4$. There shall be enough room at $c\text{-}addr_4$ to store len_3 bytes. The buffer $c\text{-}addr_4\ len_3$ is returned as a result.

Note that $c\text{-}addr_1$, $c\text{-}addr_2$ and $c\text{-}addr_3$ may be the same, and len_1 , len_2 , and len_3 shall be the same. The buffer $c\text{-}addr_4\ len_3$ shall not overlap with any of the input buffers.

7.7.2 Secure Hash Algorithm (SHA-1)

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

$$(c\text{-}addr_1\ len_1\ c\text{-}addr_2\ 2 \text{ — } c\text{-}addr_2)$$

where $c\text{-}addr_1\ len_1$ is the input buffer for computation and $c\text{-}addr_2$ is the buffer containing the result, which is always 20 bytes in length.

The 20-byte result buffer $c\text{-}addr_2$ shall not overlap the input buffer $c\text{-}addr_1\ len_1$.

7.7.3 Modulo Exponentiation

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

$$\text{result} = \text{mod}(x^y, z)$$

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

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

$$(c\text{-}addr_1\ len_1\ u\ c\text{-}addr_2\ len_2\ c\text{-}addr_3\ 3 \text{ --- } c\text{-}addr_3\ len_2)$$

where $c\text{-}addr_1\ len_1$ is the argument x , u is the exponent y , and $c\text{-}addr_2\ len_2$ is the modulus z . The result is stored in $c\text{-}addr_3$, and the buffer $c\text{-}addr_3\ len_2$ is returned as a result. It shall not overlap with any of the input buffers. There shall be enough room at $c\text{-}addr_3$ to store len_2 bytes.

7.7.4 Long Shift

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

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

The stack parameters for the **CRYPTO** token in this case are as follows: ($c\text{-}addr\ len\ num\ 4 \text{ --- } c\text{-}addr\ len$) where $c\text{-}addr\ len$ 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 $c\text{-}addr\ len$.

7.7.5 Long Subtract

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

The stack parameters for the **CRYPTO** token in this case are as follows: ($c\text{-}addr_1\ c\text{-}addr_2\ len\ 5 \text{ --- } c\text{-}addr_2\ len$) 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 $c\text{-}addr_2\ len$.

7.7.6 Incremental Secure Hash Algorithm (SHA-1)

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

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

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

7.7.7 Cyclic Redundancy Check (CRC)

The stack parameters for the **CRYPTO** token in the case of the CRC algorithm are as follows: ($c\text{-}addr\ len\ num\ u_1\ u_2\ 7\ \text{---}\ u_3$) $c\text{-}addr\ len$ 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 representing the most significant term has been discarded (e.g.: the polynomial $x^{16} + x^{12} + x^5 + 1$ is represented by the binary value 1000000100001). u_3 is the 32 bit value of the new CRC computed over the input data.

7.7.8 DES Key Schedule

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

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

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

7.7.9 DES encryption/decryption

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

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

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

7.8 Vectored Execution Sockets

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

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

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

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

7.8.1 CSS Functions

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

7.8.2 Socket Security

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

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

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

Code that wishes to restrict access to any sockets before further code is executed may execute a **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 8.22 Socket Tokens) can determine whether the request was accepted or denied by the call to **PLUGSOCKET**.

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

7.8.3 Socket Organisation

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

7.9 Module Handling Services

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

- Tokenised code
- Initialised data
- Uninitialised data allocation

- Database definitions
- TLV definitions
- Socket list
- Module interdependencies

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

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

- If modules are not stored in MDF as described in Section 9, then they shall behave as though they were.
- The VM shall protect modules within the repository from modification by any other module.
- The VM shall make provision for a new version of a given module to be placed in the repository while a module of the same module ID exists for execution purposes.

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

7.9.1 Module Loading by **MODEXECUTE**

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

Mark and Save Resources

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

- The position of the extensible memory pointer, the frame pointer, and the frame end pointer.
- The contents of the entire current socket list.
- The TLVs registered in the TLV tag name list.
- Other internal data the VM implementation needs in order to manage the activation and execution of modules. *Load Module*

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

Executable?

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

Execute Module

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

Release Resources

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

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

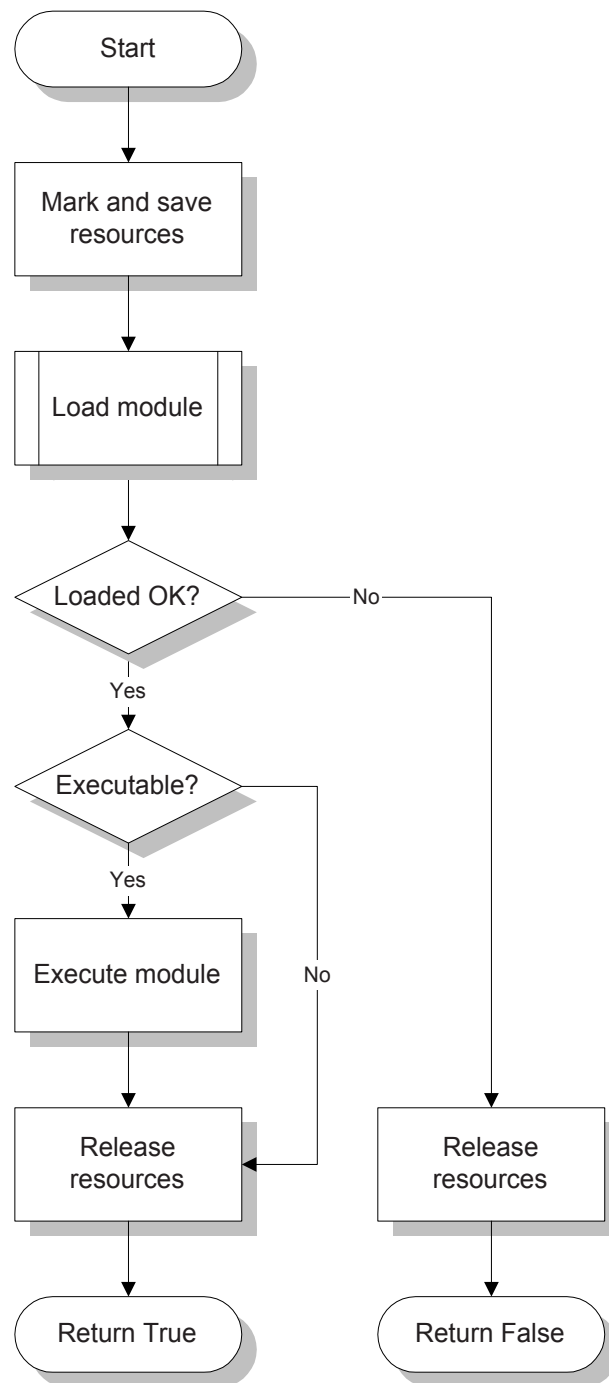


Figure 7-2 — Module execution procedure

7.9.2 Module Loading Procedure

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

General Errors While Loading

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

Order of Flowchart Items

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

Module Loaded?

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

Module in Repository?

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

Allocate Uninitialised Data Area and Zero

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

Allocate and Prepare Module's Initialised Data

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

Process Module's List of TLVs

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

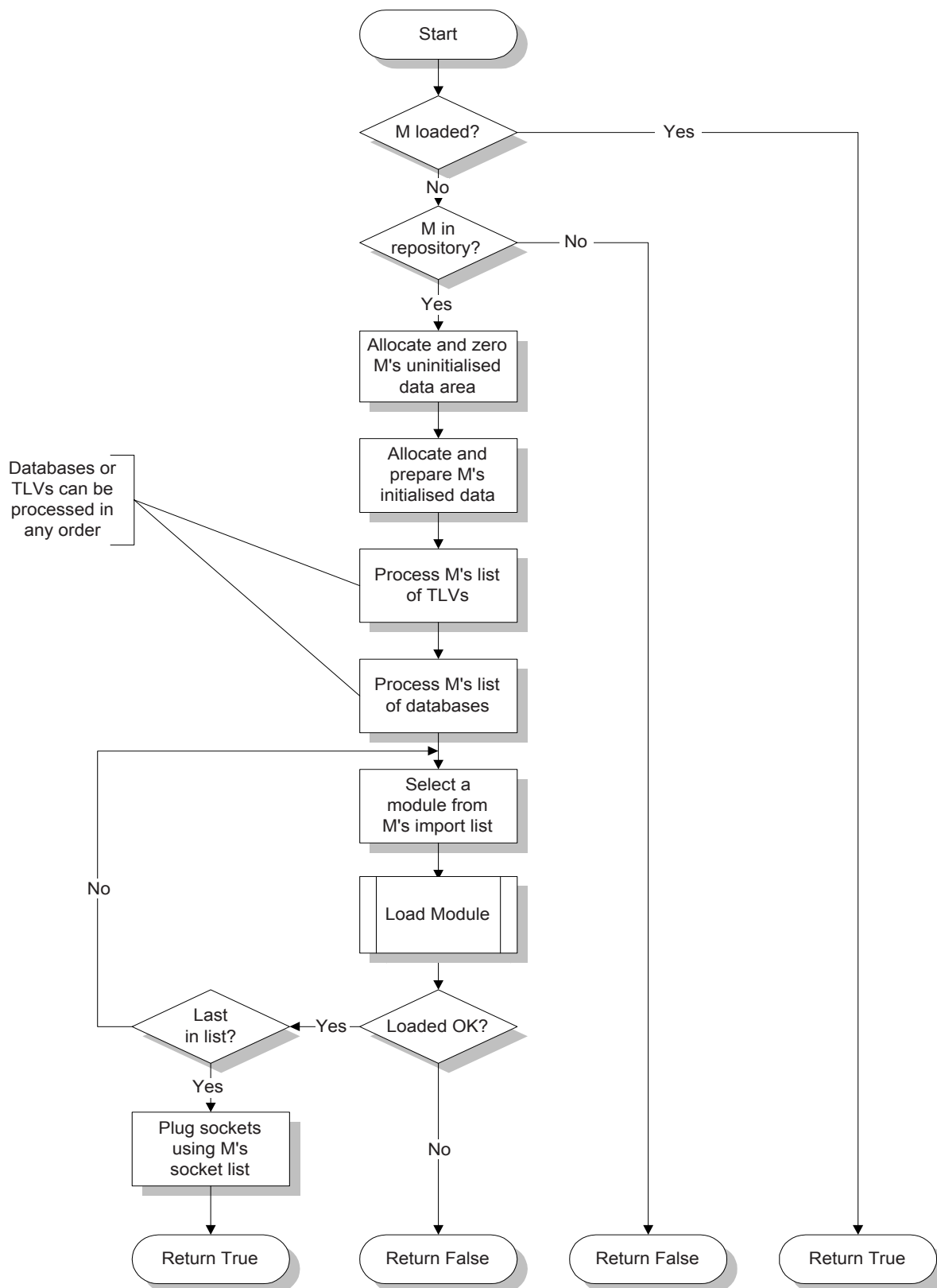


Figure 7-3 — Module loading procedure

Process Module's List of Databases

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

Select a Module from the Import List

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

Loaded OK?

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

Last Module?

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

Plug Sockets in Module's Socket List

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

7.9.3 Module Loading by MODCARDEXECUTE

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

Mark and Save Resources

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

- The position of the extensible memory pointer, the frame pointer, and the frame end pointer.
- The contents of the entire current socket list.
- The TLVs registered in the TLV tag name list.
- Other internal data the VM implementation needs in order to manage the activation of card modules.

Load Module

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

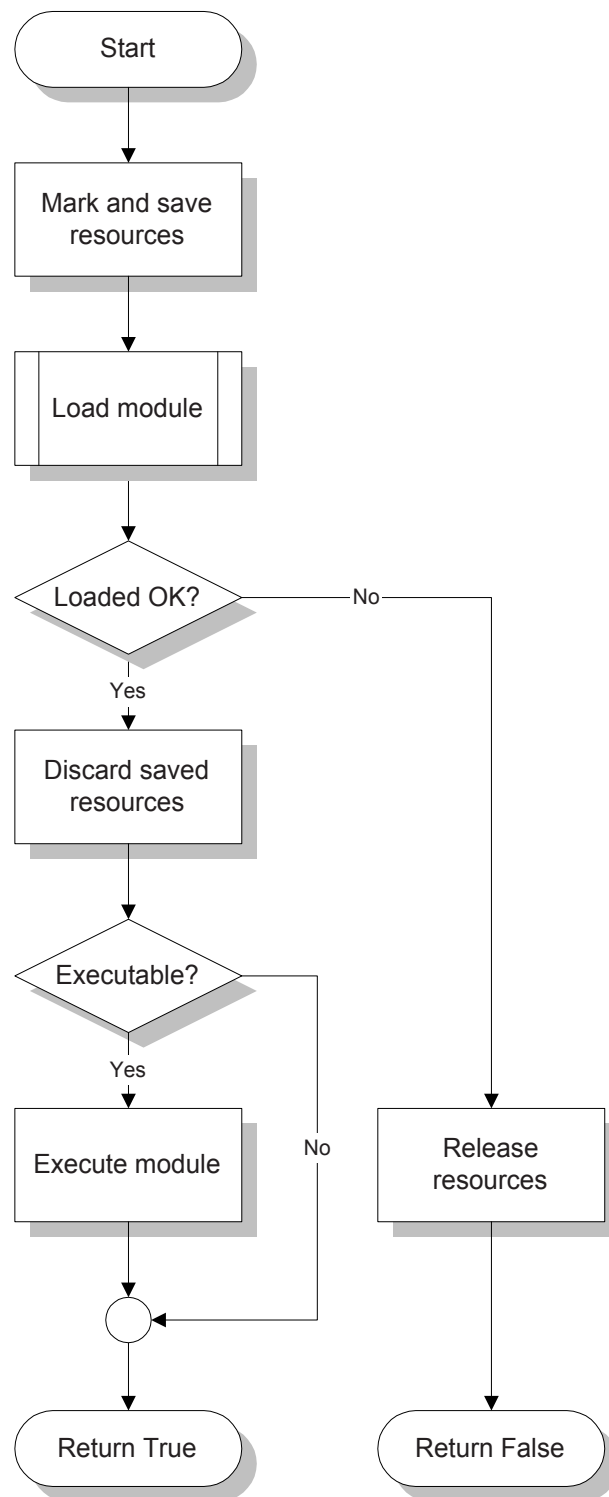


Figure 7-4 — ICC module execution procedure

Release Resources

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