Europay

Open Terminal Architecture (OTA) Specification

Volume 2: Forth Language Programming Interface

Europay International S. A
198A Chaussée de Tervuren
1410 Waterloo, Belgium

Version 3.0 – 16 February 1999

Table of Contents

About This Book vii

Purpose vii
Scope vii
Audience vii
Document Change Control viii
Additional Copies viii
How This Book is Organised viii
Aids in Using This Book viii
Related Publications ix
Reference Materials ix

1 Management Summary 1–1

2 Conventions 2–3

2.1 Abbreviations 2–3
2.2 Glossary 2–4
2.3 Data Types 2–8
2.4 Stack and Parsed-text Notation 2–9
2.5 Flags and IOR Codes 2–10
2.6 Forth Glossary Notation 2–11
2.7 Word Behaviours 2–12

3 Compiler Functions 3–13

3.1 OTA Compiler Principles 3–13
3.2 Compiler Scoping 3–14
3.2.1 Specifying a Scope 3–14
3.2.2 Effects of Scoping on Colon Definitions 3–15
3.2.3 Effects of Scoping on Data Object Defining Words 3–16
3.2.4 Scope Specification 3–17
3.3 Compiler Directives 3–18
3.4 Interpreter Control 3–19
3.4.1 Input Number Conversions 3–19
3.4.2 Other Interpreter Functions 3–21
3.5 Data Space Selectors 3–22

3.6 Data Space Management 3–23
3.7 Defining Words 3–25
3.8 Dictionary Management 3–30
3.9 Control Structures 3–31
3.10 Locals 3–36
3.10.1 Compilation behaviour 3–36
3.10.2 Syntax restrictions 3–37
3.10.3 Locals definitions 3–37

4 The OTA Wordset 4–39

4.1 Stack Manipulation 4–39
4.1.1 Data Stack 4–39
4.1.2 Return Stack 4–40
4.2 Data Access 4–41
4.3 Arithmetic 4–42
4.3.1 Single-Precision Arithmetic 4–42
4.3.2 Double-Number And Mixed-Precision Arithmetic 4–45
4.3.3 Logic 4–46
4.4 Relation and Comparison 4–47
4.5 Strings 4–49
4.6 Extensible Memory Management 4–53
4.7 Exception Handling 4–54
4.8 Date, Time, and Timing Services 4–55
4.9 Generic Device Management 4–56
4.9.1 Generic Device I/O 4–57
4.9.2 Current Device I/O 4–58
4.9.3 Communication (Modem) Services 4–61
4.10 Formatted I/O Words 4–62
4.11 Integrated Circuit Card I/O 4–64
4.12 Magnetic Stripe I/O 4–64
4.13 Sockets and Socket Management 4–66
4.14 Database Services 4–68
4.14.1 Basic Concepts 4–68
4.14.2 Database and Record Definitions 4–68
4.14.3 Database Attributes 4–72
4.14.4 Database Access and Management 4–73
4.14.5 Name List Management 4–75
4.15 Language and Message Handling 4–76
4.15.1 Cardholder Language Selection 4–76
4.15.2 Messages and Message Displays 4–78
4.16 TLV Services 4–79
4.16.1 TLV Description Syntax 4–79

4.16.1.1 Format Specifiers 4–79
4.16.1.2 Internal data structures 4–80
4.16.1.3 TLV string creation 4–80
4.16.1.4 TLV bits 4–81
4.16.2 TLV Data Definitions 4–82
4.16.3 TLV String Services 4–84
4.16.4 TLV Management 4–86
4.17 Hot Card List Management 4–88
4.18 Cryptographic Services 4–89
4.19 Operating System Interface 4–90
4.20 Miscellaneous 4–90
4.21 Debug support 4–91

5 Module Interfaces 5–93
5.1 Module Definition Words 5–93
5.2 The .DEF file format 5–96
5.3 Module Management Words 5–97

Appendix A: Bibliography A-101

Appendix B: Required Forth Functions B-103

Appendix C: Exception Codes C-111

Index 117

Control of Document 119

List of Tables

1. Reference materials for this volume. ix
2. Data type designations used in OTA 2–8
3. Forth-specific data types 2–9
4. Parsed text abbreviations 2–10
5. Compiler directives that specify scope 3–15
6. Scopes in which colon definitions are accessible 3–15
7. Scopes in which data objects are accessible 3–16
8. Interpreting behaviour of data objects 3–16
9. Number conversion prefixes 3–20
10. Memory areas in the VM. 3–22
11. Device number assignments 4–56
12. ISO parameter track selection codes 4–65
13. Named TLV formats 4–79
14. Compiling operators for TLV values 4–81

15. Codes specifying security algorithms 4–89
16. Codes identifying classes of exported words in a .DEF file 5–97
17. ANS Forth **THROW** codes in OTA kernels 111
18. OTA **THROW** codes 112
19. OTA I/O Return Codes 114

About This Book

Purpose

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

The purpose of this document is to provide a specification for the Forth language interface to a standard set of software functions to be provided in all OTA payment terminals.

Scope

OTA defines a set of standard software functions whose programming interface is common across all terminal types. These functions are based on a standard *Virtual Machine* (VM), 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 virtual machine functions.

The specification of the OTA Virtual Machine is given in Volume 1 of this series. This volume contains the Forth language bindings for VM procedures, as well as specifications for compliant Forth language compilers. Volume 3 contains equivalent C language bindings and compiler specifications.

This document does not attempt to present a tutorial on the Forth language. For those wishing such an introduction, some books and other sources for Forth information are listed in Appendix A.

Compilers generating OTA VM tokens from either Forth or C language source code have been developed for the purpose of testing and validating aspects of OTA technology. These are available separately. However, this document is not intended to mandate use of these compilers, nor are details of their implementation considered to be mandatory.

Audience

This manual is intended for anyone desiring to evaluate OTA technology, develop OTA VM implementations, or develop payment programs or libraries designed to run on OTA VM implementations. General knowledge of computers and programming is assumed. Some familiarity with the Forth programming language will be helpful; some recommended reference sources are listed in Appendix A.

Document Change Control

The information in this manual supersedes and replaces all previous drafts issued.

Additional Copies

Requests for copies of Europay publications should be made to:

Europay Documentation Centre
198A Chaussée de Tervuren
1410 Waterloo
Belgium.

How This Book is Organised

This document is divided into the following chapters:

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

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

Chapter 3 — Compiler Functions describes syntax and compiler notation for defining words and compiler directives.

Chapter 4 — The OTA Wordset presents a glossary of the generic words supported by VM tokens, as well as Forth language bindings for OTA-specific procedures, including (as appropriate) compiler syntax for defining databases, TLVs, etc.

Chapter 5 — Module Interfaces describes the syntax for declaring module import and export procedures, specifying module characteristics, and other aspects of module management.

Aids in Using This Book

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

- A list of all of the tables present in this book, found on page v.
- A list of abbreviations referred to in this book, found on page 2–3.
- A glossary of terms used in this book, found on page 2–4.
- An alphabetic list of all Forth words, with page numbers, in Appendix B.
- An index of topics covered in this book, found at the end of the document.

Related Publications

The following Europay publications contain material directly related to the content of this book.

- *Integrated Circuit Card Specifications for Payment Systems*, Version 3.0, June 30, 1996
- *Integrated Circuit Card Terminal Specification for Payment Systems*, Version 3.0, June 30, 1996
- *Integrated Circuit Card Application Specification for Payment Systems*, Version 3.0, June 30, 1996
- *Europay Open Terminal Architecture (OTA) System Architecture*, Version 1.0, October 30, 1995
- *Europay Open Terminal Architecture (OTA) Specification, Volume 1: Virtual Machine Specification*, Version 3.2, May 13, 1998

Reference Materials

The reference materials in Table 1 may be of use to the reader of this manual:.

Table 1: Reference materials for this volume.

Document	Title
ANSI X3.215:1994	Forth Programming Language
ANSI X9.31-1 (Draft)	Public key cryptography using irreversible algorithms for the financial services industry — Part 1: The RSA Signature Algorithm
ANSI X9.30-2:1993	Public key cryptography using irreversible algorithms for the financial services industry — Part 2: The Secure Hash Algorithm (SHA)
FIPS PUB 180-1:1994	Secure Hash Standard
ISO 639:1988	Codes for the representation of names and languages
ISO 3166:1993	Codes for the representation of names of countries
ISO 4217:1990	Codes for the representation of currencies and funds
ISO/IEC 8825:1990	Information technology — Open systems interconnection — Specification of basic encoding rules for abstract syntax notation one (ASN.1)

1 Management Summary

This document is the second in a series of documents containing the Implementation Specification of Europay's Open Terminal Architecture. Volume 1 specifies the OTA Virtual Machine, its services, and its token set. Volume 3 describes the C language bindings and tool chain requirements.

This technical specification contains a source language specification providing programmer access to all VM functions. This specification serves as a guide to providers of OTA VM implementations and Forth language compilers. In addition, it enables the application programmer to generate OTA-compliant application software that is compact, portable, and certifiable on all target terminals.

Except as noted, this document follows the conventions defined by the ANS Programming Language Forth. The Forth words described in this document fall into two major groups: a core wordset, composed of words selected from ANS Forth and adhering fully to this standard; and extensions designed to support payment terminal programs.

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

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

This document provides the specifications for the Forth language programming interface in several layers:

- Compiler Functions (Section 3);
- The OTA Wordset, including Forth core words and OTA extensions (Section 4);
- Module Interfacing (Section 5).

This specification does not mandate any specific implementation strategy. An OTA Forth compiler will be considered compliant if all the functions listed in this section are provided (on the host and/or in the target machine, as indicated) and perform as described herein.

2 Conventions

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

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

2.1 Abbreviations

ANS	American National Standard
API	Application Programming Interface
ASN	Abstract Syntax Notation
ATM	Automated Teller Machine
BCD	Binary Coded Decimal
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSS	Card Selected Services
DOL	Data Object List
DPB	Database Parameter Block
EMV	Europay-MasterCard-Visa consortium
EPI	Europay International, S.A.
H	Hexadecimal (base 16) when used as a subscript
ICC	Integrated Circuit Card
IDL	Interactive Development Link
IFD	IC Interface Device
ISO	International Organisation for Standardisation
K	1024
LSB	Least Significant Bit
MDF	Module Delivery Format
MID	Module ID
N/A	Not Applicable
OS	Operating System

OTA	Europay Open Terminal Architecture
PAN	Primary Account Number
PC	Personal Computer
PIN	Personal Identification Number
POS	Point Of Sale (terminal)
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
RFU	Reserved for Future Use
ROM	Read-Only Memory
RSA	Rivest, Shamir, Adleman (cryptography algorithm)
SHA	Secure Hash Algorithm (SHA-1)
TLV	Tag-Length-Value data object (per ISO/IEC 8825:1990)
TRS	Terminal Resident Services
TSS	Terminal Selected Services
VM	Virtual Machine

2.2 Glossary

Aligned address	The address of a memory location at which a character, cell, cell pair, or double-cell integer can be accessed. The OTA Virtual Machine requires that aligned addresses be exact multiples of 4.
ANS Forth	The Forth programming language as defined by the American National Standard X3.215, 1994.
Application	Software that implements a payment system product. A Terminal will select one of these during the processing of a transaction.
ASCII String	A string whose data contains one ASCII character per byte.
Big-endian	Describes a byte-ordering system in which the highest-order byte of a cell is at the lowest address (i.e., appears first in a data stream). The OTA Virtual Machine uses Big-endian byte order in token modules and card communication.
Binary	In EMV v3.0, the term *binary* is used for both numbers and bit strings. When a data element is a number, this is to be interpreted as an unsigned integer.
Binary String	A string whose data consists of a sequence of bytes, the interpretation of which is application-specific.

C	The C programming language. C language support for OTA is documented in a separate volume.
Card Selected Services (CSS)	Card-resident code that provides procedures supporting terminal transactions, usually payment service procedures that are used as part of a TSS application.
Cell	The primary unit of information storage in the architecture of an OTA system. The standard size of a cell in the OTA Virtual Machine is four bytes.
Cell pair	Two cells that are treated as a single unit.
CN String	A string whose data contains two digits per byte *(compressed numeric)*, padded with trailing hexadecimal Fs as needed.
Code space	The logical area of the virtual machine in which OTA procedures are implemented.
Compile	Transform higher-level specifications of software and/or data into executable form. The executable form for the OTA Virtual Machine is OTA tokens.
Compressed numeric	In EMV 96, the term *compressed numeric* specifies that a number is represented in BCD format, left justified and padded with trailing hexadecimal Fs.
Counted string	A data structure consisting of one character containing the length followed by 0–255 data characters.
Cross-Compilation	Generation of an executable program for a target terminal CPU on a host system that may be based on a different CPU.
Data space	The logical area of the Virtual Machine that can be accessed by OTA tokens.
Data stack	A stack that may be used for passing parameters between procedures. When there is no possibility of confusion, the data stack is referred to as "the stack." Contrast with *return stack*.
Development System	Another name for the *host*, used to develop OTA Terminal Programs and Applications.
EMV	A consortium of Europay International S.A., MasterCard International Incorporated, and Visa International Service Association.
EMV 96	*Integrated Circuit Card Specification for Payment Systems*, Version 3.0, June 30, 1996, Parts I–IV.
Exception frame	An exception frame is the implementation-dependent set of information recording the current execution state necessary for the layered exception processing used on the VM.
Exception stack	A stack used for the nesting of exception frames. It may be, but need not be, implemented using the return stack.

Execution pointer	A value that identifies the execution behaviour of a procedure. In ANS Forth, this is referred to as an *execution token;* however, in this specification, the term has been changed to avoid confusion with OTA tokens.
Host	A computer used to develop OTA terminal VM implementations, Terminal Programs, and/or OTA Payment Applications. During development, the host is usually connected to the *target* Terminal, so that program coding and testing can be carried out interactively.
Immediate Word	A Forth word whose compiling behaviour is to perform its execution behaviour. Such words are commonly used to compile program flow structures.
Input Stream	ASCII string data input to the host interpreter. It may come from an input device (such as a keyboard) or from a file. The input stream is the vehicle by which user commands, program source, and other data are provided to the host system.
Instantiate	Register a local instance of a data structure with the Virtual Machine. At power-up in a terminal, initialised data items and VM databases must be instantiated. Further data, databases and TLV definitions may be instantiated when a module is loaded.
Interpret	For OTA token code, at run time, identify the procedure associated with a token value in the code and execute it.
Kernel	The entire standardised set of procedures mandated to be present on every terminal to implement the OTA Virtual Machine. The Kernel is composed of the Core (standard language procedures conforming to ANS Forth) and Core Extension (OTA specific procedures for I/O, language selection, and other special terminal needs).
Library Module	A set of software procedures in OTA token code with a published interface, providing general support for Terminal Programs and/or Applications.
Module	A collection of software procedures and/or data, compiled together and presented in token form as a package whose delivery format is specified in Volume 1 of this OTA Specification.
Module Repository	A non-volatile medium for storing OTA modules within a terminal.
Non-volatile	Guaranteed to be preserved across module loading or terminal power-down and rebooting.
Numeric	In EMV 96, the term *numeric* specifies that a number is represented in BCD format, right justified and padded with leading hexadecimal zeroes. In this document, *numeric* indicates a single-cell binary signed integer.
Parse	To select and exclude a character string from the parse area using a specified set of delimiting characters, called delimiters.

Parse Area	The portion of the input stream that has not yet been processed by the host interpreter and is, therefore, available for processing by the host interpreter and other parsing operations.
Payment System	For the purposes of these specifications, Europay International S.A., MasterCard International Incorporated, or Visa International Service Association.
Return stack	A stack that may be used for program execution nesting, do-loop execution, temporary storage, and other purposes.
Stack	An area in memory containing a last-in, first-out list of items. Stacks in the OTA Virtual Machine are discussed in Volume 1 of this Specification.
Target	When connected to a development Host, the Terminal containing the OTA transaction processing software is often referred to as the Target.
Terminal	Any POS (Point of Sale) machine, ATM (Automated Teller Machine), vending machine, etc.
Terminal Program	The overall supervisory program that a Terminal executes. The Terminal Program contains code to select among one or more Transactions and associated payment Applications.
Terminal Resident Services (TRS)	Includes terminal-specific non-payment procedures, program-loading procedures, and the main Terminal Program loop defining the terminal's behaviour. A TRS is usually provided by a terminal's manufacturer.
Terminal Selected Services (TSS)	All payment service procedures, also known as Applications, and libraries supporting these procedures. TSS contain only terminal-independent code, and are provided by the individual Payment Systems. The TRS main program loop selects and calls TSS procedures as needed by a particular Transaction.
Token	A one- or two-byte code representing a CPU-independent instruction for the Virtual Machine.
Token Compiler	A compiler that produces OTA token modules.
Token Loader/ Interpreter	A software component on the Terminal that processes downloaded Tokens for execution on that terminal's CPU.
Token Interactive Debugging Environment (TIDE)	A software platform for testing token programs, with the ability to simulate a variety of terminal types and configurations.
Transaction	An action taken by a terminal at the user's request. For a POS terminal, transactions might be payment for goods, return of goods, etc. A Transaction selects among one or more Applications as part of its processing flow.

Virtual Machine	A theoretical microprocessor architecture. In order to provide a single standard interface, all OTA terminals are coded to emulate a common Virtual Machine whose parameters are included in this specification.

The following additional glossary definitions pertain to Forth language features in OTA.

Data Field	The data space associated with a word defined via **CREATE**.
Definition	A Forth execution procedure compiled into the dictionary.
Dictionary	An extensible structure that contains definitions and associated data space.
Name Space	The logical area of the dictionary in which definition names are stored during compilation and testing in the host computer.
Word	The name of a Forth definition.

2.3 Data Types

Table 2 gives a description of the ANS Forth notation used to refer to the different data types that may appear in stack notation or descriptions in this manual. Additional tables in this section describe OTA-specific data types and other notational conventions.

Table 2: Data type designations used in OTA

Symbol	Data type	Size on stack
a-addr	Aligned address. Must be a non-zero value exactly divisible by four.	1 cell
addr	Address. May not be zero.	1 cell
c-addr	Character-aligned address. May not be zero.	1 cell
char	Character (occupying the low-order byte of a stack item)	1 cell
d	Double-cell integer	2 cells
dev	Device number, an integer identifying a logical I/O device supported by a VM implementation. A list of defined device numbers is given in Volume 1 of this Specification.	1 cell
echar	Extended character (occupying the two low-order bytes of a stack item).	1 cell
flag	Flag (true or false). See Section 2.5 for conventional interpretations.	1 cell
fmt	TLV parameter, containing its format indicator. See Section 4.16 for usage.	1 cell
fn	Function code.	1 cell

Table 2: Data type designations used in OTA *(continued)*

Symbol	Data type	Size on stack
ior	Result of an I/O operation. See Section 4.9 for conventional interpretations.	1 cell
len	Length of a string (0–2,147,483,647). A counted string may not be longer than 255 characters.	1 cell
loop-sys	Loop-control parameters. These include implementation-dependent representations of the current value of the loop index, its upper limit, and a pointer to a *termination location* where execution continues following an exit from the loop.	Implementation dependent
nest-sys	Implementation-dependent information for procedure calls. It may be kept on the return stack.	Implementation dependent
num	Signed integer (when used to represent an in-line value in a token sequence, may occupy less than one cell).	1 cell
tag	Unsigned integer giving the value of a TLV's tag. Should always be expressed in hexadecimal.	1 cell
u	Unsigned integer (when used to represent an in-line value in a token sequence, may occupy less than one cell).	1 cell
ud	Unsigned double-cell integer.	2 cells
x	Unspecified cell.	1 cell
xp	Execution pointer, called *xt* (execution token) in ANS Forth.	1 cell

The following data types (in addition to those defined in Table 2) are used in describing stack behaviour in Forth compiler directives. The size of these items is implementation-defined.

Table 3: Forth-specific data types

Symbol	Data type
colon-sys	Definition compilation.
db-sys	Database context information (database and record settings).
dest	Control flow destination.
do-sys	Do-loop structure.
orig	Control flow origin.

2.4 Stack and Parsed-text 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.

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 2 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.

If, in addition to using stack parameters, a definition parses text, that text is specified by an abbreviation from Table 4, shown surrounded by double-quotes and placed between the *before* parameters and the "—" separator in the data stack description; for example:
(num "<spaces>name" —)

Table 4: Parsed text abbreviations

Abbreviation	Description
<char>	The delimiting character marking the end of the string being parsed.
<chars>	Zero or more consecutive occurrences of the character *char*.
<space>	A delimiting space character.
<spaces>	Zero or more consecutive occurrences of the character *space*.
<paren>	A delimiting right parenthesis.
<quote>	A delimiting double quote.
<eol>	An implied delimiter marking the end of a line.
ccc	A parsed sequence of arbitrary characters, excluding the delimiter character.
hex-digits	A sequence of characters all of which must be valid hex digits (0–9 plus A–F)
name	A name of a Forth word (either previously defined or being defined), equivalent to *ccc<space>* or *ccc<eol>*.

2.5 Flags and IOR Codes

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* (I/O Result) to report the results of an operation. See Appendix C for specific *ior* values. An *ior* may be treated as a flag, in the sense that a non-zero value is *true*; however, it is not necessarily a well-formed flag, as its value is often used to convey additional information. A returned value of zero for an *ior* shall mean successful completion (i.e., no error); non-zero values may indicate an error condition or other abnormal status, and are device-dependent (see lists in Appendix C).

2.6 Forth Glossary Notation

Words described in this manual are grouped functionally. An alphabetic list of all words is given in Appendix B.

Each entry consists of three parts: an index line, a stack notation, and a semantic description of the word. The stack notation follows the conventions in Section 2.3 and Section 2.4, above. The index line is a single-line entry containing, from left to right:

- Definition name, in upper-case, monospaced, boldface letters;
- Natural-language pronunciation (if it is not obvious from the name);
- Special designators, as applicable, indicating in which *scope* (see Section 3.2) the word is defined. The correlation between the scope in which a word is defined and where it may be accessed depends upon its type.

  A **`ASSEMBLER`** scope. These words are used to build and modify target definitions expressed in CPU machine language or VM tokens.

  C **`COMPILER`** scope. These words, which may be used in compiling mode only, are commonly used to construct program structures (e.g., loops and conditionals).

  H **`HOST`** scope. These words are used to construct **`COMPILER`** or **`INTERPRETER`** definitions.

  I **`INTERPRETER`** scope. These words are used to build data structures in the target image.

  T **`TARGET`** scope, including all the functions to be executed in the target. These functions may be executed interpretively only on systems that support an *interactive* mode (see Section 2.7 and Section 3.2) when a suitable target is available.

  In addition, some words are designated by:

  F ANS Forth word (including all optional wordsets in that standard). Additional information about this word and its usage may be found in ANS Forth.

- OTA token providing the execution behaviour of the word. If the token equivalent is a sequence (more than one token), the comment "(Sequence)" is placed here, and a possible implementation of the sequence is given in the semantic description below. If no token name is shown, it is a word whose action takes place only at compile time on the host. Note that many tokens (e.g., the **`ELIT`** family and those taking in-line offsets) have multiple forms reflecting possible compiler optimisations; only the basic form is shown here. The notation `<+addr>` (and variations thereof) represents an address offset within the module being compiled.

The first paragraph of the semantic description contains a stack notation for each stack affected by execution of the word. The remaining paragraphs contain a text description of the behaviour of the word. A description of the notation used is given in Section 2.4; a list of OTA-standard data type notation is given in Table 2, with Forth-specific notations in Table 3.

The control-flow stack (see Section 3.9, "Control Structures") *stack-id* is C:, 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.

Only the basic functionality is indicated for words taken from ANS Forth (designated with F, as described above); complete specifications for these words may be found in the standard. In addition, for OTA-specific words that correspond directly to tokens, one should consult the token entries for definitive behaviour.

In general, exception handling in the terminals is defined in the token descriptions. This section only documents compiler-level exception handling.

2.7 Word Behaviours

The description of each word explains its behaviour (semantics), which may be context-dependent. Some words in OTA are executable on both the host and target, in various circumstances—in which case, there can be more than one behaviour, and this behaviour may depend upon the state of the host. The behaviour(s) for each such word are described, as applicable, for:

Compiling: Describes an action taken on the host to modify a target image, within a colon definition (i.e., between `:` and `;`).

***name* Execution:** Describes the behaviour of *name* as executed on the target, where *name* is an instance of a class of words created by a defining word (see Section 3.7).

Interpreting: Describes an action taken on the host (specified either from the source or keyboard), which may or may not modify the target image.

Interacting: Describes behaviour when executed from an interactive development environment; used only when that behaviour differs from the run-time behaviour (below).

Run-time: Describes the behaviour as executed on the target (perhaps as a result of a compiling behaviour that constructed a program structure).

While many words (such as defining words and compiler directives) possess specific compiling behaviours, the default compilation behaviour of a word is to append its *execution behaviour* to the current definition. Separate behaviours in different modes will be shown where they differ.

Words with the C indicator will be executed (i.e., will perform their behaviour) when encountered in compiling mode. In Forth, these are known as *immediate* words. If execution of such a word will cause some run-time action in the target word being compiled, this is shown as a separate run-time behaviour.

("Compiling," in this context, applies both to direct compilation of VM implementations and to generation of tokens for the Virtual Machine.)

3 Compiler Functions

This section describes requirements for Forth compilers used in Open Terminal Architecture. There are actually two basic Forth compiler types: compilers used to produce VM implementations (called *cross-compilers*), and compilers used to produce token modules (including TRSs, token libraries, and applications). Although these have much in common with conventional Forth compilers, and much in common with each other, each has special requirements and issues that must be defined. This document covers only features that are common to both cross-compilers and token compilers, with emphasis on token equivalences. Features specific to individual VM cross-compilers are described in separate, product-specific documentation.

The goal of this document is to provide sufficient explanation of the requirements for OTA-specific compilers such that programmers implementing them, and programmers using them, may share a common set of expectations.

3.1 OTA Compiler Principles

OTA Forth compilers operate on the assumption that there are two computers involved: a *host*, on which the compiler runs and which may support other development functions for debugging, etc.; and a *target*, which may be a payment terminal (in the case of cross-compilers) and which is the Virtual Machine (in the case of a token compiler). Even in the case of a target which is physically the same as (or compatible with) the host (e.g., the Token Interactive Debugging Environment, or TIDE), the concept of logical separation between host and target is maintained.

The separation between host and target is manifested in several areas:

1. *Memory and address space.* There are a number of different address spaces: the host's own local memory, which is not normally accessible in an OTA compiler; the various regions of target memory (code space, and initialised and uninitialised data space); and a region where the host is maintaining pointers into the target's memory image. Except in special circumstances, when working with an OTA compiler, one is working in target data space.
2. *Command set.* Most Forth compilers are written in Forth. Consequently, the host provides locally executable versions of most Forth commands (or *words*), which are also present in a VM implementation. In order to distinguish between the host and target versions of these words, they are maintained in separately searchable word lists. Commands are provided to select among these word lists. The context in which a word may be accessed (either for execution or for compiling a reference to it) is called its *scope*. The default arrangement is that one is accessing the target versions of these words. Scopes are discussed further in Section 3.2.
3. *Compiler words and directives.* There are actually two compilers involved in a Forth-

based OTA compiler. The most visible compiler is the one being used to construct the target program, containing defining words (such as **:** and **;**, as well as **CONSTANT**, **VARIABLE**, etc.), flow-of-control words (such as **IF**, **THEN**, **BEGIN**, **DO**, **LOOP**, etc.), and words for managing data space (such as **,**, **ALLOT**, etc.) and other compiler words. However, an underlying compiler on the host was used to build the OTA token compiler or cross-compiler. This compiler may be extended, in order to provide special compiling or defining capabilities. For this reason, it is accessible via special scope selectors, as are the host versions of the common Forth commands.

It is not the purpose of this section to mandate any particular implementation of these functions. However, in order to facilitate movement of both source and programmers among OTA compilers, standardised compiler control and management functions have been defined which are mandatory for OTA token compilers and cross-compilers.

A Forth compiler may contain an *assembler*, which provides direct access to the native instruction set of the CPU in a terminal (in the case of a cross-compiler) or of the VM (the native instruction set of the VM is tokens). Other requirements include the ability to extend the underlying host system to modify or extend features of the target compiler itself (i.e., compiler extensibility).

The compiler can be considered to operate in one of two states: *interpreting* or *compiling*. The programmer may add new words that act in either of these states. Also, new words may be added not only to the target (the usual case), but also to the host, to provide additional support functions for the compiler.

Interactive development environments support a third state, called *interacting*. This means invoking words from the development environment's command line that will be executed in the target linked to that development environment for testing purposes (for instance, a target terminal linked to the OTA cross-compiler's host environment).

3.2 Compiler Scoping

The *scope* of a word may be defined as the context in which it may be used. Certain words may be available only in some of these states, and a few may have different actions in different states. The single-letter designators listed in Section 2.6 (page 2–11) identify the scopes in which each word may be used.

3.2.1 Specifying a Scope

Table 5 lists the compiler directives that select specific scopes. These control both access to previously defined words and also the scope(s) in which a new colon definition will be accessible. To define a word that will have different behaviours in more than one scope, it is necessary to define it in each scope to which it is relevant.

Table 5: Compiler directives that specify scope

Directive	Use
ASSEMBLER	To support programming using the native instruction set of the CPU in a terminal (in the case of a cross-compiler) or of the VM (the native instruction set of the VM is tokens).
COMPILER	To construct program structures (e.g., loops and conditionals) and perform other compile-time activities inside colon definitions.
HOST	To construct **COMPILER** or **INTERPRETER** definitions.
INTERPRETER	To build and modify data structures in the target image.
TARGET	To build the definitions that constitute the target program being compiled. This is the default scope.

TARGET is the default scope; if you select a different scope, you must reassert **TARGET**.

3.2.2 Effects of Scoping on Colon Definitions

The behaviour of words defined in each of these scopes is different for colon definitions and for data objects, and also depends upon the state (compiling or interpreting) in which they are invoked.

The default behaviour of colon definitions defined in **HOST**, **INTERPRETER**, or **TARGET** when they are invoked in the compiling state is to compile a reference to the word. The default behaviour of words defined in **COMPILER** is to execute, the consequence of which is usually to modify the definition being compiled in some way.

The default behaviour of colon definitions, when they are invoked in the interpreting state, is to be executed. However, **COMPILER** words may not be invoked in the interpreting state, and **TARGET** words may only be executed in the special interactive state supported by interactive development environments.

The accessibility of colon definitions defined in various scopes when they are invoked in interpreting and compiling states is shown in Table 6.

Table 6: Scopes in which colon definitions are accessible

Defined in	Interpreting scopes	Compiling scopes
A	A	A (used to define macros)
C	Not allowed	T
H	H, I, C, A	H, I, C, A
I	T	I
T	Not allowed	T

In the word definitions in this document, special state-dependent behaviours, if any, will be described where they differ from the default behaviour, which is that displayed when interpreting (for **HOST** and **INTERPRETER** words), compiling (for **COMPILER** words), or when

the VM is executing the word (for **TARGET** words).

3.2.3 Effects of Scoping on Data Object Defining Words

Defining words other than **:** (colon) are used to build data structures of various kinds, with various characteristic behaviours. Many such words are included in OTA, and it is also possible for programmers to construct new ones, if desired.

Normally, an OTA programmer is primarily concerned with building data structures for the target system; therefore, the dominant use of defining words is in the **TARGET** scope while in interpreting state. You may also build data objects in **HOST** that may be used in all scopes except **TARGET**; such objects might, for example, be used to control the compiling process.

Table 7: Scopes in which data objects are accessible

Defined in	Interpreting scopes	Compiling scopes
A	Not appropriate	Not appropriate
C	Not appropriate	Not appropriate
H	H, I, C, A	H, I, C, A
I	Not appropriate	Not appropriate
T	T (see actions in Table 8)	T

Data objects fall into three classes:

1. *IDATA objects:* Objects in initialised data memory (e.g., words defined by **CREATE**, **VALUE**, etc., including most user-defined words made with **CREATE … DOES>**).
2. *UDATA objects:* Objects in uninitialised data memory (e.g., words defined by **VARIABLE**, **BUFFER:**, etc.).
3. *Constants:* Words defined by **CONSTANT**, **2CONSTANT**, and **USER**, as well as database fields and **SOCKET**s.

Unlike target colon definitions, target data objects may be invoked in interpreting state. However, they may not exhibit their defined target behaviour, since that is available only in the target (or in interacting state). The default behaviours of each class of data objects when interpreted in **TARGET** scope is described in Table 8, along with usage restrictions. For objects defined within the VM, *logical address* is an actual target address. For objects defined in a module, *logical address* is an offset within the relevant data space.

Table 8: Interpreting behaviour of data objects

Usage:	IDATA	UDATA	Constants
Executing object name returns:	logical address	logical address	value
Can use object name with **,** (comma) or **C,** :	no	no	yes
Can use interpretively with data access operators (**@**, **!**, etc.):	yes	no	no

3.2.4 Scope Specification

The words in this section select the current compiler scope. See Section 3.2.2 and Section 3.2.3 for availability of words defined in each scope listed in this section.

ASSEMBLER |

(—)

Select **ASSEMBLER** scope (on compilers that support an assembler). For a token compiler, an assembler provides direct access to tokens.

COMPILER |

(—)

Select **COMPILER** scope. Words defined in this scope will be available for execution only while compiling target definitions.

Examples of **COMPILER** words include **IF**, **ELSE**, **THEN**, **BEGIN**, **UNTIL**, **DO**, **LOOP**, and similar words.

HOST |

(—)

Select **HOST** scope. **HOST** words are used to extend the host system to provide support for token compiling or cross-compiling. Since no assumptions should be made about the implementation of the compiler, the only way to return to target compilation is by the **TARGET** directive.

INTERPRETER |

(—)

Select **INTERPRETER** scope. Add the following definitions to the host environment's interpreter.

INTERPRETER words are used to create target definitions. Examples include **CREATE**, **VARIABLE**, etc.

TARGET |

(—)

Select **TARGET** scope. This is the default state of the compiler. If you select another scope, you must re-assert **TARGET** before producing any more target definitions.

Target colon definitions are available for interactive execution and testing only on development platforms that support interacting mode. Target data objects are executable in interpreting mode, according to the guidelines given in Section 3.2.3.

3.3 Compiler Directives

(paren F,H,C,I

("ccc<paren>" —)

Parse *ccc* delimited by **)** (right parenthesis). Used to begin a comment that extends to the next "**)**".

(will behave the same in all states.

If the end of the parse area is reached before a right parenthesis is found in a source file, refill the input buffer from the next line of the file and resume parsing from the start of the input buffer, repeating this process until either a right parenthesis is found or the end of the file is reached.

INCLUDE H,I

*(i*x "<spaces>name" — j*x)*

Open the file *name* and make it the input source. Other stack effects are due to the words included.

Save the input source specification. Repeat until end of file: read a line from the file, fill the input buffer from the contents of that line, and begin interpretation from the start of the input buffer. When the end of the file is reached, close the file and restore the input source specification to its saved value.

An ambiguous condition exists if the named file cannot be opened, if an I/O exception occurs reading the file, or if an I/O exception occurs while closing the file. When an ambiguous condition exists, the status (open or closed) of any files that were being interpreted is implementation-defined.

MARKER F,H,I

("<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution behaviour defined below.

name Execution:*(—)*

Restore all dictionary-allocation and search-order pointers to the state they had just prior to the definition of *name*. Remove the definition of *name* and all subsequent definitions. Restoration of any structures still existing that could refer to deleted definitions or deallocated data space is not necessarily provided. No other contextual information such as numeric base is affected.

[DEFINED] bracket-defined H,C,I

("<spaces>name" — flag)

Find the target word *name*. If the word is not found, return false *flag*. If the word is found, return true *flag*.

[ELSE] bracket-else F,H,C,I

("<spaces>name ... " —)

Skipping leading spaces, parse and discard space-delimited words from the

parse area, including nested occurrences of **[IF]** ... **[THEN]** and **[IF]** ... **[ELSE]** ... **[THEN]**, until the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, make an attempt to refill it from the current input source. **[ELSE]** is an immediate word.

[IF] bracket-if F,H,C,I

(flag | flag "<spaces>name ... " —)

If *flag* is true, do nothing. Otherwise, skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of **[IF]** ... **[THEN]** and **[IF]** ... **[ELSE]** ... **[THEN]**, until either the word **[ELSE]** or the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, make an attempt to refill it from the current input source. **[IF]** is an immediate word.

An ambiguous condition exists if **[IF]** is **POSTPONE**d, or if the end of the input buffer is reached and cannot be refilled before the terminating **[ELSE]** or **[THEN]** is parsed.

[THEN] bracket-then F,H,C,I

(—)

Do nothing. **[THEN]** is an immediate word.

[THEN] must exist in the source in order to be parsed by **[IF]** or **[ELSE]** and, thus, mark the end of the scope of these words. It has no function by itself.

[UNDEFINED] bracket-undefined H,C,I

("<spaces>name" — flag)

Find the target word *name*. If the word is found, return false *flag*. If the word is not found, return true *flag*.

\ backslash F,H,C,I

("ccc<eol>" —)

Parse and discard the remainder of the parse area. \ is an immediate word.

Used to begin a comment that extends to the end of the line.

3.4 Interpreter Control

The words in this section allow the programmer to have access to the compile-time dictionary, as well as control over the text interpreter.

3.4.1 Input Number Conversions

When a host Forth system encounters numbers in the input stream, they are automatically converted to binary. If the system is in compile mode—i.e., between a **:** (colon) and **;** (semicolon)—it will compile a reference to the number as a literal and, when the word being compiled is executed, that number will be pushed onto the stack. If the system is interpreting, the number will be pushed onto the host's stack directly.

All number conversions in Forth are controlled by the user variable **BASE**. The host system's **BASE** controls all input number conversions on the host; there is also a **BASE** (**USERVAR 0**) in the VM that controls number conversions performed by the target system. The words in this section are used both on the host and in target programs to control **BASE**. In each case, the requested mode will remain in effect until explicitly changed.

In addition, OTA input number conversion may be directed to convert a particular number in a particular base, specified by a prefix character from Table 9. Following such a conversion, **BASE** remains unchanged from its prior value. If the number is to be negative, the minus sign must *follow* the prefix and *precede* the most significant digit.

Table 9: Number conversion prefixes

Prefix	Conversion base	Example
%	Binary	**%10101010**
@	Octal	**@177**
#	Decimal	**#-13579**
$	Hex	**$FE00**

BINARY H,T (Sequence)

Interpreting: *(—)*

Set the number conversion base to 2 (binary). Used for controlling **BASE** on the host during compilation.

Run-time: *(—)*

Set the number conversion base in the target to 2 (binary).

The sequence may be implemented as **LIT2 LIT0 USERVAR STORE**.

DECIMAL F,H,T (Sequence)

Interpreting: *(—)*

Set the number conversion base to 10 (decimal). Used for controlling **BASE** on the host during compilation.

Run-time: *(—)*

Set the number conversion base on the target to 10 (decimal).

The sequence may be implemented as **LIT10 LIT0 USERVAR STORE**.

HEX F,H,T (Sequence)

Interpreting: *(—)*

Set the number conversion base to 16 (hexadecimal). Used for controlling **BASE** on the host during compilation.

Run-time: *(—)*

Set the number conversion base on the target to 16 (hexadecimal).

The sequence may be implemented as **SLIT 16 LIT0 USERVAR STORE**.

3.4.2 Other Interpreter Functions

' tick F,H,I

("<spaces>name" — xp)

Skip leading space delimiters. Parse *name* delimited by a space. Find *name* and return *xp*, the execution pointer for *name*. An ambiguous condition exists if *name* is not found.

The "Interpreting" columns in Table 6 (page 3–15) and Table 7 (page 3–16) show the applicable scopes in which words may be found.

>BODY to-body F,H,I

(C: xp — a-addr)

Returns *a-addr*, which is the data-field address corresponding to *xp*. An ambiguous condition exists if *xp* is not for a word defined via **CREATE**.

CHAR char F,H,I

("<spaces>name" — char)

Skip leading space delimiters. Parse *name* delimited by a space. Put the value of its first character onto the stack.

FIND F,H,I

(c-addr — c-addr 0 | xp 1 | xp -1)

Find the definition named in the counted string at *c-addr*. If the definition is not found, return *c-addr* and *zero*. If the definition is found, return its execution pointer *xp*. If the definition is immediate, also return 1; otherwise, also return -1.

The "Interpreting" columns in Table 6 (page 3–15) and Table 7 (page 3–16) show the applicable scopes in which words may be found.

IMMEDIATE F,H,I

(—)

Make the most recent host definition an *immediate word*. This marks the word so that, when it is encountered in subsequent colon definitions on the host, it is immediately executed (i.e., at compile time), rather than compiling a reference to be executed later (i.e., at run time). An ambiguous condition exists if the most recent definition does not have a name or was not defined in **COMPILER** scope.

RESTORE-INPUT F,H,I

(x_n .. x_1 n — flag)

Attempt to restore the input source specification to the state described by x_1–x_n and the number of stack items *n*. The *flag* is true if the input source specification cannot be restored. An ambiguous condition exists if the input source represented by the arguments is not the same as the current input source.

SAVE-INPUT F,H,I

(— x_n .. x_1 n)

x_1–x_n and the number of stack items *n* describe the current state of the input source specification for later use by **RESTORE-INPUT**.

WORD F,H

(char "<chars>ccc<char>" — c-addr)

Skip leading delimiters. Parse characters *ccc* delimited by *char*. An ambiguous condition exists if the length of the parsed string is greater than the implementation-defined length of a counted string.

c-addr is the address of a transient region containing the parsed word as a counted string. If the parse area was empty, or contained no characters other than the delimiter, the resulting string has a zero length. A space, not included in the length, follows the string.

3.5 Data Space Selectors

OTA compilers define three types of memory area, as shown in Table 10. These memory areas correspond to the normal set of code, initialised data, and uninitialised data.

Table 10: Memory areas in the VM

Memory Type	Selector	Contents	Compiler Access
Initialised RAM	**IDATA**	Data initialised at power-up	Read/Write
Uninitialised Data	**UDATA**	Uninitialised data. May also be used to define any other memory-mapped region.	None

The **IDATA** and **UDATA** selectors affect the actions of the following words: **,**, **C,**, **ALLOT**, **HERE**, and **ALIGN** (all described in Section 3.6), **CREATE** (see Section 3.7), and **ALIGNED** (see Section 4.2). These words act on the data area selected by the most recent use of **IDATA** or **UDATA**. The group of affected words is referred to as the *data allocation words*. When the compiler starts, the default is **IDATA**.

IDATA i-data I

(—)

Vector the data allocation words to refer to initialised data space. This is the default behaviour for these words.

UDATA u-data I

(—)

Vector the data allocation words to refer to uninitialised data space. An ambig-

uous condition exists if the compiler attempts to fetch from, or store into, uninitialised data space.

3.6 Data Space Management

The words in this section support the management of initialised and uninitialised data space.

A system guarantees that a region of data space allocated using **ALLOT**, **,** (comma), **C,** (c-comma), and **ALIGN** shall be contiguous with the last region allocated with one of the above words, unless the restrictions in the following paragraphs apply. The data-space pointer **HERE** always identifies the beginning of the next data-space region to be allocated. As successive allocations are made, the data-space pointer increases. A program may perform address arithmetic within contiguously allocated regions. The last region of data space allocated by using the above operators may be released by allocating a corresponding, negatively sized region using **ALLOT**—subject to the restrictions of the following paragraph.

CREATE establishes the beginning of a contiguous region of data space, whose starting address is returned by the **CREATE**d definition. This region is terminated by compiling the next definition.

, comma F,H,I

(x —)

Reserve one cell of the currently selected data space and store *x* in the cell. The data space pointer shall be adjusted, as necessary, to ensure that the data will be at a cell-aligned address. This word may not be used to store pointers, due to relocation requirements (see Vol. 1, Section 6.3). An ambiguous condition exists if the **UDATA** selector is current.

," comma-quote H,I

("ccc<quote>" —)

Parse *ccc* delimited by **"** (double-quote). Compile a counted ASCII string containing *ccc* in initialised data space.

,BCD comma-b-c-d I

(u —)

Compile *u* in **TLV-N** format (see Table 14), starting at the next location in initialised data space. This definition compiles a variable number of bytes with, at most, one leading pad of zero if the number of digits is odd.

,BN comma-b-n I

(u —)

Compile *u* in **TLV-BN** format (see Table 14), starting at the next location in initialised data space. This definition compiles one to four bytes, depending on the value of *u*.

A" a-quote I

("ccc<quote>" —)

Parse *ccc* delimited by **"** (double-quote). Compile the string in **TLV-AN** or **TLV-ANS** format (see Table 14), starting at the next location in initialised data space.

ALIGN F,H,I

(—)

If the pointer in the currently selected data-space is not aligned, **ALLOT** enough space to align it.

ALLOT F,H,I

(num —)

Reserve *num* address units of the currently selected data space, and modify the value of the data space pointer. If *num* is less than zero, release |*num*| address units of data space. If *num* is zero, leave the data space pointer unchanged.

C, c-comma F,H,I

(char —)

Reserve space for one character in the currently selected data space, and store *char* there. An ambiguous condition exists if the **UDATA** selector is current.

CELL H,T **LIT4**

(— num)

Return *num*, the size in address units of a cell. In OTA, this value is 4.

CN" c-n-quote I

("ccc<quote>" —)

Parse *ccc* delimited by **"** (double-quote). Compile the string in **TLV-CN** format (see Table 14), starting at the next location in initialised data space. The number is formatted with each character in the input string occupying a 4-bit nibble in the output string, according to the current number conversion **BASE**. Any **BASE** higher than 16 is not allowed. If the input string has an odd number of characters, the last nibble of the output string will be a hexadecimal F.

CODE, code-comma I

(x —)

Reserve one cell of initialised data space and store *x* in the cell. The address will be marked as a pointer to code space for relocation.

CODE! code-store I

(x a-addr —)

Store *x* at *a-addr* in initialised data space. Mark *a-addr* as a pointer to code space.

DATA! data-store I

(x a-addr —)

Store *x* at *a-addr* in initialised data space. Mark *a-addr* as a pointer to initialised data space.

DATA, data-comma I

(x —)

Reserve one cell of initialised data space and store *x* in the cell. The address will be marked as a pointer to initialised data space for relocation.

HERE F,H,I

(— addr)

Return *addr,* the current value of the pointer in the currently selected data space.

UDATA! u-data-store I

(x a-addr —)

Store *x* at *a-addr* in initialised data space. Mark *a-addr* as a pointer to uninitialised data space.

UDATA, u-data-comma I

(x —)

Reserve one cell of initialised data space and store *x* in the cell. The address will be marked as a pointer to uninitialised data space for relocation.

3.7 Defining Words

The words in this section all create new definitions—except **DOES>**, which assigns a behaviour to a new class of definitions.

: colon F,H,I **PROC**

(C:"*<spaces>name" — colon-sys)*

Parse *name.* Create a definition for *name.* Enter compiling state and start the current definition, producing *colon-sys.*

The execution behaviour of *name* will be terminated by the words compiled into the body of the definition. The current definition shall not be available in the dictionary until it is ended.

name Execution:*(i*x — j*x) (* R: *— nest-sys)*

Execute the definition *name*. The stack effects *i*x* and *j*x* represent arguments to, and results from, *name*.

When *name* begins executing, implementation-dependent *nest-sys* information is pushed on to the return stack. It will be removed by the run-time behaviour of **;** (semi-colon) or **DOES>**.

; semi-colon F,C,H,T **ENDPROC**

Compiling: *(* C: *colon-sys* —)

Append the run-time behaviour below to the current definition. End the current definition, allow it to be found in the dictionary, and enter interpretation state, consuming *colon-sys.*

Run-time: *(—) (* R: *nest-sys —)*

Return to the calling definition (specified by *nest-sys*) by performing a procedure equivalent to **EXIT**.

2CONSTANT two-constant F,H,T (Sequence)

(x_1 x_2 "<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution behaviour defined below.

name's values are not in accessible data space, and cannot be changed.

name Execution:*(— x_1 x_2)*

Place cell pair x_1 x_2 on the stack.

The sequence may be implemented as **ELIT** <x_1> **ELIT** <x_2>.

2VARIABLE two-variable F,H,T **ELITU** <+addr>

("<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* containing a pointer to its associated data field in uninitialised data space. Reserve two consecutive cells of uninitialised data space. The space between these cells and any previously allocated buffer is not accessible.

name is referred to as a "two-variable."

name Execution:*(— a-addr)*

Return *a-addr*, the address of the first (lowest address) cell of two consecutive cells in uninitialised data space (reserved by **2VARIABLE** when it defined *name*).

BUFFER: buffer-colon I **ELITU** <+addr>

(len "<spaces>name" —)

Parse *name*. Create a definition for *name* containing a pointer to its associated data field in uninitialised data space. Allot *len* bytes of uninitialised data space in *name*'s data field.

name Execution:*(— addr*)

Return *addr*, the address of *name*'s data field. For buffers defined in token programs, the address is aligned, regardless of *len*. For buffers defined in virtual machine implementations, the address is aligned only if *len* is aligned (multiple of four). If the address is aligned, the space between this *addr* and any previously allocated buffer is not accessible.

CODE F,H,I **PROC**

("<spaces>name" —)

Select the **ASSEMBLER** scope (see Section 3.2) for access purposes. Skip leading space delimiters. Parse *name* delimited by a space. Create a **TOKEN** definition for *name,* called a *code definition*, with the execution behaviour defined below.

Subsequent characters in the parse area may represent VM tokens (in a token compiler), or CPU-dependent instructions (in a cross-compiler). These are processed in an implementation-defined manner.

name Execution:*(i*x — j*x)*

Execute the token sequence that was generated following **CODE**.

CONSTANT F,H,I **ELIT** <x>

(x "<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution behaviour defined below.

name's value is not in accessible data space, and cannot be changed.

name Execution:*(— x)*

Place the value *x* on the stack.

CREATE F,H,I (Sequence)

("<spaces>name" —)

Parse *name.* Create a definition for *name* containing a pointer to its associated field in the currently selected data space. **CREATE** does not allocate any data space in *name*'s data field; it is normally followed by **ALLOT**, **,** (comma), or **C,** (c-comma).

When **CREATE** is used to define a new instance *name*, it may produce the token sequence:

DOCREATE or **EDOCREATE** <data-addr>

where *data-addr* is an offset (16-bit for **DOCREATE**, 32-bit for **EDOCREATE**) pointing to the instance's initialised data space. This is required only if the instance will be exported (callable by external modules).

Subsequent references to *name* within this module will produce a token sequence that may be implemented as:

ELITD <+addr>

where *addr* is the offset to the instance's data space address.

name Execution:*(— a-addr)*

Return *a-addr,* the address of *name*'s data field in the data space that was current when **CREATE** executed. The execution behaviour of *name* may be extended in the compiler by using **DOES>**.

DOES> does F,C,H (Sequence)

Compiling: *(C: colon-sys$_1$ — colon-sys$_2$)*

Append the run-time behaviour below to the current definition. Consume *colon-sys$_1$* and produce *colon-sys$_2$*. Append the initiation behaviour given below to the current definition.

Usage: `: <class> <compilation-behaviour>`
`DOES> <instance-behaviour> ;`

The definition above creates a class-defining word *class* in the host, and a token stream in the target representing the *instance-behaviour*.

This word may only be used in **INTERPRETER** or **HOST** scope (see Section 3.2). In the **INTERPRETER** scope only, **DOES>** switches to the **TARGET** scope for access purposes, and the *instance-behaviour* following may only be executed on the target.

Run-time: *(—) (* R: *nest-sys$_1$)*

Replace the execution behaviour of the most-recent definition, referred as *name*, with the name execution behaviour given below. Return control to the calling definition specified by *nest-sys$_1$*.

When *class* (above) is used by a token compiler to define a new instance *name*, it will produce a token sequence that may be implemented as:

DOCLASS <data-addr> <code-addr>
or
EDOCLASS <data-addr> <code-addr>

where *data-addr* is an offset pointing to the instance's initialised data space, and *code-addr* is an offset pointing to the instance behaviour. Both offsets are 16-bit (for **DOCLASS**) or 32-bit (for **EDOCLASS**). Subsequent references to *name* will produce a token sequence that may be implemented as:

CALL <+addr>

where *addr* is the offset to the instance's **DOCLASS** or **EDOCLASS**.

name Execution: *(i*x — j*x) (R: — nest-sys$_2$)*

Save implementation-dependent information *nest-sys$_2$* about the calling definition. Place *name*'s data-space address on the stack. The stack effects *i*x* represent possible arguments to *name*. The result is that, when the programmed instance behaviour begins, the address of the instance's data space will be on the stack. Then execute the instance behaviour associated with *name*.

END-CODE F,H,A **ENDPROC**

(—)

Terminate a **CODE** definition and reset the scope to **TARGET** (see Section 3.2; see also **CODE**).

EQU e-q-u H **LIT** <x>

(x "<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create an **INTERPRETER** definition for *name*, with the execution behaviour defined below.

Use of **EQU** to define *name* does not affect the target image. If *name* is referenced in a **TARGET** colon definition, the result will be a compiled literal giving *name*'s value.

An **EQU** may not be **EXPORT**ed.

name Execution: *(— x)*

Place the value *x* on the stack.

USER H,I **LIT** <num> **USERVAR**

(num "<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* whose token representation selects task-specific cell *num* (values 0–15). An ambiguous condition exists if *num* is not within the range 0–15.

name is referred to as a "user variable" (see Volume 1 of this specification).

name Execution: *(— a-addr)*

Return *a-addr*, the address of the cell in data space reserved by **USER** when it defined *name*.

VALUE F,H,I **ELITD** <+addr> <method>

(x "<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* whose token representation is a pointer to a reserved cell in initialised data space with an initial value of *x*.

name is referred to as a "value."

When *name* is preceded by **TO** in a definition passed to a token compiler, the compiled token stream shall provide **STORE** as the *method;* otherwise, **FETCH** shall be compiled as the *method*.

name Execution:

Access: *(— x)*

Place *x* on the stack. The value of *x* is that which was given when *name* was created; until the phrase *x* **TO** *name* is executed, causing a new value of *x* to be associated with *name*.

Store: *(x —)*

Store *x* in *name*.

VARIABLE F,H,I **ELITU** <+addr>

("<spaces>name" —)

Skip leading space delimiters. Parse *name* delimited by a space. Reserve one cell of uninitialised data space; the space between this cell and any previously allocated buffer is not accessible. Create a definition for *name* whose token representation is a pointer to the reserved data cell.

name is referred to as a "variable."

name Execution: *(— a-addr)*

Return *a-addr*, the address of the cell in uninitialised data space that was reserved by **VARIABLE** when it defined *name*.

3.8 Dictionary Management

The words in this section provide access to, and control over, the compilation process.

[left-bracket F,C,H

(—)

Enter interpretation state. **[** is an immediate word.

['] bracket-tick F,C,H **ELITC** <+addr>

Compiling: *("<spaces>name" —)*

Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. Append a token stream providing the run-time behaviour given below to the current definition.

The "Compiling scopes" columns in Table 6 (page 3–15) and Table 7 (page 3–16) show the applicable scopes in which words may be found.

An ambiguous condition exists if *name* is not found.

Run-time: *(— xp)*

Place *name*'s execution pointer *xp* on the stack.

[CHAR] bracket-char F,C,H **SLIT** <char>

Compiling: *("<spaces>name" —)*

Skip leading space delimiters. Parse *name* delimited by a space. Append a token stream providing the run-time behaviour given below to the current definition.

Run-time: *(— char)*

Place *char*, the value of the first character of *name*, on the stack.

] right-bracket F,H,I

Execution: *(—)*

Enter compilation state.

LITERAL F,C,H,C **ELIT** <num>

Compiling: *(x —)*

Append the run-time behaviour given below to the current definition. Used to compile a literal in a definition.

Depending on the size and sign of *x*, tokens other than **ELIT** may be compiled (e.g., **LITMINUS1**, **LIT0**, **SLIT** <x>, etc.).

Run-time: *(— x)*

Place *x* on the stack.

POSTPONE F,C,H

Compiling: *("<spaces>name" —)*

Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. Append the compilation behaviour of *name* to the current definition. An ambiguous condition exists if *name* is not found.

POSTPONE may not be used in **TARGET** definitions.

RECURSE F,H,T **CALL** <+addr>

*(i*x — j*x)*

Re-execute the current definition.

3.9 Control Structures

Most of the words in this section have both compilation and run-time behaviours. A few words that have only run-time behaviours (e.g., **I** and **J**) are included here, because they are relevant only in conjunction with other words in this section. For words that manage or use loop-control parameters, the stack comments reflect the usual, but not mandatory, practice of saving these parameters on the return stack.

+LOOP plus-loop F,C,H **RPLUSLOOP**

Compiling: *(* C: *do-sys —)*

Append the run-time behaviour given below to the current definition. Resolve the location given by *do-sys* to point to the words following **+LOOP**.

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

Add *num* to the loop index represented in *loop-sys$_1$*. If the loop index crossed the boundary between the loop limit minus one and the loop limit, discard the current loop-control parameters and continue execution immediately following the loop. Otherwise, continue execution at the beginning of the loop.

An ambiguous condition exists if the loop-control parameters are unavailable.

?DO question-do F,C,H **RQDO** <+addr>

Compiling: *(C: — do-sys)*

Put *do-sys* onto the control-flow stack. Append the run-time behaviour given below to the current definition. The behaviour are incomplete until resolved by a consumer of *do-sys*, such as **LOOP**.

Run-time: *(num_1 $num_2|u_1$ u_2 —) (* R: — | *loop-sys)*

If $num_1|u_1$ is equal to $num_2|u_2$, discard both values and continue execution at the location given by the consumer of *do-sys*. Otherwise, set up loop-control parameters with an index (whose initial value is $num_2|u_2$) and an upper limit ($num_1|u_1$), and continue executing immediately following **?DO**. Anything already on the return stack becomes unavailable until the loop-control parameters are discarded. An ambiguous condition exists if $num_1|u_1$ and $num_2|u_2$ are not both of the same type.

AGAIN F,C,H **BRA** <+addr>

Compiling: *(* C: *dest —)*

Append the run-time behaviour given below to the current definition, resolving the backward reference *dest*.

Run-time: *(—)*

Continue execution at the location specified by *dest*. If no other control-flow words are used within the code that is enclosed by the **AGAIN** and its preceding **BEGIN**, any program code after **AGAIN** will not be executed.

BEGIN F,C,H

Compiling: *(C: — dest)*

Put the next location for a transfer of control, *dest*, onto the control-flow stack. Append the run-time behaviour given below to the current definition.

Run-time: *(—)*

Continue execution.

CASE F,C,H

Compiling: *(C: — case-sys)*

Mark the start of a **CASE** ... **OF** ... **ENDOF** ... **ENDCASE** structure. Append the run-time behaviour given below to the current definition.

Run-time: *(—)*

Continue execution.

CS-PICK c-s-pick F,C,H

Compiling: *(C: $dest_u$ .. $orig_0|dest_0$ — $dest_u$ . . $orig_0|dest_0$ $dest_u$)*
(S: u —)

Remove *u*. Copy $dest_u$ to the top of the control-flow stack. An ambiguous condition exists if there are fewer than *u*+1 items, each of which shall be an *orig* or a *dest*, on the control-flow stack before **CS-PICK** is executed. If the control-

flow stack is implemented using the data stack, *u* shall be the top item on the data stack.

Run-time: *(—)*

Continue execution.

CS-ROLL c-s-roll F,C,H

Compiling: *(C: $orig_u|dest_u$ $orig_{u-1}|dest_{u-1}$ .. $orig_0|dest_0$ —*
$orig_{u-1}|dest_{u-1}$ .. $orig_0|dest_0$ $orig_u|destu$)

(S: u —)

Remove *u*. Rotate *u*+1 elements on top of the control-flow stack, such that $orig_u|dest_u$ is on top of the control-flow stack. An ambiguous condition exists if there are fewer than *u*+1 items—each of which shall be an *orig* or a *dest*—on the control-flow stack before **CS-ROLL** is executed. If the control-flow stack is implemented using the data stack, *u* shall be the top item on the data stack.

Run-time: *(—)*

Continue execution.

DO F,C,H **RDO** <+addr>

Compiling: *(* C: *— do-sys)*

Place *do-sys* onto the control-flow stack. Append the run-time behaviour given below to the current definition. The behaviour will be incomplete until resolved by a consumer of *do-sys*, such as **LOOP**.

Run-time: *(num_1 $num_2|u_1$ u_2 —) (* R: *— loop-sys)*

Set up loop-control parameters with an index (whose initial value is $num_2|u_2$) and an upper limit *($num_1|u_1$),*and continue executing immediately following **DO**. Anything already on the return stack becomes unavailable until the loop-control parameters are discarded. An ambiguous condition exists if $num_1|u_1$ and $num_2|u_2$ are not both of the same type.

ELSE F,C,H **BRA** <+addr>

Compiling: *(* C: *$orig_1$ — $orig_2$)*

Put the location of a new, unresolved, forward reference $orig_2$ onto the control-flow stack. Append the run-time behaviour given below to the current definition. The behaviour will be incomplete until $orig_2$ is resolved. Resolve the forward reference $orig_1$, using the location following the appended execution behaviour.

Run-time: *(—)*

Continue execution at the location given by the resolution of $orig_2$.

ENDCASE end-case F,C,H **DROP**

Compiling: *(* C: *case-sys —)*

Mark the end of the **CASE** ... **OF** ... **ENDOF** ... **ENDCASE** structure. Use *case-sys* to resolve the entire structure. Append the run-time behaviour given below

to the current definition.

Run-time: *(x —)*

Discard the case selector *x* and continue execution.

ENDOF end-of F,C,H **BRA** <+addr>

Compiling: *(* C: *case-sys$_1$ of-sys — case-sys$_2$)*

Mark the end of the **OF** ... **ENDOF** part of the **CASE** structure. The next location for a transfer of control resolves the reference given by *of-sys*. Append the run-time behaviour given below to the current definition. Replace *case-sys$_1$* with *case-sys$_2$* on the control-flow stack, to be resolved by **ENDCASE**.

Run-time: *(—)*

Continue execution at the location specified by the consumer of *case-sys$_2$*.

IF F,C,H **BZ** <+addr>

Compiling: *(* C: *— orig)*

Put the location of a new, unresolved, forward reference *orig* onto the control-flow stack. Append the run-time behaviour given below to the current definition. The behaviour will be incomplete until *orig* is resolved by **THEN** or **ELSE**.

Run-time: *(x —)*

If *x* equals zero, continue execution at the location specified by the resolution of *orig*.

I F,C,H,T **RI**

(— num) (R: *loop-sys — loop-sys)*

Return *num*, a copy of the current (innermost) loop index. Since loop-control information may be held on the return stack, the use of words that place data on the return stack while within a loop renders the loop index inaccessible.

J F,C,H,T **RJ**

(— num) (R: *loop-sys — loop-sys)*

Return *num*, a copy of the next outer loop index. Since loop-control information may be held on the return stack, the use of words that place data on the return stack while within a loop renders the loop index inaccessible.

LEAVE F,C,H,T **RLEAVE**

(—) (R: *loop-sys —)*

Discard the current loop-control parameters. An ambiguous condition exists if they are unavailable. Continue execution immediately following the innermost, syntactically enclosing **DO** ... **LOOP** or **DO** ... **+LOOP**.

LOOP F,C,H **RLOOP**

Compiling: *(* C: *do-sys —*)

Append the run-time behaviour given below to the current definition. Resolve the location given by *do-sys* to point to the words following **LOOP**.

Run-time: *(—) (* R: *loop-sys$_1$ — | loop-sys$_2$)*

Add one to the loop index. If the loop index is then equal to the loop limit, discard the loop parameters and continue execution immediately following the loop. Otherwise, continue execution at the beginning of the loop.

OF F,C,H **ROF** <+addr>

Compiling: *(* C: *— of-sys)*

Put *of-sys* onto the control-flow stack. Append the run-time behaviour given below to the current definition. The behaviour will be incomplete until resolved by a consumer of *of-sys*, such as **ENDOF**.

Run-time: *(num$_1$ num$_2$ — | num$_1$)*

If the two values on the stack are not equal, discard the top value and continue execution at the location specified by the consumer of *of-sys* (e.g., following the next **ENDOF**). Otherwise, discard both values and continue execution in line.

REPEAT F,C,H **BRA** <+addr>

Compiling: *(* C: *orig dest —)*

Append the run-time behaviour given below to the current definition, resolving the backward reference *dest.* Resolve the forward reference *orig*, using the location following the appended execution behaviour.

Run-time: *(—)*

Continue execution at the location given by *dest.*

THEN F,C,H

Compiling: *(C: orig —)*

Resolve the forward reference *orig* using the location of the execution behaviour. Append the run-time behaviour given below to the current definition.

Run-time: *(—)*

Continue execution.

UNTIL F,C,H **BZ** <+addr>

Compiling: *(* C: *dest —)*

Append the run-time behaviour given below to the current definition, resolving the backward reference *dest.*

Run-time: *(x —)*

If *x* equals zero, continue execution at the location given by *dest.*

WHILE F,C,H **BZ** <+addr>

Compiling: *(* C: *dest — orig dest)*

Put the location of a new, unresolved, forward reference *orig* onto the control-flow stack. Append the run-time behaviour given below to the current definition. The behaviour will be incomplete until *orig* and *dest* are resolved (e.g., by **REPEAT**).

Run-time: *(x —)*

If *x* equals zero, continue execution at the location specified by the resolution of *orig*.

3.10 Locals

A *local* is a data object which, when executed, returns its value; whose scope is limited to the definition in which it is declared; and whose use in a definition does not preclude re-entrancy or recursion.

During the compilation of a definition after **:** (colon) or **DOES>**, a program may begin naming locals. The process begins with the use of **LOCALS|**, and ends when the terminating **|** is encountered. The system keeps track of the names, order, and number of identifiers contained in the complete sequence.

3.10.1 Compilation behaviour

The compiler, upon encountering **LOCALS|**, takes the following actions:

1. Creates temporary dictionary entries for each of the identifier names that follow, such that each identifier will behave as a local. These temporary dictionary entries shall vanish at the end of the definition—denoted by **;** (semicolon) or **DOES>**—and hence may not be referenced outside the definition in which they occur. Local identifier names take precedence over any other named definitions, in case of name conflicts.
2. For each local identifier, the compiler generates an appropriate code sequence that does the following at execution time:

   (a) Allocates a storage resource adequate to contain the value of a local. The storage shall be allocated in a way that does not preclude re-entrancy or recursion in the definition using the local.

   (b) Initialises the value using the top item on the data stack. If more than one local is declared, the top item on the stack shall be moved into the first local identified, the next item shall be moved into the second, and so on.
3. Arranges that any of the legitimate methods of terminating execution of a definition—specifically **;** (semicolon), **DOES>**, or **EXIT**—will release the storage resource allocated for the locals, if any, declared in that definition. **CATCH** and **THROW** shall release such resources for all definitions whose execution is being terminated.

The storage resource may be the frame stack, or may be implemented in other ways, such as in registers. The storage resource shall not be the data stack. Use of locals does not restrict use of the data stack before or after the point of declaration.

Separate sets of locals may be declared in defining words before **DOES>** (for use on the host by the defining word), and after **DOES>** (for use by the word defined).

The VM shall support the declaration of at least eight locals in a definition.

3.10.2 Syntax restrictions

Immediate words defined on the host may use **LOCALS|** to implement syntaxes for local declarations, with the following restrictions:

- A program shall not compile any executable code into the current definition, between the time **LOCALS|** is executed to identify the first local for that definition and the terminating **|**.
- The position in program source at which the sequence of **LOCALS|** appears, referred to here as "the point at which locals are declared," shall not lie within the scope of any control structure.
- Locals shall not be declared until values previously placed on the frame stack within the definition have been removed.
- After a definition's locals have been declared, a program may place data on the frame stack. However, if this is done, locals shall not be accessed until those values have been removed from the frame stack.
- Words that return execution tokens—such as **'** (tick), **['']**, or **FIND**—shall not be used with local names.
- Locals may be accessed or updated within control structures, including do-loops.
- Local names shall not be referenced by **POSTPONE**.

3.10.3 Locals definitions

LOCALS| locals-bar F,C,H

Compiling: *("<spaces>name$_1$" "<spaces>name$_2$" ... "<spaces>name$_n$" "\|" —)*

Create up to eight local identifiers by repeatedly skipping leading spaces, parsing *name*, and creating a temporary definition, as described above. The list of locals to be defined is terminated by the **|** character. Append the run-time behaviour given below to the current definition.

Run-time: *(x_n ... x_2 x_1 —)*

Initialise up to eight local identifiers, as described above, each of which takes as its initial value the top stack item, removing it from the stack. Identifier *name$_1$* is initialised with x_1, identifier *name$_2$* with x_2, etc. When invoked, each local will return its value. The value of a local may be changed by using **TO**.

4 The OTA Wordset

The words in this section constitute the main set of OTA commands. Most of the words in this section have **TARGET** definitions and token equivalents. A few host words are included here that are used to build databases, TLVs, and other OTA-specific data structures. These are described here, rather than in the "Compiler Functions" section, because they are more easily explained and understood in the context of the words that access these data structures.

4.1 Stack Manipulation

4.1.1 Data Stack

The words in this section support direct manipulation of items on the data stack.

-ROT minus-rote H,T **MINUSROT**

(x_1 x_2 x_3 — x_3 x_1 x_2)

Place the top stack item under the next two.

2DROP two-drop F,H,T **TWODROP**

(x_1 x_2 —)

Drop cell pair x_1 x_2 from the stack.

2DUP two-dupe F,H,T **TWODUP**

(x_1 x_2 — x_1 x_2 x_1 x_2)

Duplicate cell pair x_1 x_2.

2OVER two-over F,H,T **TWOOVER**

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

Copy cell pair x_1 x_2 to the top of the stack.

2SWAP two-swap F,H,T **TWOSWAP**

(x_1 x_2 x_3 x_4 — x_3 x_4 x_1 x_2)

Exchange the top two cell pairs.

2ROT two-rote F,H,T **TWOROT**

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

Rotate the top three cell pairs on the stack, bringing cell pair x_1 x_2 to the top of the stack.

?DUP question-dupe F,H,T **QDUP**

(x — 0 | x x)

Duplicate *x*, if it is non-zero.

DROP F,H,T **DROP**

(x —)

Remove the top stack item.

DUP dupe F,H,T **DUP**

(x — x x)

Duplicate the top stack item.

NIP F,H,T **NIP**

(x_1 x_2 — x_2)

Remove the second item on the stack.

OVER F,H,T **OVER**

(x_1 x_2 — x_1 x_2 x_1)

Copy x_1 to the top of the stack.

PICK F,H,T **PICK**

(x_u .. x_0 u — x_u..x_0 x_u)

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

ROT rote F,H,T **ROT**

(x_1 x_2 x_3 — x_2 x_3 x_1)

Bring the third item on the stack to the top (i.e., “rotate” the top three items on the stack).

SWAP F,H,T **SWAP**

(x_1 x_2 — x_2 x_1)

Exchange the two top stack items.

TUCK F,H,T **TUCK**

(x_1 x_2 — x_2 x_1 x_2)

Place a copy of the top stack item beneath the second stack item.

4.1.2 Return Stack

The words in this section support limited use of the return stack for temporary storage. Since the return stack may be used for system functions in the Virtual Machine, it is essential that the rules given in Volume 1, Section 3.2.4 be observed.

>R to-r F,H,T **TOR**

(x —) (R: *— x)*

Move *x* from the data stack to the return stack.

2>R two-to-r F,H,T **TWOTOR**

(x_1 x_2 —) (R: *— x_1 x_2)*

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

2R> two-r-from F,H,T **TWORFROM**

(— x_1 x_2) (R: *x_1 x_2 —)*

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

2R@ two-r-fetch F,H,T **TWORFETCH**

(— x_1 x_2) (R: *x_1 x_2 — x_1 x_2)*

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

R> r-from F,H,T **RFROM**

(— x) (R: *x —)*

Move *x* from the return stack to the data stack.

R@ r-fetch F,H,T **RFETCH**

(— x) (R: *x — x)*

Copy *x* from the return stack to the data stack.

4.2 Data Access

The words in this section support transfers between the data stack and data storage.

! store F,H,T **STORE**

(x a-addr —)

Store *x* in the cell at *a-addr.*

This word may not be used interpretively in the host system to store pointers in initialised data space, due to relocation requirements (see Vol. 1, Section 6.3).

@ fetch F,H,T **FETCH**

(a-addr — x)

Return *x,* the value stored in the cell at *a-addr.*

2! two-store F,H,T **TWOSTORE**

(x_1 x_2 a-addr —)

Store two stack items x_1 x_2 into memory, with x_2 at *a-addr* and x_1 at the next consecutive cell.

This word may not be used interpretively in the host system to store pointers in initialised data space, due to relocation requirements (see Vol. 1, Section 6.3).

2@ two-fetch F,H,T **TWOFETCH**

(a-addr — x_1 x_2)

Fetch the contents of two cells from memory. x_2 is stored at *a-addr* and x_1 at the next consecutive cell.

ALIGNED F,H,T (Sequence)

(addr — a-addr)

Return *a-addr*, the first aligned address in the currently selected data space (see Section 3.5) greater than or equal to *addr*.

The sequence may be implemented as **SADDLIT3 NLIT 4 AND**.

C! c-store F,H,T **CSTORE**

(char c-addr —)

Store *char* at *c-addr*.

C@ c-fetch F,H,T **CFETCH**

(c-addr — char)

Fetch the character stored at *c-addr.*

CELLS F,H,T **SMULLIT 4**

(num_1 — num_2)

Return num_2, the size in address units of num_1 cells. Used for indexing into data by a given number of cells.

TO F,T (Sequence)

*(i*x "<spaces>name" —)*

Store *i*x* in *name*. An ambiguous condition exists if *name* was not defined by **VALUE** or a database field (see Section 4.14.1). In the case of database fields, **TO** invokes the field's *store* method. In this case, the stack arguments must comply with requirements of the field; otherwise, the stack effect is *(x —)*.

The sequence compiled depends upon the type of the data object specified by *name*.

4.3 Arithmetic

4.3.1 Single-Precision Arithmetic

The words in this section operate on single-cell integers. For the OTA Virtual Machine, this means 32-bit integers.

+ plus F,H,T **ADD**

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

Add $num_2|u_2$ to $num_1|u_1$, giving the sum $num_3|u_3$.

+! plus-store F,H,T **INCR**

(num|u a-addr —)

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

- minus F,H,T **SUB**

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

Subtract $num_2|u_2$ from $num_1|u_1$, giving the difference $num_3|u_3$.

***** star F,H,T **MUL**

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

Multiply $num_2|u_2$ by $num_1|u_1$, giving the product $num_3|u_3$.

***/** star-slash F,H,T (Sequence)

(num_1 num_2 num_3 — num_4)

Multiply num_1 by num_2, producing the intermediate double-cell result *d*. Divide *d* by num_3, giving the single-cell quotient num_5.

The sequence may be implemented as **TOR MMUL RFROM MSLMOD NIP**.

***/MOD** star-slash-mod F,H,T (Sequence)

(num_1 num_2 num_3 — num_4 num_5)

Multiply num_1 by num_2, producing the intermediate double-cell result *d*. Divide *d* by num_3, producing the single-cell remainder num_4 and the single-cell quotient num_5.

The sequence may be implemented as **TOR MMUL RFROM MSLMOD**.

/ slash F,H,T **DIV**

(num_1 num_2 — num_3)

Divide num_1 by num_2, giving the quotient num_3.

/MOD slash-mod F,H,T (Sequence)

(num_1 num_2 — num_3 num_4)

Divide num_1 by num_2, giving the single-cell remainder num_3 and the single-cell quotient num_4.

The sequence may be implemented as either:

TOR DUP SETLT RFROM MSLMOD

or

TWODUP MOD MINUSROT DIV.

1+ one-plus F,H,T **ADDLIT1**

($num_1|u_1$ — $num_2|u_2$)

Add one (1) to $num_1|u_1$, giving the sum $num_2|u_2$.

1- one-minus F,H,T **SUBLIT1**

($num_1|u_1$ — $num_2|u_2$)

Subtract one (1) from $num_1|u_1$, giving the difference $num_2|u_2$.

ABS abs F,H,T **ABS**

(num — u)

Return *u*, the absolute value of *num*.

C+! c-plus-store H,T (Sequence)

(num c-addr —)

Add *num* to the byte at *c-addr*. The sequence may be implemented as:
DUP CFETCH ROT ADD SWAP CSTORE

CELL+ cell-plus F,H,T **SADDLIT 4**

($a\text{-}addr_1$ — $a\text{-}addr_2$)

Add the size (in address units) of a cell to $a\text{-}addr_1$, giving $a\text{-}addr_2$.

CHAR+ char-plus F,H,T **ADDLIT1**

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

Add the size (in address units) of a character to $c\text{-}addr_1$, giving $c\text{-}addr_2$.

MAX F,H,T **MAX**

(num_1 num_2 — $num_1|num_2$)

Return the greater of num_1 and num_2.

MIN F,H,T **MIN**

(num_1 num_2 — $num_1|num_2$)

Return the lesser of num_1 and num_2.

MOD F,H,T **MOD**

(num_1 num_2 — num_3)

Divide num_1 by num_2, giving the single-cell remainder num_3.

NEGATE F,H,T **NEGATE**

(num_1 — num_2)

Negate num_1, giving its arithmetic inverse num_2.

U/ u-slash F,H,T **DIVU**

(u_1 u_2 — u_3)

Divide u_1 by u_2, giving the quotient u_3.

UMOD u-mod H,T **MODU**

(u_1 u_2 — u_3)

Divide u_1 by u_2, giving the single-cell remainder u_3.

4.3.2 Double-Number And Mixed-Precision Arithmetic

These words provide a limited set of integer procedures involving double-cell (64-bit) numbers.

D+ d-plus F,H,T **DADD**

(d_1 d_2|ud_1 ud_2 — d_3|ud_3)

Add d_2|ud_2 to d_1|ud_1, giving the sum d_3|ud_3.

D< d-less-than F,H,T **DCMPLT**

(d_1 d_2 — flag)

Return true *flag* if and only if d_1 is less than d_2.

DABS d-abs F,H,T (Sequence)

(d — ud)

Return *ud*, the absolute value of *d*.

The sequence may be implemented as:
DUP SETLT SBZ <addr+2> **DNEGATE**

DNEGATE d-negate F,H,T **DNEGATE**

(d_1 — d_2)

Return d_2, the arithmetic inverse of d_1.

M* m-star F,H,T **MMUL**

(num_1 num_2 — d)

Multiply num_1 by num_2, giving the signed, double-cell product *d*.

M/MOD m-slash-mod H,T **MSLMOD**

(d num_1 — num_2 num_3)

Divide *d* by num_1, giving the single-cell quotient num_3 and remainder num_2. Identical to **SM/REM**, but included here because this name is in common usage.

S>D s-to-d F,H,T (Sequence)

(num — d)

Convert *num* to the double-cell number *d* with the same numerical value.

The sequence may be implemented as **DUP SETLT**.

SM/REM s-m-slash-rem F,H,T **MSLMOD**

(d num_1 — num_2 num_3)

Divide *d* by num_1, giving the single-cell quotient num_3 and remainder num_2. Identical to **M/MOD**, but included here for ANS Forth compatibility.

UM* u-m-star F,H,T **MMULU**

(u_1 u_2 — ud)

Multiply u_2 by u_1, giving the double-cell product *ud*.

UM/MOD u-m-slash-mod F,H,T **MSLMODU**

(ud u_1 — u_2 u_3)

Divide *ud* by u_1, giving the remainder u_2 and the quotient u_3.

4.3.3 Logic

These words provide logical and shifting operations on single-cell (32-bit) operands.

2* two-star F,H,T **SHL**

(x_1 — x_2)

Return x_2, the result of shifting x_1 one bit toward the most-significant bit, filling the vacated, least-significant bit with zero.

2/ two-slash F,H,T (Sequence)

(x_1 — x_2)

Return x_2, the result of shifting x_1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

The sequence may be implemented as **LIT1 SHRN**.

AND F,H,T **AND**

(x_1 x_2 — x_3)

Return x_3, the bit-by-bit *and* of x_1 with x_2.

INVERT F,H,T (Sequence)

(x_1 — x_2)

Invert all bits of x_1, giving the bit-by-bit inverse x_2.

The sequence may be implemented as **NEGATE SUBLIT1**.

LSHIFT l-shift F,H,T **SHLN**

(x_1 u — x_2)

Perform a logical left shift of (u mod 32) bit-places on x_1, giving x_2. Put zero into the least-significant bits vacated by the shift.

NOT H,T **SETEQ**

(x — $flag$)

Return true *flag* if and only if x is equal to zero. **NOT** is a synonym for **0=** (defined for readability purposes), and is used to invert a logical condition.

OR F,H,T **OR**

(x_1 x_2 — x_3)

Return x_3, the bit-by-bit *inclusive or* of x_1 with x_2.

RSHIFT r-shift F,H,T **SHRNU**

(x_1 u — x_2)

Perform a logical right shift of (u mod 32) bit-places on x_1, giving x_2. Put zero into the most-significant bits vacated by the shift.

RSHIFTS r-shifts F,H,T **SHRN**

(x_1 u — x_2)

Shift x_1 arithmetically right by mod(u,32) places, propagating the most-significant bit, to give x_2.

WIDEN F,T **WIDEN**

(char — num)

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

XOR F,H,T **XOR**

(x_1 x_2 — x_3)

Return x_3, the bit-by-bit *exclusive or* of x_1 with x_2.

4.4 Relation and Comparison

These comparison operations deal with single-cell (32-bit) values.

< less-than F,H,T **CMPLT**

(num_1 num_2 — flag)

Return true *flag* if and only if num_1 is less than num_2.

<= less-than-or-equal F,H,T **CMPLE**

(num_1 num_2 — flag)

Return true *flag* if and only if num_1 is less than or equal to num_2.

<> not-equal F,H,T **CMPNE**

(x_1 x_2 — flag)

Return true *flag* if and only if x_1 is not, bit-for-bit, the same as x_2.

= equal F,H,T **CMPEQ**

(x_1 x_2 — flag)

Return true *flag* if and only if x_1 is equal to x_2.

> greater-than F,H,T **CMPGT**

(num_1 num_2 — flag)

Return true *flag* if and only if num_1 is greater than num_2.

>= greater-than-or-equal F,H,T **CMPGE**

(num_1 num_2 — flag)

Return *true* if and only if num_1 is greater than or equal to num_2.

0< zero-less-than F,H,T **SETLT**

(num — flag)

Return true *flag* if and only if *num* is less than zero.

0<= zero-less-or-equal H,T **SETLE**

(num — flag)

Return true *flag* if and only if *num* is less than, or equal to, zero.

0<> zero-not-equal F,H,T **SETNE**

(x — flag)

Return true *flag* if and only if *x* is not equal to zero.

0= zero-equal F,H,T **SETEQ**

(x — flag)

Return true *flag* if and only if *x* is equal to zero.

0> zero-greater F,H,T **SETGT**

(num — flag)

Return true *flag* if and only if *num* is greater than zero.

0>= zero-greater-or-equal H,T **SETGE**

(num — flag)

Return true *flag* if and only if *num* is greater than, or equal to, zero.

U< u-less-than F,H,T **CMPLTU**

(u_1 u_2 — flag)

Return true *flag* if and only if u_1 is less than u_2.

U<= u-less-than-or-equal H,T **CMPLEU**

(u_1 u_2 — flag)

Return true *flag* if and only if u_1 is less than, or equal to, u_2.

U> u-greater-than F,H,T **CMPGTU**

(u_1 u_2 — flag)

Return true *flag* if and only if u_1 is greater than u_2.

U>= u-greater-than-or-equalH,T **CMPGEU**

(u_1 u_2 — flag)

Return true *flag* if and only if u_1 is greater than, or equal to, u_2.

WITHIN F,H,T **WITHIN**

(num_1 num_2 num_3 | u_1 u_2 u_3 — flag)

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

4.5 Strings

Forth implementations commonly store strings internally as *counted strings*, which contain a length byte followed by that many bytes of data. External references to strings usually involve address and length parameters for the actual data passed on the stack.

OTA defines and uses three principal types of strings: ASCII strings, containing one character per byte of data; numeric strings, containing two digits per byte of data; and binary strings, containing sequences of data bytes whose interpretation is application specific. Numeric strings are further divided into two types: BCD strings, in which each four-bit nibble is a binary-coded decimal digit, and with at least one leading zero nibble if the number of significant digits is odd; and CN *(compressed numeric)* strings, in which each four-bit nibble is a digit, according to the current value of **BASE**, with at least one trailing F nibble (all bits set) if the number of significant digits is odd.

!BCD store-b-c-d T **BCDSTORE**

(u c-addr len —)

Store number *u* as a BCD string into the memory at *c-addr*, for *len* bytes **BASE** is not used in the conversion. Leading nibbles shall be filled with zeroes, if needed. The most significant part of the number shall be truncated if *len* is not sufficient to hold all the digits.

!BN store-b-n T **BNSTORE**

(u c-addr len —)

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

!CN store-c-n T **CNSTORE**

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

Convert the ASCII string at $c\text{-}addr_1$, for len_1 bytes, into a compressed number in the CN string at $c\text{-}addr_2$, for len_2 bytes. The number is formatted with each

character in the input string occupying a 4-bit nibble in the output string, according to the current number conversion **BASE**. Trailing nibbles shall be filled with Fs if needed. The number shall be truncated if len_2 is not sufficient to hold all the characters (len_2<[len_1+1]/2). A **DIGIT-TOO-LARGE** exception may be thrown if a character in the input string is not a number in the current **BASE**. Care must be taken when using this word in hex base, since any pad will appear as a valid digit.

-ZEROS minus-zeros T **MINUSZEROS**

(c-addr len_1 — c-addr len_2)

Eliminate trailing nulls. If len_1 is greater than zero, len_2 is equal to len_1 less the number of binary zero (null) bytes at the end of the string specified by *c-addr* len_1. If len_1 is zero, or if the entire string consists of nulls, len_2 is zero.

-TRAILING minus-trailing F,H,T **MINUSTRAILING**

(c-addr len_1 — c-addr len_2)

Eliminate trailing spaces. If len_1 is greater than zero, len_2 is equal to len_1 less the number of spaces at the end of the ASCII string specified by *c-addr* len_1. If len_1 is zero, or if the entire string consists of spaces, len_2 is zero.

+STRING plus-string T **PLUSSTRING**

(c-addr$_1$ len_1 c-addr$_2$ len_2 — c-addr$_2$ len_3)

Append the string at *c-addr*$_1$, for len_1 bytes, to the end of the string at *c-addr*$_2$, for len_2 bytes. Return the beginning of the destination string (*c-addr*$_2$) and the sum of the two lengths (len_3). It is the programmer's responsibility to ensure that there is room at the end of the destination string to hold both strings.

/STRING slash-string F,H,T **SLASHSTRING**

(c-addr$_1$ len_1 num — c-addr$_2$ len_2)

Adjust the ASCII string at *c-addr*$_1$ by *num* characters. The resulting ASCII string—specified by *c-addr*$_2$ len_2—begins at *c-addr*$_1$ plus *num* characters, and is len_1 minus *num* characters long.

@BCD fetch-b-c-d T **BCDFETCH**

(c-addr len — u)

Fetch number *u* from a BCD string at *c-addr* for *len* bytes. The number is formatted with each digit representing 4-bit nibbles in the input string. **BASE** is not used in the conversion. An **INVALID-NUMERIC-ARG** exception may be thrown, if any nibble is not a valid BCD digit. If the number in the input string has a value greater than 32 bits, the least-significant 32 bits shall be returned on the stack.

@BN fetch-b-n T **BNFETCH**

(c-addr len — u)

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

at that location, the least-significant 32 bits shall be returned on the stack.

@CN fetch-c-n T **CNFETCH**

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

Fetch an ASCII string to the temporary location *c-addr$_2$*, for *len$_2$* bytes, that represents the TLV Type 3 compressed number in the CN string at *c-addr$_1$* for *len$_1$* bytes (see Section 4.16.1.1). The number is formatted with each character of the output string representing a 4-bit nibble in the input string, according to the current number conversion **BASE**. The output string shall be terminated when a nibble with all bits set (F_H) or the end of the string is encountered. A **DIGIT-TOO-LARGE** exception may be thrown, if a nibble in the input string is not a number in the current **BASE**. The output string must be moved to a more-permanent location immediately. Care must be taken when using this word in hex base, since an F will be interpreted as a terminating nibble following the end of a valid CN string.

BLANK F,H,T (Sequence)

(c-addr len —)

If *len* is greater than zero, store the ASCII character value for space in *len* consecutive character positions, beginning at *c-addr*.

The sequence may be implemented as **SLIT 20$_H$ FILL**.

C+STRING c-plus-string T (Sequence)

(char c-addr len — c-addr len+1)

Append *char* to the string *c-addr len*.

The sequence may be implemented as:
TWODUP TWOTOR ADD CSTORE TWORFROM ADDLIT1

COMPARE F,H,T **COMPARE**

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

Compare the ASCII string specified by *c-addr$_1$ len$_1$* to the ASCII string specified by *c-addr$_2$ len$_2$*. The strings are compared character-by-character—beginning at the given addresses—up to the length of the shorter string, or until a difference is found. If the two strings are identical, *num* is zero. If the two strings are identical up to the length of the shorter string, *num* is -1 if *len$_1$* is less than *len$_2$*, and is 1 otherwise. If the two strings are not identical up to the length of the shorter string, *num* is -1 if the first non-matching character in the string specified by *c-addr$_1$ len$_1$* has a lesser numeric value than the corresponding character in the string specified by *c-addr$_2$ len$_2$*, and is 1 otherwise.

COUNT F,H,T **COUNT**

(c-addr$_1$ — c-addr$_2$ len)

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

ERASE F,H,T (Sequence)

(addr len —)

If *len* is greater than zero, clear all bits in each of *len* consecutive address units of memory, beginning at *addr*.

The sequence may be implemented as **`LIT0 FILL`**.

FILL F,H,T **FILL**

(c-addr len char —)

If *len* is greater than zero, store *char* in each of *len* consecutive characters of memory, beginning at *c-addr*.

HEX" hex-quote I,C,H (Sequence)

Interpretation: *("hex-digits<quote>" — c-addr len)*

Parse *hex-digits*, delimited by **"** (double-quote). Store the resulting binary string *c-addr len* at a temporary location. The maximum length of the temporary buffer is implementation dependent, but shall be large enough to store at least 32 characters of input. Subsequent uses of **S"** or **HEX"** may overwrite this temporary buffer.

Compilation: *("hex-digits<quote>" —)*

Parse *hex-digits*, delimited by **"** (double-quote). Assemble a binary string representing the *hex-digits*, ignoring any spaces. An abort will occur if a character is encountered that is not a space or a valid hex digit. If the result is an odd number of nibbles, a zero nibble shall be appended to the string. Append the run-time semantics given below to the current definition.

Run-time: *(— c-addr len)*

Return *c-addr* and *len*, describing a binary string assembled from the characters in *hex-digits*. Program validation will ensure that the returned string is not altered subsequently.

The sequence may be implemented as **`ELITD`** `<+addr>` **`SLIT`** `<u>`.

MOVE F,H,T **MOVE**

($addr_1$ $addr_2$ len —)

If *len* is greater than zero, copy the contents of *len* consecutive address units at $addr_1$, to the *len* consecutive address units at $addr_2$ (non-destructively, if the areas overlap). After **MOVE** completes, the *len* consecutive address units at $addr_2$ are the same as the *len* consecutive address units at $addr_1$ before the move.

S" s-quote F,H,C,I **STRLIT**

Interpretation: *("ccc<quote>" — c-addr len)*

Parse *ccc* delimited by **"** (double-quote). Store the resulting ASCII string *c-addr len* at a temporary location. The maximum size of the temporary buffer is implementation dependent, but shall be no fewer than 80 characters. Subsequent uses of **S"** or **HEX"** may overwrite this temporary buffer.

Compilation: *("ccc<quote>" —)*

Parse *ccc* delimited by **"** (double-quote). Append the run-time semantics given below to the current definition.

Run-time: *(— c-addr u)*

Return *c-addr* and *u*, describing an ASCII string consisting of the characters *ccc*. Program validation will ensure that a program does not alter the returned string.

SCAN H,T **SCAN**

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

Parse the ASCII string at $c\text{-}addr_1$, for len_1 bytes, for bytes containing *char*. $c\text{-}addr_2$ is the address of the byte where *char* is found, or is the end of the string (i.e., effectively discarding any string contents before *char*). len_2 is the length, in characters, of the remaining string at $c\text{-}addr_2$, and will be zero if *char* was not found.

SKIP H,T **SKIP**

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

Parse the ASCII string at $c\text{-}addr_1$, for len_1 bytes, skipping bytes that contain *char*. $c\text{-}addr_2$ is the address of the first byte that differs from *char*, or is the end of the string (i.e., effectively discarding any string contents before *char*). len_2 is the length, in characters, of the remaining string at $c\text{-}addr_2$, and will be zero if the string was completely filled with *char*.

4.6 Extensible Memory Management

Dynamic memory is provided by the OTA Virtual Machine 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. An automatic release of dynamically allocated memory also occurs when a module's execution is completed, limiting the effects of program failure to release memory cleanly.

This section lists the words that provide these Extensible Memory Management facilities.

CEXTEND c-extend T **CEXTEND**

(len — c-addr)

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

EXTEND T **EXTEND**

(len — a-addr)

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

RELEASE T **RELEASE**

(addr —)

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

4.7 Exception Handling

OTA uses the ANS Forth Exception Wordset (the words **CATCH** and **THROW**) for exception handling.

CATCH and **THROW** provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of procedure calls. It is similar in spirit to the *non-local return* mechanisms of other languages—such as C's `setjmp()` and `longjmp()`, and LISP's `CATCH` and `THROW`. In the Forth context, **THROW** may be described as a "multi-level **EXIT**," with **CATCH** marking a location to which a **THROW** may return.

A list of OTA-defined exception numbers, used as **THROW** codes, is found in Appendix C.

CATCH F,H,T **CATCH**

*(i*x xp — j*x 0 | i*x num)*

Push an exception frame on the exception stack, then call the procedure at *xp* (as with **EXECUTE**) in such a way that control can be transferred to a point just after **CATCH** if **THROW** is executed during the execution of *xp*.

If the execution of *xp* completes normally (i.e., the exception frame pushed by this **CATCH** is not popped by an execution of **THROW**), pop the exception frame and return zero on top of the data stack, above whatever stack items have been returned by the execution of *xp*. See **THROW** for definition of the return state otherwise.

THROW F,H,T **THROW**

*(k*x num — k*x | i*x num)*

If any bits of *num* are non-zero, pop the top exception frame from the exception stack, along with everything on the return stack above that frame. Then adjust the depths of all stacks so they are the same as the depths saved in the exception frame (i.e., *i* is the same number as the *i* in the input arguments to the corre-

sponding **CATCH**), put *num* on top of the data stack, and transfer control to a point just after the last executed **CATCH**. Restore the frame pointers to the values they had before the corresponding **CATCH**. In addition, if the word executed by **CATCH** causes the execution of **DOMODULE** or **CARD-MODULE**, restore the allocation of extensible memory to its condition at the time of execution of the **MODULE** word.

A list of **THROW** codes and their meanings may be found in Appendix C.

?THROW question-throw T **QTHROW**

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

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

4.8 Date, Time, and Timing Services

The words in this section provide access to the Virtual Machine's internal clock and calendar.

GET-MSECS get-m-secs T **GETMS**

(— u)

Return current system free-running milliseconds timer value. The timer value wraps from 2^{32}-1 to zero.

MS m-s F,H,T **MS**

(u —)

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

SET-TIME&DATE set-time-and-date F,H,T **SETTIME**

(u_1 u_2 u_3 u_4 u_5 u_6 *—)*

Set the current time and date. u_1 is the second {0...59}, u_2 is the minute {0...59}, u_3 is the hour {0...23}, u_4 is the day {1...31}, u_5 is the month {1...12}, and u_6 is the year {0...9999}. Some terminals may not be able to support this function. On these terminals, an **UNSUPPORTED-OPERATION** exception may be thrown.

TIME&DATE time-and-date F,H,T **GETTIME**

(— u_1 u_2 u_3 u_4 u_5 u_6 *)*

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

4.9 Generic Device Management

The words in this section provide generic support for I/O devices, as well as a specific method for routing output to a system printer or ticket printer. The generic I/O words include the concept of a *logical device*, which may be defined in the implementation to be a physical device, task, or other entity responsible for I/O procedures. The returned I/O result (*ior*) may contain an implementation-defined value, but in all cases zero must indicate *device ready* or *successful completion,* and a non-zero value may be interpreted as an error code.

Standard OTA device number assignments are shown in Table 11. The *device name* is the name of a constant that returns the given device number. Devices should be referred to by this name, rather than number, where possible. Additional device assignments, if any, are implementation defined. Device names for the extra serial ports, modems and printers are to be constructed as follows. For the serial ports whose device number ranges between 32 and 47 the name shall be **DEV-SER***x* where **DEV-SER***x* corresponds to device number *x-2+31* (*x*

Table 11: Device number assignments

Number	Device	Device Name
-1	Debug device	**DEV-DEBUG**
0	Primary keyboard	**DEV-KBD1**
1	Primary display	**DEV-DISPLAY1**
2	System printer	**DEV-PRINTER1**
3	Ticket printer or receipt printer	**DEV-PRINTER2**
4	Modem (primary serial port)	**DEV-MODEM**
5	ICC card reader 1	**DEV-ICC1**
6	ICC card reader 2	**DEV-ICC2**
7	Magnetic stripe device	**DEV-MAGSTRIPE**
8	Secondary keyboard	**DEV-KBD2**
9	Secondary display	**DEV-DISPLAY2**
10	Secondary serial port	**DEV-SER2**
11	First parallel port	**DEV-PAR1**
12	Second parallel port	**DEV-PAR2**
13	Power management device	**DEV-PWR**
14	Vending machine device	**DEV-VEND**
15-31	RFU	-
32-47	Serial Ports	**DEV-SER***x*
48-63	Modems	**DEV-MODEM***y*
64-79	Printers	**DEV-PRINTER***z*
80-	RFU	-

can take on any value from 3 to 18). Similarly the extra modems shall be named **DEV-MODEM***y* corresponding to device number *y-1+47* (y can take on any value from 2 to 17) and the extra printers shall be named **DEV-PRINTER***z* corresponding to device number *z-2+63* (z can take on any value from 3 to 18). E.g. **DEV-SER5** corresponds to device number 34, **DEV-MODEM12** corresponds to device number 58 and **DEV-PRINTER8** corresponds to device number 69.

4.9.1 Generic Device I/O

The words in this section all return *ior* result codes. A complete list of these codes is given in Appendix C.

OPEN T **DEVOPEN**

(dev — ior)

Open device *dev*. On buffered devices **OPEN** clears all data from the buffers. Ior **DEV-MUST-BE-OPENED** shall be returned by all device words applied on a device that is not open. All devices are closed by start-up of the terminal. Applying the **OPEN** word on a device that is already open, results in a no-op and an ior **DEV-ALREADY-OPEN**.

CLOSE T **DEVCLOSE**

(dev — ior)

Close the given device *dev*. After closing the device, the device is not accessible anymore and all device words applied on a closed device return ior **DEV-MUST-BE-OPENED**. Note that closing a modem device with an open connection implicitly performs a hangup of the modem.

TIMED-READ T **DEVTIMEDREAD**

(c-addr len_1 dev — c-addr len_2 ior)

Read a string of bytes from input device *dev*, returning a device-dependent *ior*. An *ior* of zero means success, and any other value is both device and implementation dependent. The size of an element within the string is device dependent. *c-addr* is the destination address for the string, and len_1 is its maximum length. On return, len_2 gives the actual length of the string that was read. If the requested number of elements are not received within the specified period of time, ior **DEV-TIMEOUT** is returned. A time-out of 0 will cause the word to return immediately. If the Virtual Machine has buffered enough data, the word will return immediately ior **SUCCESS**. The time-out may be set by using **IOCTRL**, as described in Appendix C to Volume 1 of these Specifications.

IOCTRL i-o-control T **DEVIOCTL**

(a-addr num fn dev — ior)

Perform **IOCTRL** function *fn* for device *dev*, with *num* cell-sized arguments in the array at *a-addr*. Individual operations are device dependent and are defined against supported devices in Appendix C to Volume 1. Exception **UNSUPPORTED-OPERATION** shall be thrown when in a particular implementation device *dev* is supported but function *fn* is not. It is mandatory for a VM to implement every **IOCTL** function that is supported by the

underlying hardware.. Device code assignments are shown in Table 11.

OUTPUT T **DEVOUTPUT**

(xp dev — ior)

Execute the procedure whose execution pointer is given by *xp*, with output being directed to device *dev*. On return from **OUTPUT**, the current output device is unaffected. *ior* is zero, if the procedure *xp* can be started on the device *dev*; or is an appropriate non-zero value from Appendix C, if it cannot. All exceptions arising from the execution of *xp*—those that are not trapped by a **CATCH** within the code of *xp*—cause immediate termination of *xp*, without an exception being thrown to any exception handler external to *xp*. *xp* is required to have no stack effect; if it does, terminal action is undefined.

READ T **DEVREAD**

(c-addr len dev — ior)

Read a string of *len* bytes from input device *dev*, returning a device-dependent *ior*. An *ior* of zero means success, and any other value is device dependent. The size of an element within the string is device dependent. For example, on a keyboard, each element is two bytes: an extension code in the first byte, and the key value in the second byte.

STATUS T **DEVSTATUS**

(dev — ior)

Return the status *ior* of the resource associated with device *dev*, where in the general case *ready* and *serviceable* is indicated by zero, and *not ready* is indicated by any other value. A specific device may return non-zero *ior* values that have significance for that device, as detailed in Appendix C. If the device is currently occupied because it has been selected by a previous execution of **OUTPUT**, **STATUS** will return the standard *ior* code for *device busy*, until execution of the procedure passed to **OUTPUT** completes.

WRITE T **DEVWRITE**

(addr len dev — ior)

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

4.9.2 Current Device I/O

Words in this section manage and perform I/O on the current input and output devices. User variable 1 (DEVOP) contains the current output device, and user variable 2 (DEVIP) contains the current input device. The current output device may be temporarily routed to the system printer or ticket printer by the words **PRINT** and **TICKET**, respectively. In this way, one may define high-level text output procedures that will work equivalently on any text device.

!IP store-i-p T (Sequence)

(dev —)

Set the current input device to *dev*.

The sequence may be implemented as **`LIT2 USERVAR STORE`**.

`!OP` store-o-p T **`SETOP`**

(dev —)

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

`@IP` fetch-i-p T (Sequence)

(— dev)

Return the device code *dev* for the current input device.

The sequence may be implemented as **`LIT2 USERVAR FETCH`**.

`@OP` fetch-o-p T **`GETOP`**

(— dev)

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

`.` dot F,H,T (Sequence)

(num —)

Display *num.* The number is displayed without any leading zeros. If negative, it is preceded by a minus sign.

The sequence may be implemented as:
`DUP ABS LIT0 LTNMBR NMBRS ROT SIGN NMBRGT`
`GETOP DEVWRITE THROW SLIT 20`$_{H}$ **`GETOP DEVEMIT`**

`."` dot-quote F,H,T (Sequence)

Compilation: *(“ccc<quote>” —)*

Parse *ccc* delimited by **`"`** (double-quote). Append the run-time semantics given below to the current definition.

Run-time: *(—)*

Display *ccc*.

The sequence may be implemented as:
`STRLIT` `<ccc...>` **`GETOP DEVWRITE THROW`**.

`.R` dot-r F,H,T (Sequence)

(num_1 num_2 —)

Display num_1, right aligned in a field num_2 characters wide. If the number of characters required to display num_1 is greater than num_2, all digits are displayed, with no leading spaces, in a field as wide as necessary.

The sequence may be implemented similarly to the one for **`.`** (dot).

`AT-XY` at-x-y F,H,T (Sequence)

(num_1 num_2 —)

Execute a device-dependent *set absolute position* action on the current device,

using num_1 as the horizontal (x) coordinate and num_2 as the vertical (y) coordinate. The coordinates of the upper-left corner of a display are (0,0). Exception **OUTSIDE-BORDER** shall be thrown if the co-ordinates are outside the device's display region.

The sequence may be implemented as **GETOP DEVATXY**.

BEEP T (Sequence)

(—)

Request the current output device to generate a sound.

The sequence may be implemented as **LIT7 GETOP DEVEMIT**.

BL F,H,T (Sequence)

(— 32)

Return the ASCII character for a space.

The sequence may be implemented as **SLIT 20**$_H$.

CR F,H,T (Sequence)

(—)

Execute a device-dependent *new line* action on the current device .

The sequence may be implemented as **LIT13 GETOP DEVEMIT**.

EKEY e-key F,H,T (Sequence)

(— echar)

Read an extended character from the current input device.

The sequence may be implemented a **LIT2 USERVAR FETCH DEVEKEY**.

EKEY? e-key-question F,H,T (Sequence)

(— flag)

Return true *flag* if an extended character is ready to be read from the current input device. The character shall be returned by the next execution of **EKEY**. If **EKEY?** returns a true *flag*, subsequent executions of **EKEY?** (prior to the execution of **EKEY**) shall also return a true *flag*, without discarding keyboard events.

The sequence may be implemented as:
LIT2 USERVAR FETCH DEVEKEYQ

EMIT F,H,T (Sequence)

(char —)

Send the single character defined by *char* to the current output device.

The sequence may be implemented as **GETOP DEVEMIT**.

PAGE F,H,T (Sequence)

(—)

Execute a device-dependent form feed action—such as *clear screen* (terminal

display) or *page eject* (printer)—on the current output device.

The sequence may be implemented as **`LIT12 GETOP DEVEMIT`**.

PRINT C (Sequence)

("<spaces>name" — ior)

Pass the word *name* to the printer device for execution, and return zero if the word was accepted for processing. A word processed in this way may not take stack arguments or return results, but is assumed to generate output which will be routed to the printer. Depending upon the implementation, execution of *name* may proceed concurrently, with further processing in the word calling **PRINT**. Programs may use **STATUS** (see previous section) to determine whether processing is complete.

The sequence may be implemented as:
`ELITC` `<+addr>` **`LIT2 DEVOUTPUT`**

SPACE F,H,T (Sequence)

(—)

Send a space character to the current output device.

The sequence may be implemented as **`SLIT 20`**$_H$ **`GETOP DEVEMIT`**.

SPACES F,H,T (Sequence)

(u —)

If *u* is greater than zero, send *u* space characters to the current output device.

The sequence may be implemented as:
`LIT0 RQDO` `<+addr>` **`SLIT 20`**$_H$ **`GETOP DEVEMIT RLOOP`**

TICKET T (Sequence)

("<spaces>name" — ior)

Process *name*, as described in **PRINT**, except output is routed to the ticket printer device.

The sequence may be implemented as **`ELITC`** `<+addr>` **`LIT3 DEVOUTPUT`**.

TYPE F,H,T (Sequence)

(c-addr len —)

If *len* is greater than zero, send the ASCII string specified by *c-addr len* to the display.

The sequence may be implemented as **`GETOP DEVWRITE THROW`**.

When passed an ASCII character whose bits have a value between 20_H and $7E_H$, inclusive, the corresponding standard character is displayed. The output behaviour of certain control characters is defined for OTA in Volume 1 of this Specification. Behaviour of characters outside the range $20–7E_H$ is undefined.

4.9.3 Communication (Modem) Services

Modem status (*ior*) values that can be returned by the procedures described in this section are

given in Appendix C.

DEVBREAK T **DEVBREAK**

(dev — ior)

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

DEVCALL T **DEVCONNECT**

(c-addr len dev —ior)

Connect to remote system using device *dev.* As long as the connection is not established, **STATUS** applied on device *dev* returns ior **MDM-CONN-IN-PROGRESS**. If the connection is successfully established, **STATUS** returns ior 0. Every other ior indicates an error.

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

DEVHANGUP T **DEVHANGUP**

(dev — ior)

End the current modem session on the given device. This definition should not return until the next output command can be performed immediately.

4.10 Formatted I/O Words

See also **BINARY**, **DECIMAL**, and **HEX** in Section 3.4.1.

number-sign F,H,T **NMBR**

(ud_1 — ud_2)

Convert the least-significant digit of ud_1, in the current number **BASE**, to ASCII representation, and add the resulting character to the left-hand end of the pictured numeric output string, leaving ud_2 as the quotient of ud_1 divided by the current value of **BASE**. Exception **OUTPUT-STRING-OVERFLOW** may be thrown if the internal buffer for numeric output formatting is overrun. Exception **OUT-OF-CONTEXT** may be thrown if **#** is not used within a **<#** ... **#>** delimited number conversion.

#> number-sign-greater F,H,T **NMBRGT**

(d — c-addr len)

Drop *d*. Make the pictured numeric output string available as an ASCII string, at *c-addr* for *len* bytes. A program may replace characters within the string.

Used to terminate a number-conversion process started by **<#**. Exception **OUT-OF-CONTEXT** may be thrown if **#>** is not preceded by **<#**, with no intervening occurrences of **#>**.

#S number-sign-s F,H,T **NMBRS**

(ud_1 — ud_2)

Convert at least one digit of ud_1, according to the rule for **#**, and continue conversion until the quotient ud_2 is zero. Exception **OUTPUT-STRING-OVERFLOW** may be thrown if the internal buffer for number formatting is overrun. Exception **OUT-OF-CONTEXT** may be thrown if **#S** is not used within a **<#** ... **#>** delimited number conversion.

<# less-number-sign F,H,T **LTNMBR**

(—)

Initialise an internal buffer for number formatting.

>NUMBER to-number F,H,T **TONUMBER**

(ud_1 $c\text{-}addr_1$ len_1 — ud_2 $c\text{-}addr_2$ len_2)

Convert ASCII string to number. ud_2 is the unsigned result of consecutively converting the characters within the string (specified by $c\text{-}addr_1$ len_1) into digits, using the current **BASE**; and adding each into ud_1 (after multiplying ud_1 by the number in **BASE**). Conversion continues left-to-right until a character that is not convertible (including any "+" or "-") is encountered, or until the string is entirely converted. $c\text{-}addr_2$ is the location of the first unconverted character, (or of the first character past the end of the string, if the string was entirely converted). len_2 is the number of unconverted characters in the string. Exception **INVALID-NUMERIC-ARG** may be thrown if ud_2 overflows during the conversion.

BASE F,H,T **LIT0 USERVAR**

(— a-addr)

Return the address of the current base for input and output number conversions. Set by **HEX**, **DECIMAL**, and **BINARY** (see Section 3.4.1).

When executed on the host system (e.g., in text being interpreted or in a **COMPILER** definition), the host's **BASE** is returned; otherwise, it is the target's **BASE**.

HOLD F,H,T **HOLD**

(char —)

Add *char* to the current left-hand end of the pictured numeric output string. Exception **OUTPUT-STRING-OVERFLOW** may be thrown if the internal buffer for numeric output formatting is overrun. Exception **OUT-OF-CONTEXT** may be thrown if **HOLD** is not used within a **<#** ... **#>** delimited number conversion.

SIGN F,H,T **SIGN**

(num —)

If *num* is less than zero, add an ASCII "-" character to the current left-hand end of the pictured numeric output string. Exception **OUTPUT-STRING-OVERFLOW** may be thrown if the internal buffer for numeric output formatting is overrun. Exception **OUT-OF-CONTEXT** may be thrown if **SIGN** is not used

within a **<#** ... **#>** delimited number conversion.

4.11 Integrated Circuit Card I/O

This specification provides for one or two ICC readers on a terminal. If there are two, the second reader is normally reserved for maintenance or other special-purpose functions. The status codes returned as an *ior* (I/O result) parameter by ICC access words are given in Appendix C.

-CARD no-card T **CARDABSENT**

(— flag)

Return true *flag*, if an ICC is not present in the reader.

/CARD slash-card T **CARDINIT**

(dev — ior)

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

CARD T **CARD**

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

Send the data in buffer *c-addr$_1$ len$_1$* to the card, and receive data at *c-addr$_2$ len$_2$*. The returned *len$_3$* gives the actual length of the string received. The two buffers may share the same memory region.

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

CARD-OFF T **CARDOFF**

(—)

Power off ICC. Executed when all transactions are complete.

CARD-ON T **CARDON**

(c-addr len$_1$ — c-addr len$_2$ ior)

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

4.12 Magnetic Stripe I/O

Magnetic Stripe I/O is described in further detail in Volume 1 of this Specification. The magstripe status values (*ior*) shown in Appendix C can be returned by the procedures described in

this section.

Table 12: ISO parameter track selection codes

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

MAGREAD T **MAGREAD**

(c-addr len_1 u — c-addr len_2 ior)

Read one or more ISO magstripes. See Table 12 for a definition of the track *u* parameter.

c-addr is the destination address for the string and len_1 is its maximum length. On return, len_2 gives the actual length of the string read. Refer to ISO Standard 7813 for a description of the data formats. A particular device is not required to support all of these possibilities.

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

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

For example :

```
BYTES FROM CARD          BYTES DELIVERED TO APPLICATION
B1 23 4D 7E 5A xx   =>   31 32 33 34 0D 37 0E 35
^             ^ ^LRC
STX           ETX
```

Exception **UNSUPPORTED-OPERATION** shall be thrown if the reader does not support one or more of the requested tracks and the content of the returned buffer shall be undefined. If the requested tracks are supported by the reader but are not present on the card swiped, then ior **SUCCESS** is returned and the buffer contains those tracks that are available on the card.

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

MAGWRITE T **MAGWRITE**

(c-addr len num — ior)

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

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

For example :

```
BYTES DELIVERED BY APPLICATION    BYTES TO CARD
31 32 33 34 0D 37 0E 35        => B1 23 4D 7E 5A xx
                                  ^           ^ ^LRC
                                  STX         ETX
```

4.13 Sockets and Socket Management

These words provide the basic socket management procedures described in Volume 1 of this Specification.

!PLUG-CONTROL store-plug-control T (Sequence)

(xp —)

Establish *xp* as the current procedure for validating socket plug actions. The sequence may be implemented as:
0 PLUGSOCKET SETEQ ELIT INVALID-NUMERIC-ARG QTHROW

INTO C,I (Sequence)

Interpretation: *(xp“<spaces>name”)*

Place *xp* into the socket list position corresponding to *name*.

Compilation: *(“<spaces>name”)*

Append to the current definition a token stream providing the execution semantics defined below.

Execution: *(xp — flag)*

Attempt to plug *xp* into the socket corresponding to *name*, and return a false *flag* if the action was disallowed.

The sequence may be implemented as **LIT** <socket#> **PLUGSOCKET**.

Normally used with **PLUG**, as follows:

```
10 SOCKET FUNC
...
: OP  <some operation> DROP ;
```

Then,

```
PLUG OP INTO FUNC
```

stores the execution pointer for **OP** in the socket **FUNC**. Subsequent use of **FUNC** will cause **OP** to be executed if the **PLUG** succeeded.

All words defined for use in sockets other than socket zero must have no stack effect. The socket-control procedure which is plugged into socket zero takes a socket number from the stack and returns a flag, indicating whether the given socket may be modified.

PLUG C,I **ELITC** <+addr>

Interpretation: *("<spaces>name" — xp)*

Return the execution pointer for *name*. Normally used to provide a value to be plugged into a socket by **INTO**.

Compilation: *("<spaces>name" —)*

Compile the execution pointer for *name*.

Execution: *(— xp)*

Return the execution pointer for *name*.

SOCKET I **DOSOCKET** <n>

(num "<spaces>name" —)

Define a named procedure associated with socket *num* (0–63). When *name* is invoked, the procedure whose execution pointer is presently in socket *num* will be executed.

The behaviour associated with a socket may be specified using **PLUG** and **INTO**.

All words whose behaviour may be associated with a **SOCKET** must have no stack effects.

name Execution: *(—)*

Execute the definition whose execution pointer has been associated with *name* using **PLUG** ... **INTO**. For example:

```
31 SOCKET TRM\ Defines socket 31
: MY-TRM   <words> ; \ Local TRM function
...
PLUG MY-TRM INTO TRM \ At initialisation
```

4.14 Database Services

Generic database support procedures are described in Volume 1 of this Specification. This section describes Forth support for database definition and usage.

The words included here are designed to support all non-TLV internal data structures in terminal programs, such as tables of supported languages, the language messages, the transaction log, and any other application-specific structures.

4.14.1 Basic Concepts

A *database* consists of a variable number of records, each of fixed size and containing a specified list of fields. A database contains certain attributes: its origin or starting location, record size, and next available record. These reside in a Database Parameter Block (DPB), described in more detail in Volume 1 of this Specification.

A database is selected by executing its name. This specifies it as the *current database*—at any given time, there is always one current database. There is also a *current record*, selected by the phrase `<r#>` **SELECT**. **SELECT** will ensure that the requested record is within the range of the current database (0 to **AVAILABLE** -1). The current record number will be invalid after a database is initialised, the current record has been deleted, or an error has occurred.

There are presently three field classes defined: single bytes (used for flags and small integers), 32-bit integers, and strings of a specified length. Each class has *methods* associated with it for fetching and storing the contents. The default behaviour when a field is invoked by name is to fetch the value; byte and integer values are returned on the stack, and string values are returned as *c-addr len*. The phrase **TO** *field-name* stores a value in the field *field-name*. It cannot be used on read-only databases. Byte and integer values are stored from the stack; strings are stored from a given *c-addr len*. The length of the string fetched or stored is given by its field definition.

Except for compiled databases, databases are initialised to have zero records.

A database definition begins with the word **DB:** (or a variant thereof, depending upon the database type), and ends with **END-DB**. Between these words, the fields in this database are defined. Usage for the various defining words is given below.

4.14.2 Database and Record Definitions

The words in this section are used by the compiler to define databases, and record structures within databases.

32BIT: 32-bit-colon I **ELIT** `<num1> <method>`

(num_1 "<spaces>name" — num_2)

Define a 32-bit field *name*, at the specified offset num_1 from the beginning of a record, and increment the offset by four to give num_2. If **32BIT:** is used to construct a compiled database, relocation requires that the *name* be cell-aligned. It is the programmer's responsibility to ensure this (e.g., by using **ALIGNED**).

When *name* is preceded by **TO** in a definition passed to a token compiler, the compiled token stream shall provide **DBSTORE** as the *method*. Otherwise, **DBFETCH** shall be compiled as the *method*.

name Execution:

(access): *(— x)*

The access method of a **32BIT:** field is to return its value on the stack.

(store): *(x —)*

The store method (selected by **TO**) stores the value from the top of the stack.

For example:

```
...  \ preceding part of database/record definition
32BIT: AMOUNT
...  \ remainder of database/record definition
AMOUNT \ returns the value of AMOUNT in the current
     \ record of the current database
1000000 TO AMOUNT \ stores 1000000 in AMOUNT in the
     \ current record of the current database
```

This may not be used on a read-only database. A **DB-INVALID-RECORD** exception shall be thrown if the current record number (**#RECORD**) is not within the range of 0 to **AVAILABLE**-1 for the current database.

BYTE: byte-colon | **ELIT** <num$_1$> <method>

Compilation: *(num$_1$ "<spaces>name" — num$_2$)*

Define a one-byte field *name* at the specified offset *num$_1$* from the beginning of a record, and increment the offset by one to give *num$_2$*.

When *name* is preceded by **TO** in a definition passed to a token compiler, the compiled token stream shall provide **DBCSTORE** as the *method*. Otherwise, **DBCFETCH** shall be compiled as the *method*.

name Execution:

(access): *(— char)*

The access method of a **BYTE:** field is to return its value on the stack

(store): *(char —)*

The store method (selected by **TO**) stores the least-significant byte from the top of the stack.

For example:

```
...  \ preceding part of database/record definition
BYTE: AGE
...  \ remainder of database/record definition
AGE  \ returns the value of AGE in the current
     \ record of the current database
10 TO AGE \ stores 10 in AGE in the current
     \ record of the current database
```

This may not be used on a read-only database. A **DB-INVALID-RECORD** exception shall be thrown if the current record number (**#RECORD**) is not within the range of 0 to **AVAILABLE**-1 for the current database.

COMPILED-DB: compiled-d-b-colon I (Sequence)

Compilation: *("<spaces>name" — 0)*

Same as **DB:**, except define a sequential, pre-initialised database. The data for a compiled database must immediately follow its definition; all bytes, in all fields, must be explicitly allocated. When an instance of a compiled database is created, it is cell-aligned. Fields in the database need not be cell-aligned, except **32BIT:** fields.

For example:

```
COMPILED-DB: MY-DB
     BYTE: LENGTH
     15 STRING: CUSTOMER
     32BIT: ACCOUNT#
END-DB

   ," SMITH, ROBERT A"
   #15 ,
   ," JONES, EDWARD B"
   #16 ,
   ,"                "
   0 ,
   ,"                "
   0 ,
   ,"                "
   0 ,

END-DATA
```

Note that **,"** compiles a counted string. In this example, the first byte (count) appears as the **LENGTH** field, and the 15 ASCII characters following are the **CUSTOMER** field. These two fields leave the record cell-aligned for the **ACCOUNT#** field. If the database is marked read/write, empty records (15 ASCII blanks) may be compiled for temporary use; but note that any modifications to these will be lost upon the next module reload. Other methods of allocating data space (e.g., **ALLOT**) may be used, providing alignment restrictions are observed.

name Execution: *(—)*

Make *name* the current database.

DB: d-b-colon I (Sequence)

("<spaces>name" — 0)

Begin the definition of the database *name*. Return an initial field offset (zero).

The instance is cell-aligned. The sequence may be implemented as **ELITD** <+addr> **DBMAKECURRENT**, where *addr* is a pointer to the database's DPB.

name Execution: *(—)*

Make *name* the current database (sets user variable 3, DBCURRENT). If *name* is executed on the host, the DPB address is returned, instead of setting

DBCURRENT.

Usage:

```
DB: MY-DB
    16 STRING: CUSTOMER
    32BIT: ACCT#
END-DB
```

END-DATA I

(—)

Terminate a compiled database, setting its **AVAILABLE** to the number of records compiled.

END-DB end-d-b I

(num —)

Terminate the definition of a database; save its record size *(num)* in the DPB. The data-space pointer is cell-aligned after this operation. If **[KEY** ... **KEY]** has been used, as evidenced by a non-zero key length, the ordered bit in the DB kind is set. Usage:

```
DB: <database-name>
    <field definitions>
END-DB
```

KEY] key-bracket I

(num_2 — num_2)

End the specification of the key field in a database definition by subtracting num_1 (the starting offset—saved by **[KEY**, see below) from num_2 (the current offset within the record), to record the key length.

NV-DB: n-v-d-b-colon I (Sequence)

("<spaces>name" — 0)

Same as **DB**:, except define a *non-volatile* database that will not be cleared after power down. Non-volatile databases are used for data that must survive a power-down or module re-load.

For example:

```
NV-DB: MY-DB
    16 STRING: CUSTOMER
    32BIT: ACCOUNT#
END-DB
```

name Execution:*(—)*

Make *name* the current database.

STRING: string-colon I (Sequence)

(num_1 len "<spaces>name" — num_2)

Define a string field named *name*, of the specified length *len* at the specified

offset num_1 from the beginning of a record, and increment the offset by *len* to give num_2.

The sequence may be implemented as:

ELIT <num$_1$> **LIT** <len> <method>

When *name* is preceded by **TO** in a definition passed to a token compiler, the compiled token stream shall provide **DBSTRSTORE** as the <method>. Otherwise, **DBSTRFETCH** shall be compiled as the <method>.

name Execution:

(access): *(— c-addr len)*

The access method of a **STRING** field is to return its address and length.

(store): *(c-addr len —)*

The store method (selected by **TO**) stores the string at *c-addr len.* If the given string is shorter than the actual field size, the field shall be padded with blank (ASCII 20_H) characters to fill the remaining space.

For example:

```
...  \ preceding part of database/record definition
16 STRING: NAME
...  \ remainder of database/record definition
NAME \ returns the addr len of NAME in the
     \ current record of the current database
S" ABC" TO NAME \ stores the string ABC in NAME in
     \ the current record of the current database
```

This may not be used on a read-only database. A **DB-INVALID-RECORD** exception shall be thrown if the current record number (**#RECORD**) is not within the range of 0 to **AVAILABLE**-1 for the current database.

[KEY bracket-key |

(num_1 — num_1)

Start the specification of the key field in a database definition by recording num_1 (the current offset within the record being defined) as the starting offset of the key. For example:

```
DB: MY-DB
     [KEY  16 STRING: LAST-NAME
             16 STRING: FIRST-NAME  KEY]
     32BIT: ACCT#
     ...   \ more fields
END-DB
```

In this database, the two fields **LAST-NAME** and **FIRST-NAME** together form the key. Program validation will require that key fields are read-only.

4.14.3 Database Attributes

The words in this section allow to set attributes of databases. Each of these attribute words should immediately follow the database definition (**DB:** and variants) or one of the other

attribute words.

[READ-ONLY] bracket-read-only I

(—)

Sets an attribute in the DPB indicating that this database is to be read only. Any write operation shall cause exception **DB-INVALID-FUNCTION**. This word should immediately follow the database definition (**DB:** and variants) or one of the other attributes.

[SEQUENTIAL] bracket-sequential I

(—)

Sets an attribute in the DPB indicating that this database is to be optimised for sequential insertion and deletion. VM implementations are not required to recognise this attribute. This word should immediately follow the database definition (**DB:** and variants) or one of the other attributes.

[SECURE] bracket-secure-memoryI

(—)

Sets an attribute in the DPB indicating that this database is to be stored in secure memory. VM implementations are not required to recognise this attribute. If a module is loaded that requires this feature in a database and the VM does not support this attribute, a **MOD-CANNOT-LOAD** exception shall be thrown. This word should immediately follow the database definition (**DB:** and variants) or one of the other attributes.

4.14.4 Database Access and Management

The words in this section are available to terminal programs for managing databases.

#RECORD number-record T (Sequence)

(— u)

Return the record number of the currently selected record.

The sequence may be implemented as **LIT4 USERVAR FETCH**.

ADD-RECORD T **DBADDREC**

(—)

Add one record at the end of the current database, at the record number given by **AVAILABLE**. The new record becomes the current record for this database and its content is unspecified. Exception **DB-INVALID-FUNCTION** shall be thrown if the current database type is read-only, compiled, or ordered. Storage requirements are increased by this definition, and it shall cause an **OUT-OF-MEMORY** exception to be thrown.

AVAILABLE T **DBAVAIL**

(— u)

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

DBSIZE d-b-size T **DBSIZE**

(— len)

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

DELETE-RECORD T **DBDELREC**

(—)

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

A **DB-INVALID-FUNCTION** exception shall be thrown if the current database type is read-only or compiled. A **DB-INVALID-RECORD** exception shall be thrown if the current record number (**#RECORD**) is not within the range of 0 to **AVAILABLE**-1 for the current database.

ENTIRE T (Sequence)

Provides a global reference to a *field* consisting of the entire current record of the current database, handled as a single binary string.

(access): *(— addr len)*

The access method of the **ENTIRE** field is to return its address and length.

The sequence may be implemented as **LIT0 DBSIZE DBSTRFETCH**.

(store): *(c-addr len —)*

The store method (selected by **TO**) stores the string at *c-addr len* into the current record. If the given string is shorter than the actual field size, the field shall be padded with blank (ASCII 20_H) characters to fill the remaining space.

For example:

```
DBSIZE BUFFER: TEMP
4 SELECT ENTIRE  TEMP SWAP MOVE
5 SELECT  TEMP DBSIZE TO ENTIRE
```

In this example, the first usage of **ENTIRE** returns the *addr len* of the fifth record of the current database; the second usage of **ENTIRE** copies the entire fifth record into the sixth record of the current database. Intermediate storage is required, since there is only one record buffer per database.

The sequence may be implemented as **LIT0 DBSIZE DBSTRSTORE**.

A **DB-INVALID-FUNCTION** exception shall be thrown if the store method is used on a read-only database. A **DB-INVALID-RECORD** exception shall be thrown if the current record number (**#RECORD**) is not within the range of 0 to **AVAILABLE**-1 for the current database.

INITIALIZE T **DBINIT**

(—)

Delete all records in the current database and set *available* record number of the database (see **AVAILABLE**) to zero. A **DB-INVALID-FUNCTION** exception shall be thrown if the current database type is read-only or compiled. The currently selected record number shall be set to -1 (invalid) by this operation.

RECORDS T (Sequence)

(— num_1 num_2)

Return **DO** ... **LOOP** arguments for the total number of active records in the current database.

The sequence may be implemented as **DBAVAIL LIT0**.

RESTORE-DB restore-d-b T **DBRESTORE**

(—) (R: u_1 u_2 —)

Restore from the return stack the database and record settings that were saved by **SAVE-DB** in the current definition. u_1 is the user variable DBCURRENT, and u_2 is the user variable DBRECNUM.

SAVE-DB save-d-b T **DBSAVE**

(—) (R: *— x_1 x_2)*

Save the current database context on the return stack. u_1 is the user variable DBCURRENT (identifying the current database), and u_2 is the user variable DBRECNUM (identifying the current record). Each usage of **SAVE-DB** must be paired with a corresponding usage of **RESTORE-DB** in the same definition.

SELECT T **DBSELECT**

(num —)

Select record *num* in the currently selected database, and set user variable DBRECNUM to *num*. A **DB-INVALID-RECORD** exception shall be thrown if *num* is not within the range of 0 to **AVAILABLE**-1 for the current database.

4.14.5 Name List Management

The definitions in this section facilitate management of database records maintained in ascending order of a key field.

ADD-NAME T **DBADDBYKEY**

(c-addr len — flag)

Search the current database for a key field that matches the ASCII string specified by *c-addr* and *len*. *len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 20_H) characters. If a matching record already exists, make the matching record current and set *flag* false. If the match is not successful, a new record shall be inserted at the correct position in the database, and the *flag* is true. This new record becomes the current record, and its content is unspecified except for

its key field, which shall contain the given key. Storage requirements are increased by this definition, and it shall cause an **OUT-OF-MEMORY** exception to be thrown.

A **DB-INVALID-FUNCTION** exception shall be thrown if the current database type is not ordered, or if it is read-only or compiled.

DEL-NAME T **DBDELBYKEY**

(c-addr len — flag)

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

A **DB-INVALID-FUNCTION** exception shall be thrown if the current database type is not ordered, or if it is read-only or compiled.

FIND-NAME T **DBMATCHBYKEY**

(c-addr len —flag)

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

A **DB-INVALID-FUNCTION** exception shall be thrown if the current database type is not ordered.

4.15 Language and Message Handling

4.15.1 Cardholder Language Selection

Cardholder language selection may be provided at some terminals, optionally including support for the ICC to request services in the cardholder's native language (using ICC-supplied data). This means that different languages (i.e., multiple message tables) must be supported, potentially involving code page changes to provide special character sets. Provision is made for downloading both message tables and code pages from an ICC.

CHOOSE-LANGUAGE T **CHOOSELANG**

(c-addr — flag)

Select the language whose ISO 639 language code is given by the two characters at c-*addr*. If *flag* is true, the language was found and is now the current language. Otherwise, the calling program should select another language. At

least one language (the terminal's native language) shall always be available. It is the responsibility of the program using **CHOOSE-LANGUAGE** to make the current language selection available to any client program using **MSGLOAD**.

CODE-PAGE? code-page-question T **CODEPAGE**

(u — flag)

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

LANGUAGE-CNAME language-c-name T (Sequence)

(— addr len)

Return the ASCII string parameters for the name (in code page characters) used to describe the currently selected record in the terminal's language database, which must be the current database (see **LANGUAGES**). Any use while another database is selected is ambiguous.

The sequence may be implemented as **DBSTRFETCHLIT 18 16**.

LANGUAGE-NAME T (Sequence)

(— addr len)

Return the ASCII string parameters for the name (in 7-bit ASCII codes) used to describe the currently selected record in the terminal's language database, which must be the current database (see **LANGUAGES**). Any use while another database is selected is ambiguous.

The sequence may be implemented as **DBSTRFETCHLIT 2 16**.

LANGUAGE-PAGE T (Sequence)

(— char)

Return the ISO 8859 code page number used by the currently selected record in the terminal's language database, which must be the current database (see **LANGUAGES**). Any use while another database is selected is ambiguous.

The sequence may be implemented as **DBCFETCHLIT 34**.

LANGUAGE-SYMBOL T (Sequence)

(— addr len)

Return the ASCII string parameters for the ISO 639 two-character symbol used to refer to the currently selected record in the terminal's language database, which must be the current database (see **LANGUAGES**). Any use while another database is selected is ambiguous.

The sequence may be implemented as **DBSTRFETCHLIT 0 2**.

LANGUAGES T **LANGUAGES**

(—)

Make the terminal's languages database the current database. This is a read-only, non-volatile database.

LOAD-PAGE T **LOADPAGE**

(c-addr — flag)

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

4.15.2 Messages and Message Displays

A terminal displays most transaction information to the cardholder through numbered messages. A terminal may contain more than one of these sets of messages, in different languages. A terminal may also download a new set of messages from an ICC, either to support a new language or to provide additional messages.

/MESSAGES slash-messages T **MSGINIT**

(—)

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

DELETE-MESSAGES T **MSGDELETE**

(c-addr — flag)

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

GET-MESSAGE T **MSGFETCH**

(u — c-addr len)

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

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

LOAD-MESSAGES T **MSGLOAD**

(c-addr — flag)

Install a message table at *c-addr*, including message numbers in the range 40–FF_H in the transient messages buffer, overwriting any previous messages with the same message numbers. Messages 40–BF_H may be provided only by a terminal application; messages C0–FF_H may be provided by an ICC. The message table must be formatted as described in Volume 1 of these Specifications, Section 4.4. *flag* is true, if the messages were loaded successfully; or false, if the message table's code page is not currently selected or if the messages are outside the range 40–FF_H.

UPDATE-MESSAGES T **MSGUPDATE**

(c-addr —)

Install a message table, including message numbers (in the range 1–FF_H), at *c-addr* into the resident language database. If a language with the same code is already present, new messages will replace previous messages with the same

number. Exception **LANGUAGE-FILE-FULL** may be thrown if there is not sufficient space for the new language.

c-addr gives the location of the message table definition, formatted as described in Volume 1 of these Specifications, Section 4.4.

4.16 TLV Services

The words in this section assist in managing TLV (Tag, Length, Value) data structures. TLV services provided by the Virtual Machine are described in Volume 1 of this Specification.

4.16.1 TLV Description Syntax

This section describes the notation used in Forth for defining TLV data structures.

4.16.1.1 Format Specifiers

There are seven TLV data types, using three basic formats: straight binary; numeric strings, storing two digits per byte; and ASCII strings, storing one digit per byte. Table 13 provides a list of the named constants used to specify each type, along with a complete description of the format and its stack usage.

Table 13: Named TLV formats

Value	Name	Description	Format	Stack
0	**TLV-N**	Right-justified BCD nibbles; if needed, leading pad(s) of nibbles with all bits zero.	Integer	*u*
1	**TLV-B**	Sequence of bytes, application-defined.	String	*c-addr len*
2	**TLV-BN**	Binary number; msb is the first byte.	Integer	*u*
3	**TLV-CN**	Left-justified four-bit binary coded nibbles; if needed, trailing pad(s) of nibbles with all bits set.	CN string	*c-addr len*
4	**TLV-AN**	Alphabetic and numeric characters only, with trailing zeroes.	ASCII string (unchanged)	*c-addr len*
5	**TLV-ANS**	Full ASCII character set with trailing zeroes.	ASCII string (unchanged)	*c-addr len*
6	**TLV-VAR**	Variable format.	String	*c-addr len*

The words shown in Table 13 may be used to supply the stack argument *fmt*, as called for in the words below, with the format selector in the low-order byte, as shown. Integer formats **TLV-N** and **TLV-BN** are handled on the stack (*u*); and strings, either ASCII or numeric, are referred to by address and length in bytes (*c-addr len*).

4.16.1.2 Internal data structures

TLV data structures are allocated by the host compiler in initialised memory. The definition-dependent fields for a TLV instance are filled by the defining word **TLV:**, and the remaining

fields are initialised. The defining word **TLV:** expects two values on the stack, plus a following name. The values are the type code (see above) and the tag, expressed as a hex number. A TLV instance is accessed by executing its name. The invocation of the name places the address of the TLV instance (the TLV *access parameter*) on the stack.

It is recommended to edit source text containing many TLV definitions in a line-oriented, tabulated form for ease of readability. Comments may be added to each line. Here is an example:

```
\ Format   Tag       Name                    Description
  TLV-N    $5F24 TLV: TLV-EXPIRATION-DATE   \ Application Expiration …
  TLV-VAR  $75   TLV: TLV-RESPONSE-MESSAGE  \ Response Msg Template
  TLV-BN   $81   TLV: TLV-AMOUNT-BINARY     \ Amount, Authorised …
  TLV-B    $82   TLV: TLV-AIP               \ Application Interchange …
  TLV-ANS  $84   TLV: TLV-DF-NAME           \ Dedicated File Name
```

In this example, the first line is a comment, providing column headings. The columns contain:

Format	Format specifier, from Table 13.
Tag	Tag sequence (in hex) for the item.
(blank col.)	Defining word **TLV:** that defines the data item.
Name	Name used to refer to this TLV within the source code.
Description	Comment describing the item.

A table, such as that given in the example, is processed by the compiler to produce definitions of the data structures. When the compiled program's initialised data space is constructed in the target, the tags will be linked to their respective data areas. This enables the terminal program to parse a TLV sequence, based on its tags.

Data may be assigned to TLV instances by **TLV-PARSE**, **!TLV**, and **!TLV-RAW**. TLV structures are decoded by **TLV-PARSE** for each tagged value encountered in a string. Program data may be converted by **!TLV**, or stored directly by **!TLV-RAW**. Assigned data may be retrieved by **@TLV**, **@TLV-RAW**, **TLV+DOL**, or **TLV+STRING**. Single bits of binary strings may also be set, reset, and queried.

4.16.1.3 TLV string creation

A separate set of definitions is provided to build named strings containing one or more TLVs and their values. These may be used for the terminal's internal data, and also to provide card data structures to permit testing in the absence of cards. This specification only deals with those definitions used to build terminal data structures, which reside in initialised data space. The compiling operators for the different TLV types are given in Table 14.

Table 14: Compiling operators for TLV values

Type	Compiling Word	Example
TLV-AN	**A"**	**$5F2D [TLV (Language Preference) A" enfres" TLV]**
TLV-ANS	**A"**	**$5F20 [TLV (Cardholder Name) A" John Major" TLV]**
TLV-B	**C,**	**$9F07 [TLV (Appl. Usage Control) HEX FF C, C0 C, TLV]**

Table 14: Compiling operators for TLV values

Type	Compiling Word	Example
TLV-BN	**,BN**	**$5F40 [TLV (Purse balance) 10000 ,BN TLV]**
TLV-CN	**CN"**	**$5F25 [TLV (Effective Date) CN" 951231" TLV]**
TLV-N	**,BCD**	**$5F24 [TLV (Expiration Date) 951231 ,BCD TLV]**
TLV-VAR	**C,**	**$BF0C [TLV (Issuer Discr. Data) 0 C, TLV]**

To define TLV strings in the terminal, you may use **[TLV-CREATE** to begin a named structure definition, ending it with **TLV]**. Within this structure, use **$**<tag> **[TLV** <data-value> **TLV]** to define specific tags. The structure name must then be passed to **TLV-PARSE** for actual assignment of data to the TLVs.

To build actual values within a TLV string, follow **[TLV** with sequences consisting of the desired value and a compiling operator, depending upon the format. Appropriate operators and examples are shown in Table 14. To build a constructed TLV, the data value itself is made up of one or more primitive TLV definitions. Here are two examples, first a primitive TLV and then a constructed TLV:

```
[TLV-CREATE  LANGUAGE-PREFERENCE
     $5F2D [TLV  A" enfres"  TLV]
TLV]

[TLV-CREATE  DATES
     $7FFF [TLV
          $5F25 [TLV  ( Eff.)  CN" 951231"  TLV]
          $5F24 [TLV  ( Exp.)  951231 ,BCD  TLV]
     TLV]
TLV]
```

4.16.1.4 TLV bits

The EMV specification defines a number of bit fields in TLVs that contain flags. These individual bits may be named, using the defining word **TLV-BIT:**. The syntax is:

<bit-number> **TLV-BIT:** <bit mask name> <TLV name>

Bit-mask names provide data hiding and improved source-code readability. TLVs containing bit fields must be Type 1 (**TLV-B**). A value must be stored into the TLV before defining individual bit-mask names; the size of the value determines the maximum *bit-number* available. Bit numbers are zero-relative (note that EMV 96 labels bits one-relative).

For example:

```
\ Card Data Input Capabilities
7 TLV-BIT: MANUAL-KEY-ENTRY   TLV-TERMINAL-CAP
6 TLV-BIT: MAGNETIC-STRIPE    TLV-TERMINAL-CAP
5 TLV-BIT: IC-WITH-CONTACTS   TLV-TERMINAL-CAP
```

This defines bit mask names for three bits in the TLV whose name is **TLV-TERMINAL-CAP**.

The phrase *mask name* **@TLV-BIT** returns a *flag* which is true if the bit defined is on in the internal data buffer for this TLV.

The phrase *flag mask name* **!TLV-BIT** will set the bit in the internal data buffer for the TLV defined, according to the truth state of the *flag*.

4.16.2 TLV Data Definitions

The following definitions may be used in the host to build run-time data structures for TLVs in initialised data space.

TLV-AN t-l-v-a-n T **LIT4**

(— 4)

Return the constant 4, used to define alphanumeric character strings.

TLV-ANS t-l-v-a-n-s T **LIT5**

(— 5)

Return the constant 5, used to define ASCII strings.

TLV-B t-l-v-b T **LIT1**

(— 1)

Return the constant 1, used to define a sequence of bytes.

TLV-BIT: t-l-v-bit-colon I (Sequence)

(u “<spaces>name$_1$” “<spaces>name$_2$” —)

Skip leading space delimiters. Parse *name$_1$* and *name$_2$*, delimited by spaces. *name$_2$* must be the name of a previously defined TLV. Create a definition for *name$_1$*, whose behaviour is given below.

When token compiling, *name$_1$* is defined as a procedure that contains the tokens **SLIT** *u* **ELITD** *a-addr*. References to a **TLV-BIT:** imported from a different module would replace the **ELITD** with an **IMCALL**.

name$_1$ Execution: *(— u a-addr)*

Return the specified bit number and the access parameter of the TLV *name$_2$*.

TLV-BN t-l-v-b-n T **LIT2**

(— 2)

Return the constant 2, used to define a binary number.

TLV-CN t-l-v-c-n T **LIT3**

(— 3)

Return the constant 3, used to define BCD nibbles padded with trailing F’s.

TLV-N t-l-v-n T **LIT0**

(— 0)

Return the constant 0, used to define BCD nibbles padded with leading zeros.

TLV-VAR t-l-v-var T **LIT6**

(— 6)

Return the constant 6, used to define variable strings.

TLV: t-l-v-colon I **ELITD <addr>**

(num_1 num_2 "<spaces>name")

Skip leading space delimiters. Parse *name* delimited by spaces. Create a definition for *name*, which is a TLV data item whose format is num_1 (see Table 13), whose tag is num_2, and whose value is in initialised data space. For example:

```
\ Fmt  Tag          Name       Description
TLV-B  $9F06  TLV: TLV-AID  \ Application Identifier
```

name Execution: *(— a-addr | c-addr len)*

Return the access parameter(s) of the TLV *name*.

TLV] t-l-v-bracket I

Compilation: *(c-addr —)*

Close the TLV data structure that began at *c-addr*. The structure may be moved back, and the data space pointer adjusted, so that the length bytes at the given address follow ISO/IEC 8825 rules. For TLVs that may be passed between the terminal and an ICC, the total adjusted length of this structure may not exceed 252 bytes of data, plus three length bytes. For TLVs that will be assigned data by **TLV-PARSE**, the data field may not exceed 256 bytes.

[TLV bracket-t-l-v I

Compilation: *(num — c-addr)*

Begin a TLV data structure by compiling the tag name (given by the hex value on the stack), followed by a default three-byte length. The address returned is the beginning of this three-byte length sequence that will be adjusted by **TLV]** when the structure is closed. These TLV structures can be nested, as needed, as long as the total length of the topmost structure does not exceed 65535 bytes.

[TLV-CREATE bracket-t-l-v-create I (Sequence)

("<spaces>name" — c-addr)

Define a named TLV structure that will be closed by **TLV]**. The address returned is the beginning of the length sequence, as described in **[TLV**, above. It is assumed that additional **[TLV** … **TLV]** sequences will be defined within this structure.

The sequence may be implemented as:
ELITD <+addr> **LIT3 TLVFETCHLENGTH DROP NIP**

name Execution: *(— c-addr len)*

Return the starting address and length of the data in the TLV structure *name*. The structure definition must follow ISO/IEC 8825 rules. The address and length are suitable to be passed to **TLV-PARSE**, for data assignment to the indi-

vidual TLVs in the structure.

4.16.3 TLV String Services

The words in this section manage the strings used for TLV-encoded data objects, and the string formats used in TLVs. Also see the general string-handling words in Section 4.5.

!TLV store-t-l-v T **TLVSTORE**

(u a-addr | c-addr len a-addr —)

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

!TLV-BIT store-t-l-v-bit T **TLVBITSTORE**

(flag u a-addr —)

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

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

!TLV-RAW store-t-l-v-raw T **TLVSTORERAW**

(c-addr len a-addr —)

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

@TLV fetch-t-l-v T **TLVFETCH**

(a-addr — u | c-addr len)

Return the data assigned to the TLV definition whose access parameter is *a-addr*. Apply a conversion, according to the associated type field—see Table 13 (page 4–79). Type codes 0 and 2 return an unsigned number *u* on the stack, while the others return string parameters *c-addr len*. For type 3 (CN), the address returned is valid only until the next execution of **@CN**, or until a reference to another CN string. The *len* returned for strings shall be the same as that last stored in the buffer. The *len* or *u* returned for a TLV which has not had data assigned to it shall be zero.

Note: If any change is made to the data returned by this operator, it is not guaranteed to be recorded until **!TLV** is executed.

@TLV-BIT fetch-t-l-v-bit T **TLVBITFETCH**

(u a-addr — flag)

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

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

@TLV-LENGTH fetch-t-l-v-length T **TLVFETCHLENGTH**

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

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

@TLV-RAW fetch-t-l-v-raw T **TLVFETCHRAW**

(a-addr — c-addr len)

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

If any change is made to the data returned by this operator, it is not guaranteed to be recorded until **!TLV-RAW** is executed.

@TLV-TAG fetch-t-l-v-tag T **TLVFETCHTAG**

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

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

@TLV-VALUE fetch-t-l-v-value T **TLVFETCHVALUE**

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

Split the input string at $c\text{-}addr_1$, which is expected to contain the length and value fields of a TLV. Decode the length field. Return the trailing string $c\text{-}addr_2$ len_2, the value field $c\text{-}addr_3$ len_3, and a true *flag*, if the information in the length and value fields is entirely contained within the input string. Otherwise, return a false *flag*, with the remaining parameters being undefined.

TLV-BIT-CLEAR T **TLVBITCLEAR**

(u a-addr —)

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

Byte 1.

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

TLV-BIT-SET T **TLVBITSET**

(u a-addr —)

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

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

4.16.4 TLV Management

TLV+DOL t-l-v-plus-d-o-l T **TLVPLUSDOL**

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

Process *len$_1$* bytes at *c-addr$_1$*, for a sequence of tag and length fields (a *Data Object List*). For all tag fields encountered, the corresponding value fields are concatenated, without delimiters, into the buffer at *c-addr$_2$* (up to its maximum size, *len$_2$*). The parts of the resulting string are filled, or truncated where a specified length field differs from the size of the existing value; see EMV *Integrated Circuit Card Specification for Payment Systems*, Part 2 (Data Elements and Commands). A **STRING-TOO-LARGE** exception may be thrown if *len$_2$* is not large enough to hold all the values. This word returns the beginning of the destination string (*c-addr$_2$*), and the resulting output length (*len$_3$*, not greater than *len$_2$*).

TLV+STRING t-l-v-plus-string T **TLVPLUSSTRING**

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

Retrieve tag, length, and assigned value from the TLV definition whose access parameter is *a-addr*. Append a well-formed TLV sequence to the end of the output string at *c-addr len$_1$*. Append an empty TLV, if no data has been assigned. Return the beginning of the destination string (*c-addr*) and the sum of the two lengths (*len$_2$*). It is the programmer's responsibility to ensure that there is sufficient room at the end of the output string to hold both strings.

TLV-CLEAR t-l-v-clear T **TLVCLEAR**

(a-addr —)

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

TLV-FIND t-l-v-find T **TLVFIND**

(u — a-addr | 0)

Return the access parameter for the TLV whose tag is given as *u;* or return zero, if there is no TLV defined for tag *u*. For example, **$9F18 TLV-FIND** returns *addr*.

TLV-FORMAT t-l-v-format T **TLVFORMAT**

(a-addr – fmt)

Return the format code associated with the TLV definition whose access parameter is *a-addr*. The returned *fmt* is a format indicator 0–6 (see Section 4.16.1.1).

TLV-INITIALIZE t-l-v-initialise T **TLVINIT**

(—)

Clear all internally maintained data associated with TLV definitions, and set the status of all TLV definitions to *not assigned*. In the TLV *status character*, only bit 0 is affected. This command is provided to satisfy the security requirement that an application must not be able to access data belonging to another application.

TLV-PARSE t-l-v-parse T **TLVPARSE**

(c-addr len —)

Process *len* bytes at *c-addr* for TLV sequences, according to ISO/IEC 8825 rules, expecting a valid combination of primitive or constructed TLV definitions. For each TLV encountered in the string, the value field is assigned to the TLV definition associated with that tag. A **STRING-TOO-LARGE** exception may be thrown if the tag, length, and value fields are not entirely contained within the input string, or if the length field indicates a value field larger than 252 bytes. For each successfully parsed TLV, the assigned status bit in the definition record shall be set. If a tag is not found, an exception shall not be thrown. When a constructed TLV is encountered, whether or not its tag is found, its value field shall be recursively parsed for embedded TLV sequences.

TLV-STATUS t-l-v-status T **TLVSTATUS**

(a-addr — char)

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

Bit position	Meaning
0	0 = value field not assigned 1 = value field assigned
1–7	Reserved for future use

A null or empty field is a valid, assigned value field. Programs should not assume that status bits reserved for future use are zero.

TLV-TAG t-l-v-tag T **TLVTAG**

(a-addr — u)

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

TLV-TRAVERSE t-l-v-traverse T **TLVTRAVERSE**

(c-addr len xp —)

Process *len* bytes at *c-addr* for TLV sequences, according to ISO/IEC 8825 rules, expecting a valid combination of primitive or constructed TLV definitions. For each TLV encountered in the string, the tag and the whole of the value field is sent to the method *xp*. A **STRING-TOO-LARGE** exception may be thrown if the tag, length, and value fields are not entirely contained within the input string. When a constructed TLV is encountered, its tag and value fields shall be passed to method *xp;* then the value field is recursively traversed for embedded TLV sequences which are, in turn, passed to *xp*.

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

4.17 Hot Card List Management

This section includes procedures specific to the management of a large hot card list, as discussed in Section 4.6 of Volume 1.

+LIST plus-list T **HOTADD**

(c-addr len — flag)

Add an entry to the list, where *c-addr len* is the data for the entry. The data is in compressed numeric form, and may be up to 10 bytes long (19 digits). If it is shorter, the VM will pad it with trailing F_H, as needed. The input string may be constructed with, e.g., **CN"**.

The returned *flag* is true, if the addition was successful. Possible reasons for failure to add include: no room in the list, an entry with exactly the same PAN (including wildcards) already exists, and data for the entry is invalid—e.g., contains wildcard(s) followed by digit(s).

/LIST slash-list T **HOTINIT**

(—)

Initialise the hot card list to an empty state.

-LIST minus-list T **HOTDELETE**

(c-addr len — flag)

Delete the entry with data *c-addr len* from the list. An entry shall be deleted from the list only if there is an exact match; no wild card search is performed.

The returned *flag* is true, if the deletion was successful (the entry was found).

?LIST question-list T **HOTFIND**

(c-addr len — flag)

Search the list for an entry that matches the input PAN *c-addr len*. The input data may be up to 10 bytes long (19 digits).

The returned *flag* is true, if the list contained a matching entry (i.e., identical up to the first F_H in the entry).

4.18 Cryptographic Services

A number of cryptographic algorithms and auxiliarey routines are provided by the standard OTA VM implementation. The algorithms are selected by using a code from Table 15 as the top-of-stack argument to the **CRYPTO** procedure. Exception **UNSUPPORTED-OPERATION** shall be thrown if a particular function is not supported by the VM. Detailed descriptions of these algorithms are given in Volume 1 of this Specification.

Table 15: Codes specifying security algorithms

Value	Description
1	Modulo multiplication
2	SHA-1 (as specified in FIPS 180-1)
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

CRYPTO T **CRYPTO**

*(i*x num — j*x)*

Apply the cryptographic algorithm specified by *num*, with parameters *i*x*, and return results *j*x*. The values of *num* are specified in Table 15. Specific input and output stack parameters for each algorithm are given in Volume 1 of this Specification.

4.19 Operating System Interface

OSCALL o-s-call T **OSCALL**

(a-addr num fn —)

Call an operating system function *fn* with *num* cell-sized arguments in the array at *a-addr*. Individual functions are terminal dependent and are defined in Appendix D to Volume 1 of this specification. Exception **UNSUPPORTED-OPERATION** shall be thrown if a particular function is not supported by the

Virtual Machine.

SETCALLBACK T **SETCALLBACK**

(xp —)

Announce *xp* as an OTA routine that may be called by the underlying operating system.

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

4.20 Miscellaneous

BOUNDS H,T (Sequence)

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

Given the address and length of a string or array, return suitable parameters for a **DO** or **?DO** loop (see Section 3.9, “Control Structures”) to process it.

The sequence may be implemented as **OVER ADD SWAP**.

CHARS chars F,H,T

(num$_1$ — num$_2$)

Return *num$_2$*, the size in address units of *num$_1$* characters. This definition is a null operation in OTA (*num$_2$ = num$_1$*), and does not generate any token code; it is included for source code compatibility reasons.

DEPTH F,H,T **DEPTH**

(— +num)

Return current depth of the data stack. *+num* is the number of single-cell values that were contained in the data stack before *+num* was placed on the stack.

EXECUTE F,H,T **ICALL**

*(i*x xp — j*x) (* R: *nest-sys —)*

Perform the behaviour associated with the execution pointer *xp*. Other stack effects are due to the word executed.

EXIT F,H,T **RETURN**

(—) (R: *nest-sys —)*

Return control to the calling definition specified by *nest-sys*. **EXIT** may not be used to exit from **DO** ... **LOOP** or **DO** ... **+LOOP** structures.

If local variables are used, this function will be implemented by a sequence such as **RELFRAME RETURN**.

FALSE F,H,T **LIT0**

(— flag)

Return a false *flag*, a single-cell value with no bits set.

GOTO go-to T **IJMP**

(xp —)

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

TRUE F,H,T **LITMINUS1**

(— flag)

Return a true *flag*, a single-cell value with all bits set.

4.21 Debug support

The words in this section are used in debugging; their action may be removed from final applications.

BREAKPOINT T **BREAKPNT**

(—)

Cause a breakpoint, which is handled by special debugging code.

5 Module Interfaces

OTA Forth compilers may be used for several purposes: to compile a VM implementation, to compile target-specific code that may be downloaded for immediate testing, or to prepare a token module for installation in a terminal.

In the latter case, provision is made for establishing links with other modules. Three kinds of inter-module interactions are provided:

1. A module may import pointers to procedures in other modules (such as library modules).
2. A module may export procedures for use by other modules.
3. A module may provide alternative procedures to be plugged into sockets (provided the previously loaded module allows it), as described in Section 4.13.

General principles of module management are described in Volume 1 of this Specification. This section provides the Forth words for creating and managing modules, as well as importing and exporting procedures.

5.1 Module Definition Words

The words in this section provide for the definition of OTA modules.

Modules are created and named with **MODULE:**, and the definition is terminated by **END-MODULE**. All code must be compiled, and all directives (e.g., set module attributes, import words, export procedures, etc.) must be used after **MODULE:** and before **END-MODULE**.

Mandatory module attributes are the module identifier and module version number, which are set by **MODULE-ID** and **VERSION#**, respectively. The module entry point attribute is optionally set, using **ENTRY:** as necessary. Only executable modules have entry points; modules referenced only through export procedures (such as libraries) do not have entry points.

After the module definition is terminated by the action of **END-MODULE**, the compiler will typically generate an OTA module under a filename based upon the module name. To allow the greatest portability of files, it is recommended that the name of a module be limited to eight characters, and contain no characters that could cause problems when moved across platforms.

A module may export up to 256 procedures for use within other modules. The phrase **EXPORT** *name* adds a procedure, defined in this module, to the export list (see the section "Module Delivery Format" in Volume 1 of this Specification). This may take place any time after the procedure *name* is defined, but the recommended practice is to list all exported procedures after all the code is compiled for the module.

To use exported procedures, a module adds entries to its own import list by executing **FROM** *module name*, followed by a list of **IMPORT** *procedure* entries. This makes specifically referenced procedures from particular modules available. A module can import procedures

from up to 256 other modules, giving a maximum of 65536 possible imported words.

END-MODULE I

(—)

Terminate the definition of the current module. Update appropriate fields in the module header image. This must be done before another module can be compiled. If there is no current module, the compiler may report an error. **END-MODULE** may check for the existence of a valid version number and module identifier, or this step may be left to the module-certification process. Following **END-MODULE**, no module is current. The specific actions taken by **END-MODULE** are implementation-dependent.

ENTRY: entry-colon I

("<spaces>name" —)

Define the entry point of the current module as the address of the procedure *name*. Store the entry point in the **MDF-ENTRY** field in the module header image. If there is no current module, or if *name* has not been defined, the compiler may report an error. Setting a module entry point is an optional procedure, needed only for an executable module.

EXPORT I

("<spaces>name" —)

Add the procedure *name* to the Module Export List, for the module being compiled. The procedure *name* becomes available for import by other modules. If *name* is undefined, the compiler may report an error.

FROM I

("<spaces>name" —)

Enter the specified module *name* into the currently compiling module's Module Import List (see OTA Volume 1, Section 6.4), and make *name* the currently selected module for importing into the module being defined. Following **FROM**, any procedures to be made available in the imported module must be explicitly specified with **IMPORT**. If the module *name* does not exist, or cannot be found, the compiler may report an error.

It is the programmer's responsibility to ensure that an appropriate file named *name***.DEF** is available to the compiler for generating import references. See Section 5.2 for a description of a **.DEF** file.

IMPORT I

("<spaces>name" —)

Import the procedure *name* from the currently selected import module. Only explicitly imported procedures are available to the current module. An imported procedure is available until it is redefined, either explicitly or by importing a word with the same name from a different module (see examples below). If *name* has not been exported by the currently selected import module, or if there is no imported module, the compiler may report an error.

MODULE: module-colon I

("<spaces>name" —)

Begin a module definition, given its *name* that follows in the input stream, and make it the current module. The *name* is used as the basis for various filenames generated by the compiler, and due consideration should be given to the length of, and characters within, the *name*.

MODULE-ID module-i-d I

("<spaces>identifier"—)

Set the Module Identifier for the current module, given its *identifier* that follows in the input stream. The format and meaning of a module identifier is the same as the Application Identifier (AID) defined by ISO/IEC Standard 7816-5, and is further described in Volume 1 of this Specification. A module must have its identifier set, and the **MODULE-ID** declaration must immediately follow the **MODULE:** definition. If there is no current module, or if *identifier* is not a valid module identifier, the compiler may report an error.

VERSION# version-number I

(num —)

Specify the version number *num* of the current module. The version number is arbitrary, but is limited to 16 bits (65535 or less). A module must have a version number set. If there is no current module, or if *num* is greater than 16 bits, the compiler may report an error.

Following are two examples of module definitions, illustrating the use of words described in this section.

Example 1

```
MODULE: MODTEST1
MODULE-ID F101010101
0100 VERSION#

FROM MODULE1
     IMPORT WORD1
     IMPORT WORD2
     IMPORT WORD3

FROM MODULE2
     IMPORT WORD4
     IMPORT WORD5
     IMPORT WORD6

: TEST1
     WORD1          \ from module 1
     WORD2          \ from module 1
     WORD3          \ from module 1
     WORD4          \ from module 2
     WORD5          \ from module 2
     WORD6          \ from module 2
```

```
;
ENTRY: TEST1
END-MODULE
```

Example 2

```
MODULE: MODTEST2
MODULE-ID F202020202
0101 VERSION#

FROM MODULE1
     IMPORT WORD1
     IMPORT WORD2

: TEST2
     WORD1     \ from module 1
     WORD2     \ from module 1
;

FROM MODULE2
     IMPORT WORD1

: TEST3
     WORD1     \ from module 2
     WORD2     \ from module 1
;

     IMPORT WORD3

FROM MODULE1
     IMPORT WORD1

: TEST4
     WORD1     \ from module 1
     WORD2     \ from module 1
     WORD3     \ from module 2
;

END-MODULE
```

5.2 The .DEF file format

The **.def** file is a plain text file produced by a token compiler, in addition to the **.mdf** file containing the actual tokens. The **.def** file enumerates the exported words of a module, to enable the token compiler to compile import references to it in a subsequent module compilation. The **.def** file is, essentially, a Forth source file with the exports made using the following words. The **.def** file will have one instance of **EXPORT-ID** and any number of instances of **EXPORT-**xx up to the maximum number of exported words.

EXPORT-ID

("<spaces>MID" —)

Defines the module identifier, MID, of the module associated with the **.def**

file. This must precede any subsequent **EXPORT-**xx declarations.

EXPORT-xx

(u "<spaces>name" —)

Specifies an exported word *name* from the module identifier given by **EXPORT-ID** and refers to it by function number *num*. The two letter code *xx* is used to identify the class of the word. The defining words associated with each code are given in Table 16.

Table 16: Codes identifying classes of exported words in a .DEF file

Code	Class
:	**:** (colon)
CN	**CONSTANT**
VR	**VARIABLE**
BF	**BUFFER**
2C	**2CONSTANT**
2V	**2VARIABLE**
CR	**CREATE**
US	**USER**
VL	**VALUE**
DB	**DB:**
NV	**NV-DB:**
CD	**COMPILED-DB:**
32	**32BIT:**
BY	**BYTE:**
ST	**STRING:**
TC	**[TLV-CREATE**
TV	**TLV:**
TB	**TLV-BIT:**
EX	**EXPORT**

5.3 Module Management Words

The words in this section provide for the storage and execution of OTA modules from within the VM.

+MODULE plus-module T **MODAPPEND**

(c-addr len —)

Append the contents of the buffer, defined by *c-addr* and *len*, to an internal module-acquisition buffer. An **OUT-OF-MEMORY** exception may be thrown if the module buffer capacity is exceeded.

-MODULE minus-module T **MODRELEASE**

(—)

Release the resources used by the internal module-acquisition buffer. This is required, if module loading must be terminated prematurely by the application without registering the module in the module repository.

/MODULE slash-module T **MODINIT**

(—)

Prepare for reception of a new module into the repository. Details are implementation dependent.

?MODULES question-modules T **MODCHANGED**

(— u)

Return a value *u*, indicating whether any classes of modules (see Module ID Format discussion in Volume 1 of this Specification) have been updated. Bits 0–7 in *u* are separately set, if a module in the class Fn_H has been registered in the module repository since the last execution of this token. For example, a module registered with an initial module ID byte of $F4_H$ will set bit 4 in the return status. Bits 8–31 are reserved for future expansion.

ADD-MODULE T **MODREGISTER**

(— ior)

Register the module acquisition buffer in the module repository under the module's ID, which is found in its header. The resources associated with managing the module acquisition buffer are automatically released. The *ior* is zero, if the operation succeeded; for other *ior* values, see Appendix C.

CARD-MODULE T **MODCARDEXECUTE**

*(i*x a-addr — j*x flag)*

Load the module that is at *a-addr*. (*a-addr* is the internal storage address of an OTA module delivered from the card.) *flag* is true, if the module loaded successfully. *i*x* and *j*x* represent possible arguments to, and results from, execution of the module.

DEL-MODULE T **MODDELETE**

(c-addr len — ior)

Delete the module (whose ID is specified by *c-addr len)* from the module repository. The *ior* is zero, if the operation succeeded; for other *ior* values, see Appendix C.

DOMODULE T **MODEXECUTE**

*(i*x c-addr len — j*x flag)*

Load and execute a module from the module repository, using the module ID specified by *c-addr len*. *flag* is true, if the module was loaded successfully. See the section "Module Handling Services," in Volume 1 of this Specification, for an account of module loading and execution.

MDF-FLAGS m-d-f-flags T (Sequence)

(— num)

Return the VM flags in the currently selected record in the terminal's module database.

The sequence may be implemented as **DBCFETCHLIT 2**.

MDF-ID m-d-f-i-d T (Sequence)

(— c-addr len)

Return at *c-addr len* the MID in the currently selected record in the terminal's module database.

The sequence may be implemented as **DBSTRFETCHLIT 4 16**.

MDF-IDLEN m-d-f-i-d-len T (Sequence)

(— num)

Return the size of the MID in the currently selected record in the terminal's module database.

The sequence may be implemented as **DBCFETCHLIT 3**.

MDF-VER m-d-f-ver T (Sequence)

(— num)

Return the version number of the currently selected record in the terminal's module database.

The sequence may be implemented as **DBSTRFETCHLIT 0 2 BNFETCH**.

MODULES T **MODULES**

(—)

Make the terminal's module database the current database. This is a read-only, non-volatile database.

Appendix A: Bibliography

American National Standard For Information Systems: Programming Language Forth (ANSI X3.215-1994). American National Standards Institute, 11 W. 42nd St., New York, NY 10036, (212) 642-4900.

Brodie, L. *Starting FORTH*, Englewood Cliffs, NJ: Prentice-Hall, 1981, 2nd Ed. 1987.

Brodie, L. *Thinking FORTH*, Englewood Cliffs, NJ: Prentice-Hall, 1984, reprinted by the Forth Interest Group, P. O. Box 2154, Oakland, CA 94621, 1994.

Kelly, M. G., and Spies, N. *FORTH: A Text and Reference*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Koopman, P. *Stack Computers, The New Wave*. Chichester, West Sussex, England. Ellis Horwood Ltd. 1989

Martin, T. *A Bibliography of Forth References*, 3rd Ed. Rochester, NY: Institute for Applied Forth Research, 1987.

Moore, C. H. "The Evolution of Forth—An Unusual Language" *Byte*, August 1980.

Noble, J. V. *Scientific Forth*. Charlottesville, VA: Mechum Banks Publishing, 1992.

Pountain, R. *Object Oriented Forth*. New York: Academic Press, 1987.

Rather, E. D. "Forth Programming Language" *Encyclopedia of Physical Science & Technology* (V. 5) Academic Press, Inc., 1987, 1992.

Rather, E. D., Colburn, D. R., and Moore, C. H. "The Evolution of Forth" *ACM SIGPLAN Notices*, Vol. 28, No. 3, March 1993; also reprinted in *The History of Programming Languages*, T. Bergin and R. Gibson, eds., New York: ACM Press, also Addison-Wesley Publishing Co.

SENDIT Tool Architecture and *SENDIT Token Specification*, reports from ESPRIT Project EP9152, 1995; available from MicroProcessor Engineering Ltd., 133 Hill Lane, Southampton S015 5AF England.

Terry, J. D. *Library of Forth Routines and Utilities*. New York: Shadow Lawn Press, 1986.

Tracy, M. and Anderson, A. *Mastering Forth* (2nd ed.). New York: Brady Books, 1989.

Appendix B: Required Forth Functions

This section presents an alphabetical list of Forth words, derived from the index (first) line of each entry in Sections 3 through 5. Each line contains, from left to right:

- Definition name in upper-case, monospaced, boldface letters;
- Natural-language pronunciation, if it is not obvious from the name;
- Special designators, as applicable.
  - A **ASSEMBLER** scope. These words are used to build and modify target definitions, expressed in CPU machine language or VM tokens.
  - C **COMPILER** scope. These words, which may be used in compiling mode only, are commonly used to construct program structures (e.g., loops and conditionals).
  - H **HOST** scope. These words are used to construct **COMPILER** or **INTERPRETER** definitions.
  - I **INTERPRETER** scope. These words are used to build data structures in the target image.
  - T **TARGET** scope, including all the functions to be executed in the target. These functions may be executed interpretively only on systems that support an *interactive* mode (see Section 2.7 and Section 3.2) when a suitable target is available.
  - F ANS Forth word (including all optional wordsets in that standard). Additional information about this word and its usage may be found in ANS Forth.
- Equivalent OTA token(s).
- Page number for the description of the word.

Word	Pronunciation	Codes	Tokens	Page
'	tick	F,H,I		3-21
-	minus	F,H,T	**SUB**	4-43
-CARD	no-card	T	**CARDABSENT**	4-64
-LIST	minus-list	T	**HOTDELETE**	4-88
-MODULE	minus-module	T	**MODRELEASE**	5-98
-ROT	minus-rote	H,T	**MINUSROT**	4-39
-TRAILING	minus-trailing	F,H,T	**MINUSTRAILING**	4-50
-ZEROS	minus-zeros	T	**MINUSZEROS**	4-50
!	store	F,H,T	**STORE**	4-41
!BCD	store-b-c-d	T	**BCDSTORE**	4-49
!BN	store-b-n	T	**BNSTORE**	4-49
!CN	store-c-n	T	**CNSTORE**	4-49
!IP	store-i-p	T	(Sequence)	4-58
!OP	store-o-p	T	**SETOP**	4-59
!PLUG-CONTROL	store-plug-control	T	(Sequence)	4-66
!TLV	store-t-l-v	T	**TLVSTORE**	4-84
!TLV-BIT	store-t-l-v-bit	T	**TLVBITSTORE**	4-84
!TLV-RAW	store-t-l-v-raw	T	**TLVSTORERAW**	4-84
#	number-sign	F,H,T	**NMBR**	4-62

Word	Pronunciation	Codes	Tokens	Page
#>	number-sign-greater	F,H,T	**NMBRGT**	4-62
#RECORD	number-record	T	(Sequence)	4-73
#S	number-sign-s	F,H,T	**NMBRS**	4-63
(	paren	F,H,C,I		3-18
*****	star	F,H,T	**MUL**	4-43
***/**	star-slash	F,H,T	(Sequence)	4-43
***/MOD**	star-slash-mod	F,H,T	(Sequence)	4-43
,	comma	F,H,I		3-23
,"	comma-quote	H,I		3-23
,BCD	comma-b-c-d	I		3-23
,BN	comma-b-n	I		3-23
.	dot	F,H,T	(Sequence)	4-59
."	dot-quote	F,H,T	(Sequence)	4-59
.R	dot-r	F,H,T	(Sequence)	4-59
/	slash	F,H,T	**DIV**	4-43
/CARD	slash-card	T	**CARDINIT**	4-64
/LIST	slash-list	T	**HOTINIT**	4-88
/MESSAGES	slash-messages	T	**MSGINIT**	4-78
/MOD	slash-mod	F,H,T	(Sequence)	4-43
/MODULE	slash-module	T	**MODINIT**	5-98
/STRING	slash-string	F,H,T	**SLASHSTRING**	4-50
:	colon	F,H,I	**PROC**	3-25
;	semi-colon	F,C,H,T	**ENDPROC**	3-26
?DO	question-do	F,C,H	**RQDO <+addr>**	3-32
?DUP	question-dupe	F,H,T	**QDUP**	4-40
?LIST	question-list	T	**HOTFIND**	4-88
?MODULES	question-modules	T	**MODCHANGED**	5-98
?THROW	question-throw	T	**QTHROW**	4-55
@	fetch	F,H,T	**FETCH**	4-41
@BCD	fetch-b-c-d	T	**BCDFETCH**	4-50
@BN	fetch-b-n	T	**BNFETCH**	4-50
@CN	fetch-c-n	T	**CNFETCH**	4-51
@IP	fetch-i-p	T	(Sequence)	4-59
@OP	fetch-o-p	T	**GETOP**	4-59
@TLV	fetch-t-l-v	T	**TLVFETCH**	4-84
@TLV-BIT	fetch-t-l-v-bit	T	**TLVBITFETCH**	4-85
@TLV-LENGTH	fetch-t-l-v-length	T	**TLVFETCHLENGTH**	4-85
@TLV-RAW	fetch-t-l-v-raw	T	**TLVFETCHRAW**	4-85
@TLV-TAG	fetch-t-l-v-tag	T	**TLVFETCHTAG**	4-85
@TLV-VALUE	fetch-t-l-v-value	T	**TLVFETCHVALUE**	4-85
[	left-bracket	F,C,H		3-30
**['] **	bracket-tick	F,C,H	**ELITC** <+addr>	3-30
[CHAR]	bracket-char	F,C,H	**SLIT** <char>	3-30
[DEFINED]	bracket-defined	H,C,I		3-18
[ELSE]	bracket-else	F,H,C,I		3-18
[IF]	bracket-if	F,H,C,I		3-19
[KEY	bracket-key	I		4-72
[SEQUENTIAL]	bracket-sequential	I		4-73
[THEN]	bracket-then	F,H,C,I		3-19
[TLV	bracket-t-l-v	I		4-83
[TLV-CREATE	bracket-t-l-v-create	I	(Sequence)	4-83
[UNDEFINED]	bracket-undefined	H,C,I		3-19
****	backslash	F,H,C,I		3-19
]	right-bracket	F,H,I		3-30
+	plus	F,H,T	**ADD**	4-43
+!	plus-store	F,H,T	**INCR**	4-43
+LIST	plus-list	T	**HOTADD**	4-88
+LOOP	plus-loop	F,C,H	**RPLUSLOOP**	3-31
+MODULE	plus-module	T	**MODAPPEND**	5-98
+STRING	plus-string	T	**PLUSSTRING**	4-50
<	less-than	F,H,T	**CMPLT**	4-47

Word	Pronunciation	Codes	Tokens	Page
<#	less-number-sign	F,H,T	**LTNMBR**	4-63
<=	less-than-or-equal	F,H,T	**CMPLE**	4-47
<>	not-equal	F,H,T	**CMPNE**	4-47
=	equal	F,H,T	**CMPEQ**	4-47
>	greater-than	F,H,T	**CMPGT**	4-48
>=	greater-than-or-equal	F,H,T	**CMPGE**	4-48
>BODY	to-body	F,H,I		3-21
>NUMBER	to-number	F,H,T	**TONUMBER**	4-63
>R	to-r	F,H,T	**TOR**	4-41
0<	zero-less-than	F,H,T	**SETLT**	4-48
0<=	zero-less-or-equal	H,T	**SETLE**	4-48
0<>	zero-not-equal	F,H,T	**SETNE**	4-48
0=	zero-equal	F,H,T	**SETEQ**	4-48
0>	zero-greater	F,H,T	**SETGT**	4-48
0>=	zero-greater-or-equal	H,T	**SETGE**	4-48
1-	one-minus	F,H,T	**SUBLIT1**	4-44
1+	one-plus	F,H,T	**ADDLIT1**	4-44
2!	two-store	F,H,T	**TWOSTORE**	4-41
2*	two-star	F,H,T	**SHL**	4-46
2/	two-slash	F,H,T	(Sequence)	4-46
2@	two-fetch	F,H,T	**TWOFETCH**	4-42
2>R	two-to-r	F,H,T	**TWOTOR**	4-41
2CONSTANT	two-constant	F,H,T	(Sequence)	3-26
2DROP	two-drop	F,H,T	**TWODROP**	4-39
2DUP	two-dupe	F,H,T	**TWODUP**	4-39
2OVER	two-over	F,H,T	**TWOOVER**	4-39
2R@	two-r-fetch	F,H,T	**TWORFETCH**	4-41
2R>	two-r-from	F,H,T	**TWORFROM**	4-41
2ROT	two-rote	F,H,T	**TWOROT**	4-39
2SWAP	two-swap	F,H,T	**TWOSWAP**	4-39
2VARIABLE	two-variable	F,H,T	**ELITU** <+addr>	3-26
32BIT:	32-bit-colon	I	**ELIT** <num1> <method>	4-68
A"	a-quote	I		3-24
ABS	abs	F,H,T	**ABS**	4-44
ADD-MODULE		T	**MODREGISTER**	5-98
ADD-NAME		T	**DBADDBYKEY**	4-75
ADD-RECORD		T	**DBADDREC**	4-73
AGAIN		F,C,H	**BRA** <+addr>	3-32
ALIGN		F,H,I		3-24
ALIGNED		F,H,T	(Sequence)	4-42
ALLOT		F,H,I		3-24
AND		F,H,T	**AND**	4-46
ASSEMBLER		I		3-17
AT-XY	at-x-y	F,H,T	(Sequence)	4-59
AVAILABLE		T	**DBAVAIL**	4-73
BASE		F,H,T	**LIT0 USERVAR**	4-63
BEEP		T	(Sequence)	4-60
BEGIN		F,C,H		3-32
BINARY		H,T	(Sequence)	3-20
BL		F,H,T	(Sequence)	4-60
BLANK		F,H,T	(Sequence)	4-51
BOUNDS		H,T	(Sequence)	4-90
BREAKPOINT		T	**BREAKPNT**	4-91
BUFFER:	buffer-colon	I	**ELITU** <+addr>	3-26
BYTE:	byte-colon	I	**ELIT** <num1> <method>	4-69
C!	c-store	F,H,T	**CSTORE**	4-42
C,	c-comma	F,H,I		3-24
C@	c-fetch	F,H,T	**CFETCH**	4-42
C+!	c-plus-store	H,T	(Sequence)	4-44
C+STRING	c-plus-string	T	(Sequence)**)**	4-51
CARD		T	**CARD**	4-64

Word	Pronunciation	Codes	Tokens	Page
CARD-MODULE		T	**MODCARDEXECUTE**	5-98
CARD-OFF		T	**CARDOFF**	4-64
CARD-ON		T	**CARDON**	4-64
CASE		F,C,H		3-32
CATCH		F,H,T	**CATCH**	4-54
CELL		H,T	**LIT4**	3-24
CELL+	cell-plus	F,H,T	**SADDLIT 4**	4-44
CELLS		F,H,T	**SMULLIT 4**	4-42
CEXTEND	c-extend	T	**CEXTEND**	4-53
CHAR	char	F,H,I		3-21
CHAR+	char-plus	F,H,T	**ADDLIT1**	4-44
CHARS	chars	F,H,T		4-90
CHOOSE-LANGUAGE		T	**CHOOSELANG**	4-76
CLOSE		T	**DEVCLOSE**	4-57
CN"	c-n-quote	I		3-24
CODE		F,H,I	**PROC**	3-27
CODE-PAGE?	code-page-question	T	**CODEPAGE**	4-77
CODE!	code-store	I		3-24
CODE,	code-comma	I		3-24
COMPARE		F,H,T	**COMPARE**	4-51
COMPILED-DB:	compiled-d-b-colon	I	(Sequence)	4-70
COMPILER		I		3-17
CONSTANT		F,H,I	**ELIT** <x>	3-27
COUNT		F,H,T	**COUNT**	4-51
CR		F,H,T	(Sequence)	4-60
CREATE		F,H,I	(Sequence)	3-27
CRYPTO		T	**CRYPTO**	4-89
CS-PICK	c-s-pick	F,C,H		3-32
CS-ROLL	c-s-roll	F,C,H		3-33
D+	d-plus	F,H,T	**DADD**	4-45
D<	d-less-than	F,H,T	**DCMPLT**	4-45
DABS	d-abs	F,H,T	(Sequence)	4-45
DATA!	data-store	I		3-25
DATA,	data-comma	I		3-25
DB:	d-b-colon	I	(Sequence)	4-70
DBSIZE	d-b-size	T	**DBSIZE**	4-74
DECIMAL		F,H,T	(Sequence)	3-20
DELETE-MESSAGES		T	**MSGDELETE**	4-78
DELETE-RECORD		T	**DBDELREC**	4-74
DEL-MODULE		T	**MODDELETE**	5-98
DEL-NAME		T	**DBDELBYKEY**	4-76
DEPTH		F,H,T	**DEPTH**	4-90
DEVBREAK		T	**DEVBREAK**	4-62
DEVCALL		T	**DEVCONNECT**	4-62
DEVHANGUP		T	**DEVHANGUP**	4-62
DNEGATE	d-negate	F,H,T	**DNEGATE**	4-45
DO		F,C,H	**RDO** <+addr>	3-33
DOES>	does	F,C,H	(Sequence)	3-28
DOMODULE		T	**MODEXECUTE**	5-99
DROP		F,H,T	**DROP**	4-40
DUP	dupe	F,H,T	**DUP**	4-40
EKEY	e-key	F,H,T	(Sequence)	4-60
EKEY?	e-key-question	F,H,T	(Sequence)	4-60
ELSE		F,C,H	**BRA** <+addr>	3-33
EMIT		F,H,T	(Sequence)	4-60
ENDCASE	end-case	F,C,H	**DROP**	3-33
END-CODE		F,H,A	**ENDPROC**	3-28
END-DATA		I		4-71
END-DB	end-d-b	I		4-71
END-MODULE		I		5-94
ENDOF	end-of	F,C,H	**BRA** <+addr>	3-34

Word	Pronunciation	Codes	Tokens	Page
ENTIRE		T	(Sequence)	4-74
ENTRY:	entry-colon	I		5-94
EQU	e-q-u	H	**LIT** <x>	3-29
ERASE		F,H,T	(Sequence)	4-52
EXECUTE		F,H,T	**ICALL**	4-90
EXIT		F,H,T	**RETURN**	4-90
EXPORT		I		5-94
EXPORT-ID		I		5-96
EXPORT-xx		I		5-97
EXTEND		T	**EXTEND**	4-54
FALSE		F,H,T	**LIT0**	4-91
FILL		F,H,T	**FILL**	4-52
FIND		F,H,I		3-21
FIND-NAME		T	**DBMATCHBYKEY**	4-76
FROM		I		5-94
GET-MESSAGE		T	**MSGFETCH**	4-78
GET-MSECS	get-m-secs	T	**GETMS**	4-55
GOTO	go-to	T	**IJMP**	4-91
HERE		F,H,I		3-25
HEX		F,H,T	(Sequence)	3-20
HEX"	hex-quote	I,C,H	(Sequence)	4-52
HOLD		F,H,T	**HOLD**	4-63
HOST		I		3-17
I		F,C,H,T	**RI**	3-34
IDATA	i-data	I		3-22
IF		F,C,H	**BZ** <+addr>	3-34
IMMEDIATE		F,H,I		3-21
IMPORT		I		5-94
INCLUDE		H,I		3-18
INITIALIZE		T	**DBINIT**	4-75
INTERPRETER		I		3-17
INTO		C,I	(Sequence)	4-66
INVERT		F,H,T	(Sequence)	4-46
IOCTRL	i-o-control	T	**DEVIOCTL**	4-57
J		F,C,H,T	**RJ**	3-34
KEY]	key-bracket	I		4-71
LANGUAGE-CNAME	language-c-name	T	(Sequence)	4-77
LANGUAGE-NAME		T	(Sequence)	4-77
LANGUAGE-PAGE		T	(Sequence)	4-77
LANGUAGE-SYMBOL		T	(Sequence)	4-77
LANGUAGES		T	**LANGUAGES**	4-77
LEAVE		F,C,H,T	**RLEAVE**	3-34
LITERAL		F,C,H,C	**ELIT** <num>	3-31
LOAD-MESSAGES		T	**MSGLOAD**	4-78
LOAD-PAGE		T	**LOADPAGE**	4-78
LOCALS|	locals-bar	F,C,H		3-37
LOOP		F,C,H	**RLOOP**	3-34
LSHIFT	l-shift	F,H,T	**SHLN**	4-46
M*	m-star	F,H,T	**MMUL**	4-45
M/MOD	m-slash-mod	H,T	**MSLMOD**	4-45
MAGREAD		T	**MAGREAD**	4-65
MAGWRITE		T	**MAGWRITE**	4-66
MARKER		F,H,I		3-18
MAX		F,H,T	**MAX**	4-44
MDF-FLAGS	m-d-f-flags	T	(Sequence)	5-99
MDF-ID	m-d-f-i-d	T	(Sequence)	5-99
MDF-IDLEN	m-d-f-i-d-len	T	(Sequence)	5-99
MDF-VER	m-d-f-ver	T	(Sequence)	5-99
MIN		F,H,T	**MIN**	4-44
MOD		F,H,T	**MOD**	4-44
MODULE-ID	module-i-d	I		5-95

Word	Pronunciation	Codes	Tokens	Page
MODULE:	module-colon	I		5-95
MODULES		T	**MODULES**	5-99
MOVE		F,H,T	**MOVE**	4-52
MS	m-s	F,H,T	**MS**	4-55
NEGATE		F,H,T	**NEGATE**	4-44
NIP		F,H,T	**NIP**	4-40
NOT		H,T	**SETEQ**	4-46
NV-DB:	n-v-d-b-colon	I	(Sequence)	4-71
OF		F,C,H	**ROF** <+addr>	3-35
OPEN		T	**DEVOPEN**	4-57
OR		F,H,T	**OR**	4-47
OSCALL	o-s-call	F,H,T	**OSCALL**	4-89
OUTPUT		T	**DEVOUTPUT**	4-58
OVER		F,H,T	**OVER**	4-40
PAGE		F,H,T	(Sequence)	4-60
PICK		F,H,T	**PICK**	4-40
PLUG		C,I	**ELITC** <+addr>	4-67
POSTPONE		F,C,H		3-31
PRINT		C	(Sequence)	4-61
R@	r-fetch	F,H,T	**RFETCH**	4-41
R>	r-from	F,H,T	**RFROM**	4-41
READ		T	**DEVREAD**	4-58
RECORDS		T	(Sequence)	4-75
RECURSE		F,H,T	**CALL** <+addr>	3-31
RELEASE		T	**RELEASE**	4-54
REPEAT		F,C,H	**BRA** <+addr>	3-35
RESTORE-DB	restore-d-b	T	**DBRESTORE**	4-75
RESTORE-INPUT		F,H,I		3-21
ROT	rote	F,H,T	**ROT**	4-40
RSHIFT	r-shift	F,H,T	**SHRNU**	4-47
RSHIFTS	r-shifts	F,H,T	**SHRN**	4-47
S"	s-quote	F,H,C,I	**STRLIT**	4-52
S>D	s-to-d	F,H,T	(Sequence)	4-45
SAVE-DB	save-d-b	T	**DBSAVE**	4-75
SAVE-INPUT		F,H,I		3-22
SCAN		H,T	**SCAN**	4-53
SELECT		T	**DBSELECT**	4-75
SET-TIME&DATE	set-time-and-date	F,H,T	**SETTIME**	4-55
SETCALLBACK		T	**SETCALLBACK**	4-90
SIGN		F,H,T	**SIGN**	4-63
SKIP		H,T	**SKIP**	4-53
SM/REM	s-m-slash-rem	F,H,T	**MSLMOD**	4-45
SOCKET		I	**DOSOCKET** <n>	4-67
SPACE		F,H,T	(Sequence)	4-61
SPACES		F,H,T	(Sequence)	4-61
STATUS		T	**DEVSTATUS**	4-58
STRING:	string-colon	I	(Sequence)	4-71
SWAP		F,H,T	**SWAP**	4-40
TARGET		I		3-17
THEN		F,C,H		3-35
THROW		F,H,T	**THROW**	4-54
TICKET		T	(Sequence)	4-61
TIME&DATE	time-and-date	F,H,T	**GETTIME**	4-55
TIMED-READ		T	**DEVTIMEDREAD**	4-57
TLV+DOL	t-l-v-plus-d-o-l	T	**TLVPLUSDOL**	4-86
TLV+STRING	t-l-v-plus-string	T	**TLVPLUSSTRING**	4-86
TLV-AN	t-l-v-a-n	T	**LIT4**	4-82
TLV-ANS	t-l-v-a-n-s	T	**LIT5**	4-82
TLV-B	t-l-v-b	T	**LIT1**	4-82
TLV-BIT-CLEAR		T	**TLVBITCLEAR**	4-85
TLV-BIT-SET		T	**TLVBITSET**	4-86

Word	Pronunciation	Codes	Tokens	Page
TLV-BIT:	t-l-v-bit-colon	I	(Sequence)	4-82
TLV-BN	t-l-v-b-n	T	**LIT2**	4-82
TLV-CLEAR	t-l-v-clear	T	**TLVCLEAR**	4-86
TLV-CN	t-l-v-c-n	T	**LIT3**	4-82
TLV-FIND	t-l-v-find	T	**TLVFIND**	4-86
TLV-FORMAT	t-l-v-format	T	**TLVFORMAT**	4-87
TLV-INITIALIZE	t-l-v-initialise	T	**TLVINIT**	4-87
TLV-N	t-l-v-n	T	**LIT0**	4-82
TLV-PARSE	t-l-v-parse	T	**TLVPARSE**	4-87
TLV-STATUS	t-l-v-status	T	**TLVSTATUS**	4-87
TLV-TAG	t-l-v-tag	T	**TLVTAG**	4-87
TLV-TRAVERSE	t-l-v-traverse	T	**TLVTRAVERSE**	4-88
TLV-VAR	t-l-v-var	T	**LIT6**	4-83
TLV:	t-l-v-colon	I	**ELITD** <addr>	4-83
TLV]	t-l-v-bracket	I		4-83
TO		F,T	(Sequence)	4-42
TRUE		F,H,T	**LITMINUS1**	4-91
TUCK		F,H,T	**TUCK**	4-40
TYPE		F,H,T	(Sequence)	4-61
U/	u-slash	F,H,T	**DIVU**	4-44
U<	u-less-than	F,H,T	**CMPLTU**	4-48
U<=	u-less-than-or-equal	H,T	**CMPLEU**	4-48
U>	u-greater-than	F,H,T	**CMPGTU**	4-48
U>=	u-greater-than-or-equal	H,T	**CMPGEU**	4-49
UDATA	u-data	I		3-22
UDATA!	u-data-store	I		3-25
UDATA,	u-data-comma	I		3-25
UM*	u-m-star	F,H,T	**MMULU**	4-46
UM/MOD	u-m-slash-mod	F,H,T	**MSLMODU**	4-46
UMOD	u-mod	H,T	**MODU**	4-45
UNTIL		F,C,H	**BZ** <+addr>	3-35
UPDATE-MESSAGES		T	**MSGUPDATE**	4-78
USER		H,I	**LIT** <num> **USERVAR**	3-29
VALUE		F,H,I	**ELITD** <+addr> <method>	3-29
VARIABLE		F,H,I	**ELITU** <+addr>	3-30
VERSION#	version-number	I		5-95
WHILE		F,C,H	**BZ <+addr>**	3-35
WIDEN		F,T	**WIDEN**	4-47
WITHIN		F,H,T	**WITHIN**	4-49
WORD		F,H		3-22
WRITE		T	**DEVWRITE**	4-58
XOR		F,H,T	**XOR**	4-47

Appendix C: Exception Codes

This section includes all codes used as arguments to the standard exception-handling token **THROW**, and all I/O-related status codes. Status codes are normally used as returned *ior* (I/O result) values, but may be used as throw codes.

ANS Forth specifies allowable **THROW** values as follows: Values {-255...-1} shall be used only as assigned by ANS Forth (Table 9.2 in ANSI X3.215). The values {-4095...-256} shall be used only as assigned by a system. Positive values are available to programs.

Table 17 shows the ANS Forth codes used in OTA kernels. *Name* is a Forth word that returns the given value; usually this will be a simple **CONSTANT** but, on some systems, additional functionality may be provided. The additional *Description* is included only if the name is not considered self-explanatory.

Table 18 gives the OTA-specific throw codes, and Table 19 gives the OTA-specific I/O status codes assigned in this Specification. Values are given in both hex and decimal ("Dec."); the hex notation emphasises the OTA list organisation by device type and error category. *Note:* the hex notation in all tables omits the leading four Fs in a cell value (i.e., a hex value shown as FFCB is the 32-bit number FFFFFFCB). For OTA-specific codes, the least-significant byte gives the error code within the category, and the next-significant byte gives the category.

Table 17: ANS Forth THROW codes in OTA kernels

Dec.	Hex	Name	Description
-53	FFCB	**STACK-OVERFLOW**	Exception stack overflow.
-24	FFE8	**INVALID-NUMERIC-ARG**	Invalid numeric argument.
-23	FFE9	**ADDRESS-ALIGNMENT**	Address mis-alignment detected.
-21	FFEB	**UNSUPPORTED-OPERATION**	Invalid operation on this terminal.
-17	FFEF	**OUTPUT-STRING-OVERFLOW**	Pictured numeric output string overflow.
-11	FFF5	**OUT-OF-RANGE**	Result out of range.
-10	FFF6	**ZERO-DIVIDE**	Division by zero.
-9	FFF7	**INVALID-ADDRESS**	Invalid memory address.
-7	FFF9	**TOO-MANY-DO-LOOPS**	**DO** loops nested too deeply during execution.
-6	FFFA	**RETURN-STACK-UNDERFLOW**	
-5	FFFB	**RETURN-STACK-OVERFLOW**	
-4	FFFC	**DATA-STACK-UNDERFLOW**	
-3	FFFD	**DATA-STACK-OVERFLOW**	

Table 18: OTA `THROW` codes

Dec.	Hex	Name	Description
-4096	F000	**ERRCLASS-DB**	General database error classification.
-4095	F001	**DB-INVALID-RECORD**	Record number outside the range defined for the current structure (zero to **AVAILABLE** -1).
-4094	F002	**DB-INVALID-FUNCTION**	Function inappropriate for this kind of database (e.g., **DBADDBYKEY** on a non-ordered structure).
-4093	F003	**DB-CREATION-ERROR**	An NV structure could not be initialised (e.g., file marked read-only).
-4092	F004	**DB-ACCESS-ERROR**	An NV structure could not be accessed (e.g., error in opening the file).
-4091	F005	**DB-CLOSE-ERROR**	An NV structure could not be closed.
-4090	F006	**DB-SEEK-ERROR**	A selected record could not be found in an NV structure.
-4089	F007	**DB-READ-ERROR**	An NV structure could not be read.
-4088	F008	**DB-WRITE-ERROR**	An NV structure could not be written.
-4087	F009	**DB-NOT-SELECTED**	Attempt to use a database procedure prior to first use of **DBMAKECURRENT**.
-3840	F100	**ERRCLASS-TLV**	General TLV error classification.
-3839	F101	**UNDEFINED-TLV**	Cannot find TLV tag name.
-3838	F102	**TLV-TOO-LARGE**	TLV value field is longer than 252 bytes.
-3837	F103	**DUPLICATE-TLV**	A module contains a TLV definition that already exists.
-3584	F200	**ERRCLASS-MOD**	General module error classification.
-3583	F201	**MOD-CANNOT-LOAD**	Cannot load module.
-3582	F202	**MOD-CANNOT-ADD**	Cannot add module.
-3581	F203	**MOD-OPEN-ERROR**	Cannot open repository.
-3580	F204	**MOD-CREATE-ERROR**	Error creating repository.
-3579	F205	**MOD-READ-ERROR**	Error reading repository.
-3578	F206	**MOD-CLOSE-ERROR**	Error closing repository.
-3577	F207	**MOD-INVALID-LENGTH**	Invalid module length.
-3576	F208	**MOD-DELETE-NOT-ALLOWED**	Deletion of module not allowed.
-3575	F209	**MOD-BAD-CARD-MODULE**	Bad card module.
-3574	F20A	**MOD-INVALID-TOKEN**	Invalid token.
-3573	F20B	**MOD-INVALID-MDF**	Not a valid module.
-3572	F20C	**MOD-RELOCATE**	Relocation error.

Table 18: OTA `THROW` codes *(continued)*

Dec.	Hex	Name	Description
-3571	F20D	**MOD-LOAD-ERROR**	Error loading module.
-3570	F20E	**MOD-INV-OVERRIDE**	Invalid override.
-3569	F20F	**MOD-NOT-AVAILABLE**	Module not in repository.
-3567	F211	**MOD-INVALID-MID-CHAR**	Invalid character in MID.
-3566	F212	**MOD-INVALID-MID**	Invalid MID string format.
-3565	F213	**MOD-CANT-EXECUTE**	Cannot execute module.
-3564	F214	**MOD-IN-USE**	Module in use.
-3328	F300	**ERRCLASS-SOC**	General socket error classification.
-3327	F301	**SOC-INV-SOCKET**	Invalid socket.
-3072	F400	**ERRCLASS-MEM**	General memory error classification.
-3071	F401	**OUT-OF-MEMORY**	Request to allocate memory in data space cannot be satisfied.
-3070	F402	**RELEASE-ERROR**	Memory release error.
-3069	F403	**RESIZE-ERROR**	Memory resize error.
-3068	F404	**BAD-HANDLE**	Bad memory handle.
-3067	F405	**WRITE-OUT-OF-RANGE**	Attempt to write outside allocated memory.
-3066	F406	**FRAME-STACK-ERROR**	Frame of the requested size could not be built.
-512	FE00	**ERRCLASS-MISC**	General miscellaneous error classification.
-511	FE01	**ILLOP**	Illegal instruction byte code.
-510	FE02	**TOO-MANY-LANGUAGES**	Language file full. Insufficient space in terminal language tables.
-509	FE03	**OUT-OF-CONTEXT**	Attempt to use a token out of proper sequence.
-508	FE04	**UNIMPLEMENTED**	Feature not implemented. Reference to a feature or device not supported in this kernel.
-507	FE05	**STRING-TOO-LARGE**	String size is greater than 255 bytes.
-506	FE06	**DIGIT-TOO-LARGE**	Digit not within number conversion base.

Table 19: OTA I/O Return Codes

Dec.	Hex	Name	Description
0	0	**SUCCESS**	Successful operation.
-32768	8000	**ERRCLASS-GENERIC**	Generic device error classification.
-32767	8001	**DEV-NOT-PRESENT**	Device not present.
-32766	8002	**DEV-TIMEOUT**	Time-out occurred during I/O operation.
-32765	8003	**DEV-CANCELLED**	Operation cancelled by user.
-32764	8004	**DEV-ERROR**	Miscellaneous device error.
-32763	8005	**DEV-UNSUPPORTED**	Device not supported in this kernel.
-32762	8006	**DEV-NOT-INITIALISED**	Device must be initialised before use.
-32761	8007	**DEV-BUSY**	Device is busy.
-32760	8008	**INSUFFICIENT-RESOURCES**	Insufficient resources to carry out operation.
-32759	8009	**DEV-MUST-BE-OPENED**	Device must be opened.
-32758	800A	**DEV-ALREADY-OPEN**	Device already open.
-32757	800B	**DEV-CANNOT-OPEN**	Device can not be opened.
-32239	8211	**OUTSIDE-BORDER**	Outside border.
-32000	8300	**ERRCLASS-PRN**	General printer error classification.
-31983	8311	**PRN-OFFLINE**	Printer not selected.
-31982	8312	**PRN-OUT-OF-PAPER**	Printer reports out of paper.
-31981	8313	**PRN-ERROR-ASSERTED**	Printer has asserted error signal.
-31980	8314	**PRN-NOT-CONNECTED**	Printer does not appear to be connected.
-31744	8400	**ERRCLASS-MDM**	General modem error classification.
-31727	8411	**MDM-NO-CONNECTION**	No connection for on-line commands.
-31726	8412	**MDM-INVALID-PARM**	Invalid serial or modem parameters.
-31725	8413	**MDM-NO-CARRIER**	No carrier detected
-31724	8414	**MDM-NO-ANSWER**	No answer at destination.
-31723	8415	**MDM-BUSY**	Busy signal detected.
-31722	8416	**MDM-NO-DIAL-TONE**	No dial tone detected
-31721	8417	**MDM-ALREADY-OPEN**	Connection already established.
-31720	8418	**MDM-CONN-IN-PROGRESS**	Connection in progress.
-31488	8500	**ERRCLASS-ICC**	General ICC error classification.
-31471	8511	**ICC-MUTE**	Card does not respond.
-31470	8512	**ICC-NOT-PRESENT**	No card in reader.
-31469	8513	**ICC-COMM-ERROR**	Transmission error to card.

Table 19: OTA I/O Return Codes *(continued)*

Dec.	Hex	Name	Description
-31468	8514	**ICC-BUFFER-OVERFLOW**	Card buffer overflow.
-31467	8515	**ICC-PROTOCOL-ERROR**	Protocol error
-31466	8516	**ICC-NO-STATUS-WORD**	ICC response has no status word.
-31465	8517	**ICC-INVALID-BUFFER**	Invalid buffer.
-31464	8518	**ICC-OTHER-ERROR**	Other card error.
-31463	8519	**ICC-PARTIALLY-INSERTED**	Card partially in reader.
-31232	8600	**ERRCLASS-MAG**	General magstripe error classification.
-31215	8611	**MAG-COMM-ERROR**	Communications error between reader and magstripe.
-31214	8612	**MAG-BUFF-OVERFLOW**	Output buffer overflow.
-31213	8613	**MAG-WRITE-ERROR**	Write operation failed.
-30702	8812	**NON-DEPLETING-POWER**	Non-depleting power source.
-30464	8900	**ERRCLASS-VEND**	General vending error classification.
-30455	8909	**VEND-TERMINATE**	Vending terminated.

Index

aligned address 2–4
application 2–4

BASE 4–51
big-endian 2–4
binary, in EMV v2.0 2–4
blacklist management 4–88

C programming language 2–5
card selected services 2–5
CATCH 4–54
cell 2–5
cell pair 2–5
code page 4–76
code space 2–5
communications services 4–61
comparison operations 4–47
compile 2–5
compiler directives 3–17
compressed numeric 2–5
console I/O functions 4–58
control structures 3–31
counted strings 2–5, 4–49
cross-compilation 2–5
cryptographic algorithms 4–89

data field 2–8
data space 2–5
    access operations 4–41
    management 3–23
data stack 2–9
    operations 4–39
data types 2–9
database services 4–68
date and time functions 4–55
debug support 4–91
defining words 3–25
definition 2–8
development system 2–5, 2–7
dictionary 2–8
    management 3–30
double-precision arithmetic 4–45

exception
    frame 2–5
    handling 4–54
    stack 2–5
execution token 2–6
extensible memory management 4–53

host 2–6

I/O
    device 4–56
    operations 4–56
ICC I/O functions 4–64
immediate word 3–21
initialized data 3–27
instantiate 2–6
interpreter control functions 3–19

kernel 2–6

language selection 4–76
library 2–6
logical operations 4–46

magnetic stripe I/O functions 4–64
messages 4–78
modem functions 4–61
module 2–6, 5–93
    repository 2–6

name space 2–8
numeric formats 2–6, 4–79
    functions 4–62

program execution control 4–66

return stack 2–9, 4–40

single-precision arithmetic 4–42
socket 4–66
stack 2–7, 2–9
    notation 2–11
string functions 4–49, 4–84

target 2–7
terminal program 2–7
terminal resident services 2–7
terminal selected services 2–7
terminal, defined 2–7
THROW 2–5, 4–54
TLV (Tag Length Value) data structures

services 4–79
tokens 2–7
compiler 2–7
equivalent to words 2–11
loader/ interpreter (TLI) 2–7
transaction 2–7

user variables 3–28, 3–29

virtual machine 2–8

word 2–8

Control of Document

Document Name	Dept. Owner	Version Date	Master Print Date
Open Terminal Architecture (OTA) Specification, *Volume 2: Forth Language Programming Interface*	Europay – Smart Card Development	16/2/99	16 February 1999

Original Author	Dept.	Author Role	First Created
Information Systems			12/19/96 5:34 PM

Signoff Authorities	Dept	Role	Date	Signature

Version History

Version No.	Version Date	Change Request Number	Change Description
Version 1.0	Jan. 30, 1996		Submitted Version
Version 2.0	19 May 1997		Review Version
Version 3.0	19 June 1998		OTA 3 compliant

Input Documents

The following documents were used to produce this document:

- *Integrated Circuit Card Specifications for Payment Systems*, Version 3.0, June 30, 1996, Parts 1–3
- *Integrated Circuit Card Terminal Specification for Payment Systems*, Version 3.0, June 30, 1996
- *Europay Open Terminal Architecture (OTA) System Architecture*, Version 1.0, October 30, 1995

- *Europay Open Terminal Architecture (OTA) Specification, Volume 1: Virtual Machine Specification*, Version 3.2, May 13, 1998