

Antoni Urban

USM LANGUAGE Reference Manual



<http://www.softm.co.za>

USM is dedicated to You

Published by Software Makers
Johannesburg – South Africa
Draft Edition, August 2005
Copyright 2005 Antoni Urban
ALL RIGHTS RESERVED

<http://www.softm.co.za>

CONTENTS

1	INTRODUCTION	
1.1	About U SM Language	5
1.2	EFT Terminals	6
1.3	Interpreters	8
1.4	About This Manual	9
2	DEFINITION	
2.1	Syntax	10
2.1.1	Character Set	11
2.1.2	Numbers	12
2.1.3	Strings	13
2.1.4	Variables	14
2.1.5	Records	15
2.1.6	Fields	16
2.1.7	Methods	17
2.2	Semantics	18
2.2.1	Conditional Execution	19
2.2.2	Select Case	21
2.2.3	While Loop	23
2.2.4	Repeat	24
2.2.5	Method Call	26
2.2.6	Function Definition	28
2.3	Stack and Indent	30
2.4	Execution Model	32
3	OPERATORS	
3.1	Assign and Convert	36
	<code>.^ % </code> <code>.# .\$</code>	
3.2	Binary Operators	44
	<code>.and .or .xor .not</code> <code>.true .false</code>	
3.3	Relational Operators	49
	<code>.< .> .<= .>=</code> <code>.= .<></code>	
3.4	Arithmetical Operators	55
	<code>.+ .- .* ./ .mod</code>	

3.5	String Operators	60
	.& .def\$.len\$.pos	
	.del\$.ins\$.get\$.put\$	
3.6	Stack Operators	65
	.count .clear	
4	INTERPRETER SPECIFIC PROCEDURES	
4.1	Keyboard	68
	.AnyKey	
4.2	Screen	69
	.TestDisp	
	.Screen	
	.Menu	
	.Form	
4.3	Printer	73
	.TestPrint	
	.Receipt	
4.4	Card Reader	75
	.Track12	
	.Transact	
4.5	Communication Channels	78
	.DnLoadPC	
	.UpLoadPC	
5	PROGRAMMING SAMPLE	79
	INDEX	84

1. INTRODUCTION

ABOUT USM LANGUAGE

1.1

The USM language is classified as a simple interpretive programming language. It has been created after almost a decade of experimenting in the field of embedded software for EFT (Electronic Financial Transaction) terminals. A comparison can be made to the Adobe PostScript language, which was designed as page description standard.

The USM language can be applied for other small devices, although the *Reference Manual* will focus on the EFT terminals. The first interpreter of the language was built for the Cybernet JadeAire mobile terminal with GPRS capability. The second interpreter of USM runs on the Microsoft Windows platform and simulates JadeAire – offering testing facility to Cybernet software developers.

The following points will help you to understand the power of the USM language.

- ♦ **Compactness.** Terminals have limited resources, but are expected to provide more. EVD (Electronic Voucher Distribution), loyalty cards, smart vouchers, cheque verification, private labels cards, and many more applications run next to the core EFT functionality. Flexibility in multitasking and remote software upgrading require the compactness.
- ♦ **Full Execution Model.** USM is a general-purpose programming language with a strong execution model. Wide range of operators, interpreter specific procedures, programmable functions, stack, indent kind of nesting, select case, do-while loops, conditional execution, variables, convenient access to files and databases, and many more are offered by the language.
- ♦ **Readability.** The concept of the stack and original rules of structures nesting make a USM programme easy to be understood, even for somebody without programming knowledge and practice. Moreover, the programme can be edited by an internal text editor and immediately executed.
- ♦ **Simplicity of Interpretation.** The idea of execution indent simplifies the process of interpreting. As a result, interpreters are rather simple and small. It is very

important because of limited resources of small terminals.

- ♦ **Uniformity.** Strict syntax and semantic rules do not allow programmers to write monsters, understood by nobody, including themselves.

A USM programme is edited with any text editor either inside or outside a terminal. Provided the terminal has been equipped with a USM language interpreter, the programme can be launched, i.e. the interpreter commanded to scan the source code of the programme. It reads line by line, measures the indent, looks at the first character different than space and decides what to do. Sometimes the sequential progress is interrupted, and the scanner changes position in the programme. For example, if a condition is fulfilled the scanner will proceed to the next line, otherwise it will jump to the closest line with its indent not bigger than the condition line one. The interpreter will stop functioning after reaching the end of the USM programme.

Usually, the USM programme will be transmitted from a TMS (Terminal Management System) server to the terminal through one of the communication channels (serial cable, fix line modem, GSM, GPRS, etc.). The transmission is always initiated by the terminal, and supervised by a user, eliminating a chance to contract a virus. The USM programme received from the server is saved into a file with extension **U**. Several applications can be stored in a file system, and run on user's request.

To shorten the time of the transmission, and to increase the number of stored applications, the size of the interpreter as well as the size of applications must be minimal. The USM language, presented in this book, will give you the optimal result.

EFT TERMINALS

1.2

Everybody has seen a EFT terminal. It is called swipe-card machine, and looks to be brainless. Read information from a swiped card, ask a bank for approval, print a receipt, and... wait for the next card – it is the logic of the EFT terminal, for us as card holders. It seems to be completely different for somebody, who writes that “brainless” software. The EFT terminal programmer must have extensive knowledge of the following:

- (1) banking regulations (it must be either merchant or card holder losing money, the bank has to be untouchable);
- (2) merchant expectations (to be able to perform card-not-present transactions);

- (3) card holder expectations (each transaction must get a bank approval);
- (4) card types: credit, debit-savings, debit-cheque, petrol, bond, loyalty, private label, and so on;
- (5) transaction types: purchase, cash withdrawal, goods and cash, balance enquiry, reversal, refund, pre-authorisation;
- (6) modem language, mounting GPRS session, communication protocols;
- (7) message formats (e.g. ISO-8583), framing, CRC calculation, encapsulation, encryption, decryption, etc.;
- (8) PIN encryption (DUKPT, DES, 3-DES), PIN pad drivers, keys injection;
- (9) terminal supervising, i.e. reprinting, reporting, bank settlements, setup, user names and passwords, vendor details, tip facility, budget option and so on (there is no end here);
- (10) receipt layouts, logos, headers, and footers;
- (11) error diagnostic, handling and interpreting.

The above mentioned points have been selected from a much longer list of EFT terminals problems.

In addition, each terminal is like an ordinary computer with: operating system, external devices (screen, keyboard, printer, magnetic card reader, smart card reader, RS-232 port, clock, buzzer, etc.), drivers, file system, text editor, database system. Totally, software downloaded to the terminal has in size from 500 kb to 1 Mb. Moreover, we do not consider extra applications, which can take as much as it is available.

Coming back to the swiping operation, it is slightly more complicated, and can be described as follows. Validate user's name and password; read tracks of information and interpret the service code, expiry date, card number (compare it to the BIN allocation table); allow to make a manual entry and additionally ask for expiry date, CVV2 and bank authorisation ID (validate it); check if there is no difference between magnetic card number and embossed one; ask the user for the transaction type; prompt to enter an amount, tip, budget period, table number, product ID, unit price, quantity and many more depending how the terminal has been set up; if necessary ask for PIN and

perform encryption in the same way as it is done by the bank and with the same keys; connect to the bank or a switch; build a message in the right format, send and wait for a response (time out if there is no response); analyse the response and print a proper receipt; allow to perform so called immediate reversal in case something has gone wrong; print client's copy of the receipt and save the transaction to a buffer; give the option to reprint the last receipt; print a bank settlement or an end-of-day report; call a supervisor if the user is not allowed to do something.

Undoubtedly, it was the mistaken judgement that the terminal software was "brainless"!

INTERPRETERS

1.3

The USM language, as an interpretive language, must be executed by a programme classified as an interpreter. Because USM has been especially designed for EFT terminals, the interpreter resides in the permanent memory of the terminals. It must be coded in a low level language, and optimised for its speed and size.

The first task of the interpreter is to read and understand the USM language. Special scanning procedures must be designed for different types of objects, namely: numbers, strings, variables, conditions, procedures names, list of variables, white spaces, escape sequences, etc. Conversion procedures, define variable, read variable, push value to the operational stack, pop value from the operational stack, and many more should be offered as well. Additionally, scanning must have a very strong error diagnostic.

The second task of the interpreter is to execute structures, changing the sequential way of scanning. Break, return, function definition and call, select case variable and case value, repeat, loop, condition, and so on – are probably most important.

The third task of the interpreter is to recognise operators, and device specific procedures, to initiate calculations, and finally to pass results to output elements of that methods.

To achieve all of the above, the interpreter must operate: the stack, the state of processing array, variable list, dictionary of operators, procedures, and user defined functions.

You may learn more about interpreters, while practicing programming with the generic one, designed for the Microsoft Windows environment.

ABOUT THIS MANUAL**1.4**

This book, untitled *USM Language Reference Manual*, presents the syntax and semantics as well as detailed description of operators and procedures of Software Makers' "U" language. It also provides guidelines on how to build a USM interpreter. If you are interested in using the language, Software Makers will write for you the interpreter or will offer to you *USM Interpreter Technical Documentation* as well as the source code of the generic interpreter. Please visit Software Makers' site [HTTP://WWW.SOFTM.CO.ZA](http://www.softm.co.za) to find more information and contact details.

The second chapter of this book introduces an informal definition of the language, describing some basic ideas, syntax, semantics, concept of operational stack, original idea of execution indent, and skeleton of an interpreter. The description of the syntax contains important information about: character set, numbers, strings, variables names, records of a data file, fields of a record, and method names. The point about the semantics is divided for sub points: conditional execution, select case semaphore, do-while loop, repeated execution, how to define functions and how to call them.

All of the USM language built-in operators are fully described in chapter 3. The operators are categorised into the following groups: assign and convert, binary, relational, arithmetical, string, and stack. Each operator has its title, formal syntax definition, brief description, samples of usage, and important notes. The syntax definitions contain light and bold characters; words in light typeface are meta-terms only, and have to be substituted by real values; characters in bold typeface are explicit. To understand in full the syntax you must refer to the example, which is always presented in exact form. At the end of the book, operators are listed in alphabetical order (*INDEX*).

Chapter 4 *Interpreter Specific Procedures* describes procedures designed especially for the JadeAire interpreter – the first interpreter ever built. Similar procedures may be designed for other terminals or small devices. It is strongly advised to read the chapter, in which problems of global variables and operational stack are discussed. For somebody, who intends to build a new interpreter, chapter 4 will suggest how to proceed with practical implementation of the USM language.

The fifth and the last chapter of the book shows a sample of USM programmes. The sample was tested on the generic interpreter, which runs in the Microsoft Windows environment. The generic interpreter has been built to simulate the JadeAire interpreter in much more convenient environment.

2. DEFINITION

SYNTAX

2.1

Generally, a USM programme looks like a text, is readable, can be edited with any text editor. There are strong rules on how to write the programme, especially number of spaces at the beginning and the first character of a line are very important. But, there is also a little bit of freedom after the first character. You may use comment lines to improve readability. Lines must be separated by the carriage return character alone, or by the standard couple of characters – carriage return and line feed.

The USM explores so called execution indent, so you must be very careful while placing spaces at the front of programme lines. One space difference is not very clear, therefore you may use the tabulator character instead of the space. It will be interpreted as the space, but your programme will look much better. Some text editors can change the size of the tabulator – the best result is given by the size of two spaces.

Try to avoid long lines in the USM programme; sometimes you will be interested to use the terminal editor (small size of terminal's screen), or to print the source code on the terminal's printer (about 40 characters per line). Although terminals and other small computers become more powerful, with bigger memory, full colour and graphical screen – they are getting smaller in the same time. Therefore, any programming language must be compact, first of all.

Finally, I would like to share with you my programming experience. While writing programmes in any language, I apply the following rule: an algorithm must always fit on one A4 page. To achieve that, one must properly use modules. Even a very big and complex system, can be constructed in that way. The USM language offers you functions, and most importantly, they may have many elements on input and many elements on output. The USM programme is suitable for automatic conversion to a graphical form, it is possible to draw a flow chart while analysing the source code.

CHARACTER SET**2.1.1**

The character set of the USM language contains all printable characters from space to tilde, plus carriage return as programme lines separator (carriage return can be accompanied with line feed). You may use tabulator to increase indent, but it will be eventually converted to space. All other control characters (from 1 to 31) will be ignored. The wider range of characters, from 1 to 255, can be used inside strings, but only printable characters in the explicit form. Non printable characters must be declared by hexadecimal value.

There are some very important subsets of the character set, namely:

{ **0** | ... | **9** } – decimal digits;

{ **0** | ... | **9** | **A** | ... | **F** } – hexadecimal digits;

{ **A** | **B** | **C** | ... | **Z** } – uppercase letters;

{ **a** | **b** | **c** | ... | **z** } – lowercase letters;

{ **+** | **-** } – number signs;

{ **<** | **>** | **=** } – relational characters.

Some of the other characters have certain functions linked to them, for example:

? – conditional execution;

@ – do-while loop;

! – select case variable;

***** – another case value;

– repeat execution;

+ – operator, procedure, function call;

{ – function definition;

} – return or function end;

\ – break or exit;

' – beginning of comment;

% – *file%record* notation;

| – *record|field* notation;

or they are used as part of operators names, for example:

& – link strings;

+ – add numbers;

– – subtract numbers;

/ – divide numbers;

* – multiply numbers;

\$ – convert to string;

– convert to number;

and so on. Any character, different than space, period, single quote, double quote, and backslash, can be used to name methods, i.e. operators, procedures and functions.

NUMBERS

2.1.2

The USM language operates on integer numbers stored in 4 bytes (32 bits). Therefore, numbers are from the range $-2'147'483'648 \sim +2'147'483'647$. The USM language interpreter recognises a number while looking at the first character. It can be either a sign { + | – } or a decimal digit { 0 | 1 | ... | 9 }. Next digits can be taken from a bigger set of characters { 0 | ... | 9 | A | ... | F | H }, i.e. hexadecimal digits and character H.

You may specify hexadecimal numbers. There is a very simple rule of making decision

about the base of numbers. If a number contains at least one of the letters **A ~ F** or the last character is **H**, the number will be regarded as hexadecimal. For example:

0D represents hexadecimal D, i.e. decimal 13;

+FF represents hexadecimal FF, i.e. decimal 255;

100H represents hexadecimal 100, i.e. decimal 256

Otherwise, the number will be classified as decimal, e.g.

1234 -9930 +3000

STRINGS

2.1.3

String is a sequence of characters (1 byte) from the range of 1 ~ 255. Character **0** is regarded as strings terminator.

In the USM language strings are placed in between two quote signs. The second quote may be omitted, and all characters to the end of line included. For example:

"This is a sample of string."

The backslash character plays the role of "escape" sign. Immediately after the backslash you may place either another backslash sign or two hexadecimal digits in uppercase mode { **0** | ... | **9** | **A** | ... | **F** }. In the first case, two backslash characters will be converted to one. In the second case, the backslash and the both digits will be replaced by one character defined by hexadecimal number.

"This is a string" "and this (to the end of line)

"You may \22quote\22 inside"

"Or \09place \09non printable characters"

"Do not create too long strings, use files.\0D"

"An interpreter may have a limit\0D"

"but not smaller than 128."

The USM language makes distinction between numbers and strings. There are two operators converting strings to numbers (.#) and numbers to strings (.\$). Very often the conversion is performed internally, for example prior to an arithmetical operation.

Relational operators give different results on numbers and strings, numbers are compared by values, strings by character codes. For example:

123 > 32

if you compare values of the numbers, but

"123" < "32"

if you make alphabetical order of the strings.

VARIABLES

2.1.4

Variables keep both numerical and textual values, to allow easy access and repeatable execution. A variable name consists of letters (uppercase and lowercase), digits and underscore sign. The USM language is case sensitive, **n** and **N** refer to two different variables. A variable name should not begin with a digit – that is the difference between numbers and variables.

You must not declare a variable, just assign a value, either numerical or textual. The variable may switch from numbers to strings (and vice versa) during execution. If you forget to assign a value to a variable and you try to refer to it, the variable will be initiated with numerical value **0**.

Each of the variables is global. You should remember about it while calling a function. In the same time, there is no strict limit on the numbers of variables. You may use abbreviated name of the function as prefix of local variables.

Here are samples of proper names:

```

i      Func_i iFunc
FirstName  first_name
x1      x_1   x1C
str1   strA  sg   Astr

```

Variables are used as input elements of operators, procedures and functions, together with constants. To decide where to store resulting elements of calculation, you must list

variables only, for example:

```
1 5 .mod x
```

means calculate **1** modulo **5** and save result in **x**. In the sample, **1** and **5** are input elements of **.mod**, **x** is a resulting element of the operation **.mod**.

RECORDS

2.1.5

The USM language offers convenient access to data files. Provided a file has the proper structure, i.e. it is divided for records, and each record consists of fields separated by vertical bar sign, you can easily:

- (1) read and write a record;
- (2) count records;
- (3) change number of records;
- (4) delete a data file;
- (5) create a new data file.

To select a record, you simply say *file_name%record_number*, for example:

```
TENANT%1  
TENANT%n
```

If placed before a method name, the selected record will be imported from the data file.
If placed after a method name, the selected record will be exported to the data file.

The name of the file, followed by the percent sign, but without record number returns the count of records. You may also decide how many records must be in the file, and shorten or lengthen the file. If you decide the file must shorten to 0 records, the file will be erased.

```
TENANT% .^ nn  
0 .^ TENANT%
```

You should use the same naming convention as with variables, while giving names to data files. The names may contain letters, digits and underscore sign. It is recommended to use uppercase characters only for data file names.

FIELDS

2.1.6

As mentioned in the previous point, the USM language offers access to records of a data file. The file must have proper record structure. The command

```
file_name%record_number .^ string_name
```

will import one record and save in the specified variable. The record is divided for fields separated by vertical bar sign. In the similar way to accessing records, you can easily:

- (1) import and export a field;
- (2) count fields;
- (3) change number of fields.

To select a field you must give the following command:

```
string_name|field_number
```

for example:

```
rcd|3  
rcd|j
```

If placed before a method name, the field will be imported from the record. If placed after a method name, the field will be exported to the record.

The name of the record, followed by vertical bar sign, but without a field number returns the count of fields. You may also decide how many records must be in the record.

```
rcd| .^ jj  
5 .^ rcd|
```

METHODS**2.1.7**

There are three types of methods: operators, interpreter specific procedures, and user defined functions. Operators and procedures are hard-coded as a part of an interpreter. You must know how to call them. Functions are defined by a user and called in the same way as pre-defined methods. While defining functions, you must know the naming convention.

To call a method, you must place the period character, and write the name of the method directly after the period.

```
.method_name
```

The method name may contain almost any printable character, excluding space, period, backslash, single quote and double quote characters only. Here are samples of proper names of operators, procedures and functions.

```
.^      .and    .<     .+     .$len  .count  
.AnyKey .Menu   .Print .Transact  
.My1stFunction
```

SEMANTICS

2.2

While syntax describes the way in which constants, variables, methods, and other names, phrases and whole sentences are arranged – semantics explain the contextual meaning of sentences. The USM interpreter begins execution from the top of the programme source code, and reads sentences sequentially. That order will be modified by control operators, which cause conditional, repeated, and procedural execution. There are also commands causing premature return from a procedure or exit from a loop.

The interpreter may experience problems during execution. If an error condition is detected, the interpretation cannot be continued, and the error message will be displayed or printed. It will notify you about: error code, line number and error description. Here is the list of most important ones.

Code 1:	File Not Found
Code 2:	Stack Overflow
Code 3:	Stack Underflow
Code 4:	Wrong Indent
Code 5:	Return Error
Code 6:	Syntax of {
Code 7:	Syntax of !
Code 8:	Syntax of :
Code 9:	Syntax of #
Code 10:	Syntax of @
Code 11:	Syntax of ?
Code 12:	Syntax of .
Code 13:	Break Error
Code 14:	Method Not Found

General rules of formal syntax definitions apply to structures definitions. Additionally, a thin line shows how the execution indent changes within the structure. The term *group of commands* can be substituted by as many lines as necessary. It can be another structure inside the group of commands, and the indent will increase even more.

CONDITIONAL EXECUTION**2.2.1**

We begin our discussion about semantics of the USM language from the conditional execution structure. A group of commands will be executed during the interpretation process, if an opening condition is fulfilled. If the condition is false, the processing will omit the group of commands.

```
? condition
  group of
  commands
[?end]
```

The *conditional execution* is denoted by the question mark, the *condition* itself is built as relational expression, e.g. $n < 0$. The *group of commands* is highlighted by increased left indent of the source code. To show the end of the group, the indent returns to the initial position or even further to the left. You may use a comment line, pointing the end of condition execution structure.

You can use a *condition* inside another one, the *group of commands* closed like that will be executed, if the both *conditions* are true (conjunction of *conditions*).

```
? condition
  ? condition
    group of
    commands
  [?end]
[?end]
```

You can also apply a disjunction of *conditions*. List all of the *conditions* in one line, as shown below.

```
? condition ? condition
  group of
  commands
[?end]
```

There are six different types of condition:

- < smaller than;
- > greater than;

<= not greater than;
>= not smaller than;
= equal to;
<> not equal to.

The following sample shows the if-else kind of conditional execution.

```
? k = 0D  
  "Download OK" .TestPrint  
? k <> 0D  
  "Download Not OK" .TestPrint  
'?end
```

As you can see, there is no special structure of if-else execution, you must repeat the first condition but negated.

The position of the question mark is crucial in the execution indent model. Spaces (or tabulators) before the mark are counted and current indent redefined. The indent must be equal to the previous one. If the indent is smaller than the previous one, the USM interpreter will perform break (alias exit) of the current loop or repeat structure. If the indent is bigger than the previous one, the "Wrong Indent" error message will be displayed.

SELECT CASE**2.2.2**

The select case structure allows you to specify a *variable*, and depending on the value of the variable, to execute different groups of commands.

```
!variable
  : value
    group of
    commands
  : value
    group of
    commands
  ...
  : value
    group of
    commands
[!end]
```

If one of the cases does not specify a *value*, it will be regarded as unconditional execution.

```
:
  group of
  commands
[!end]
```

It should be placed at the end of the select case structure, because cases are checked sequentially. Placed somewhere else, it will close access to the following cases.

The USM language allows to create multi-value cases. The group of commands under such label will be executed, if the variable value is equal to one of the listed in the multi-value case.

```
: value : value ...
  group of
  commands
```

In the following sample, variable `iC3` is checked and one of the 6 cases executed. There is also one unconditional case with the “Not Supported” message.

```
! iC3
: 1
  30 0 .Transact ir
: 2
  10 0 .Transact ir
: 3
  20 0 .Transact ir
: 4
  3 .Traderef ir
: 5
  4 .Traderef ir
: 6
  91 0 .Transact ir
:
  "Not Supported" .TestPrint
'!end
```

WHILE LOOP**2.2.3**

A *group of commands* can be executed repeatedly. Prior to the next execution, a *condition* is checked. If the *condition* is true, the looping will be continued. If the *condition* is false, the interpretation will resume after the *group of commands*.

```
@ condition
  group of
  commands
['@end]
```

The *while loop* structure is denoted by the commercial at sign, the *condition* is constructed as relational expression, e.g. **n = 1**. The *group of commands* is highlighted by increased left indent of the source code. To show the end of the group, the indent returns to the initial position or even further to the left. You may use a comment line pointing the end of the while loop structure.

Only one expression may be used as the condition of the loop. Please be aware of an infinite loop. It will happen, for example, if you do not change the value of the variable checked in the main condition. There is also another way of making exit from the loop, i.e. so called conditional or unconditional breaks:

```
? condition \
\
```

You may perform multiple breaks in the case of nested structures. The backslash sign in the both conditional and unconditional breaks can be multiplied, as below:

```
? condition \\\
```

The presented below loop will be repeated until you have pressed the “Enter” key:

```
0 .^ k
@ k <> 0D
  "Press Enter Key"
  .AnyKey k
 '@end
```

REPEAT**2.2.4**

A *group of commands* is executed several times, where you may declare the number of repeats. The repeated structure can be applied in different ways – parameters of the first line denoted by the hash sign are optional, as show below.

```
# [to [from [step]]] [. variable]
|
| group of
| commands
|
| ['#end]
```

Terms *to*, *from*, *step* may be substituted by constants or variables names. Depending on how many of them are declared you can obtain the following models.

- | | | |
|------|--|---------------------------|
| (1) | infinite loop | # |
| (2) | repeat <i>to</i> times | # to |
| (3) | count <i>from to</i> | # to from |
| (4) | count <i>from to by step</i> | # to from step |
| (5) | count backward
(negative <i>step</i>) | # to from step |
| (6) | as (1) with counter
(<i>to</i> = 7FFFFFFF, <i>from</i> = 1, <i>step</i> = 1) | # . variable |
| (7) | as (2) with counter
(<i>from</i> = 1, <i>step</i> = 1) | # to . variable |
| (8) | as (3) with counter
(<i>step</i> = 1) | # to from . variable |
| (9) | as (4) with counter | # to from step . variable |
| (10) | as (5) with counter
(negative <i>step</i>) | # to from step . variable |

It can be summed up as follows: if *variable* is present, it will be populated by the current counter; the default value of the element *to* is **7FFFFFFF**; the default value of the element *from* is **1**; the default value of the element *step* is **1**; the element *step* can be negative (in this case *to* must be smaller than *from*).

The execution of the repeat structure can be described in steps.

- (1) Substitute *variable* with *from*.
- (2) (a) *step* > **0**; if *variable* > *to* then break the execution.
(b) *step* < **0**; if *variable* < *to* then break the execution.
- (3) Execute the *group of commands*.
- (4) Add *step* to *variable* and continue with step (2).

It is very important to be sure that:

- (a) *from* <= *to*, if *step* > 0;
- (b) *from* >= *to*, if *step* < 0.

Do not use *step* equal to **0**, apply the infinite loop instead, and break conditionally inside the structure.

Samples of the repeat structure.

```
# 5 . i
  i .TestPrint
'#end'

# 8 3 . n
  "0" .^ tref|n
'#end

# 4
  -1 0 ""
'#end
.Print
```

METHOD CALL**2.2.5**

Although in USM programmes functions must be firstly defined and then called, in our discussion we begin with method calling. As it was mentioned before, the name “method” is associated with general operators, interpreter specific procedures, and user defined functions. To call one of the methods, write the following statement.

```
input_values .method_name output_variables
```

The inverse Polish notation is applied to call methods. You may specify numbers, strings, variables, records and fields as *input_values*. They are separated by space, and may take several lines of your programme. After the *method_name* you must list *output_variables*, i.e. variables, records and fields (no constants) where results will be saved. You can use one programme line only.

```
"IDENTIFY TENANT"
"Name      |"
"Address  |"
.Form sn sa
"TENANT" sn 2 sa 3
"TEN" .Select2 ir
```

Procedure **.Form** displays three lines of text on screen, allows to capture answers in the second and third line, and stores them in variables **sn** and **sa**.

Procedure **.Select2** looks for matches in the **TENANT** data file by reading record by record, comparing the second field to **sn**, comparing the third field to **sa**, and saving matching records in the **TEN** data file. The number of matches will be returned by **.Select2** in variable **ir**.

The operational stack will be useful in the case of complex calculational expressions. Intermediate results can be stored on the stack for further reference. For example, to calculate $x = a^2 + b^2$ you may use the stack.

```
a a .* 'multiply a * a and leave on stack
b b .* 'multiply b * b and leave on stack
.+ x  'add numbers from the stack and store in x
```

The first and second lines of the above sample, cannot be combined,

```
a a .* b b .*
```

it would save a^2 in **b**, and the second multiplication will cause the “Stack Underflow” error. But, the second and the third lines can be combined

```
b b .* .+ x
```

The general operators are defined and described in Chapter 3. They are divided into six groups: assign and convert, binary, relational, arithmetical, strings, and stack and. Each of the USM interpreters must offer the basic set of general operators.

The interpreter specific procedures are discussed in Chapter 4, divided for five points: keyboard, screen, printer, card reader, and communication channels. The information presented in that chapter should be regarded as guidelines.

FUNCTION DEFINITION**2.2.6**

We know how to call functions, now we must learn how to define them. First of all, definitions must be located at the beginning of a programme, otherwise they will not be listed in the dictionary of available functions, and the “Function Not Found” error message will be displayed. Secondly, the definition structure consists of the opening statement, the body of the definition, and the closing statement.

```
{ [input_variables] .method_name  
  body  
  of the  
  definition  
} [output_variables]  
[{end}]
```

The opening statement is denoted by the left brace sign, followed by the *input_variables* list with space as the separator. The list of variables can be empty. Finally, the *method_name* is declared.

The closing statement is denoted by the right brace sign, followed by the *output_values* list with space as the separator. The list of values can be empty.

If the method is called, *input_values* of the calling statement will be assigned to the *input_variables* of the definition opening statement. The USM interpreter will continue execution from the first line of the *definition body*. Then, the *output_values* of the definition closing statement will be assigned to the *output_variables* of the calling statement, and the execution will return to the line following the calling statement.

The *body of the definition* may contain conditional or unconditional returns.

```
? condition } [output_values]  
} [output_values]
```

The second one may be used only inside a structure, otherwise it will be treated as the definition closing statement.

The below defined function *.Find* will look for a record of *file*, which contains *phrase*. The function will return the record and position of *phrase* in the record. If no record has been found, the function will return an empty string and **0**.

```
{ file phrase .Find
  file% .^ n
  n 1 .- n
  # n . i
  file%i .^ rcd
  rcd phrase .$pos j
  ? j > 0 } rcd j
} "" 0
'{}end
```

STACK AND INDENT

2.3

The operational stack and the execution indent are two major characteristics of the USM language, and both have to be very well explained.

Some programming languages (e.g. FORTH, PostScript) use stacks to pass values to procedures and receive results back from the procedures. Other programming languages (e.g. C, Basic) use variables to communicate with procedures. The USM language offers the both option. You may say

```
2005
```

and number **2005** will be placed on the stack. But, you may say

```
2005 .^ y
```

and number **2005** will be assigned to the **y** variable.
Values can be taken from the stack

```
.^ y
```

i.e. the topmost value from the stack will be assigned to the **y** variable.

While calling procedures, input values are delivered partially via stack, partially via operands. Please study the following sample

```
" MENU TITLE"  
"[1] Option 1"  
"[2] Option 2"  
iC .Menu iC
```

Three lines of text have been placed on the stack, variable **iC** as the input value shows the initial selection, the same variable as the output element returns a result of the **.Menu** procedure. You may place any number of lines on the stack – the procedure will count them. Therefore, procedures may have variable number of input elements as well as variable number of output elements.

There are four stack operators.

- | | |
|-----------------------|-----------------|
| (1) Push to stack | value |
| (2) Pop from stack | .^ variable |
| (3) Count stack items | .count variable |
| (4) Clear stack | .clear |

Operations performed on the stack can cause errors: “Stack Overflow”, “Stack Underflow”, and not diagnosed one – wrong objects on the stack. The last of the above-mentioned errors will be noticed during the procedure execution. It is recommended to clear the stock from time to time.

* * *

The concept of the execution indent makes USM programmes compact and readable. Any programming structure increases the indent by one, at the end of the structure, the indent must be decreased back to the initial position.

You can use either one space to increase the indent, or alternatively – the tabulator mark. The best result is obtained with the tabulator equal to the size of two spaces. If you increase the indent by 2, the error message “Wrong Indent” will be displayed. If you decrease the indent by 2, the interpreter will end two structures (sometimes it has to be like that).

Structures do not require a closing bracket, excluding the function definition, but in that case returning value must be declared. A decreased indent will be interpreted as the end of a structure. For readability purpose only, you may use a comment line indicating which one of the all structures ends.

The idea of descriptive indent is applied by many programmers to improve readability, but not by programming languages. In some cases (e.g. Real Basic) a specialised language editor imposes indent, although it is not required by the language.

To show exactly how the execution indent changes, a thin line will be drawn in the definitions of the structures. You may also study the programme sample presented in Chapter 5.

EXECUTION MODEL**2.4**

The previous points of the book explained main aspects of the USM programmes execution. Are we able to build a USM interpreter? If not, let us learn from the below placed description of the USM interpreter, written in... the USM language.

```
""" .^ FuncDict
"\1A" 64 .def$ State
""" .^ VarName
""" .^ VarValue
""" .^ Stack
U_Pgm .$len lenPgm
1 .^ ptrPgm
0 .^ Lev0
64 .^ Lev1
@ ptrPgm <= lenPgm
  ptrPgm .load_line ptrPgm su
  su .$len ju
  su .indent Lev iu lu
  ? Lev <= Lev1
    64 .^ Lev1
    @ iu <= ju
      ? Lev > Lev0 }
      @ Lev < Lev0
        .make_break
        Lev0 1 .- Lev0
      '@end
    su iu 1 .get$ ku
    ! ku
    : "?"
      su iu .read_value iu sc1
      su iu .read_relation iu sc2
      su iu .read_value iu sc3
      sc1 sc2 sc3 .condition m
      ? m = 0
        Lev .^ Lev1
    : "!"
      su iu .read_variable iu sv
```

```

sv .convert_constant sv
Lev0 1 .+ Lev0
Lev0 .^ Lev
@ Lev >= Lev0
    ptrPgm .load_line ptrPgm su
    su .$len ju
    su .indent Lev iu lu
    ? Lev < Lev0 \
    ? Lev = Lev0
        su iu .read_value iu sg
        ? sg = "" \
        ? sg = sv \
    '@end
    "!" .save_state
: ":"
    .make_break
: "@@"
    su iu .read_value iu sc1
    su iu .read_relation iu sc2
    su iu .read_value iu sc3
    sc1 sc2 sc3 .condition m
    ? m <> 0
        "@@" sc1 sc2 sc3 .save_state
    ? m = 0
        Lev .^ Lev1
: "#"
    su iu .read_value iu sr1
    su iu .read_value iu sr2
    su iu .read_value iu sr3
    iu 1 .+ iu
    su iu .read_variable iu sr0
    sr1 .convert_int r1
    sr2 .convert_int r2
    sr3 .convert_int r3
    r2 .variable_def sr0
    sr0 .convert_int r0
    .false .^ m
    ? r3 > 0
        ? r0 <= r1
            .true .^ m

```

```

? r3 > 0
  ? r0 <= r1
    .true .^ m
? m <> 0
  "#" r1 r2 r3 sr0 .save_state
? m = 0
  Lev .^ Lev1
: "{"
su iu .read_list iu sa
su iu .read_procedure iu sp
sp sa .save_function
Lev .^ Lev1
: "}"
su iu .read_list iu sv
sv .make_return
: "."
su iu .read_procedure iu sp
su iu read_list iu sv
sp .is_operator n
? n > 0
  sp .operator
? n = 0
  sp .is_procedure n
? n > 0
  sp .procedure
? n = 0
  sp .is_function n
? n > 0
  sv save_state
  sp .function
? n = 0 }
'?end
'?end
: "\22"
su iu .read_string iu sg
sg .push_stack
:
su iu .read_value iu sg
sg .convert_constant sg
sg .push_stack

```

```
    '!end
    iu lu .- j
    su lu j .del$ su
    ju j .- ju
    lu .^ iu
  '@end
'?end
'@end
```

3. OPERATORS

ASSIGN AND CONVERT

3.1

ASSIGN VALUE TO VARIABLE

3.1.1

`value .^ variable`

Operator `.^` assigns *value* to *variable*. *Variable* can be either present or not in the list of the currently used variables. In the first case *value* will replace the previous one. In the second case a new item will be created and assigned with the specified *value*.

The element *value* can be a name of another variable or a constant, i.e. number or string. The type of *variable* (numerical or textual) will be remembered.

EXAMPLE

```
1 .^ n
"Hello" .^ sg
sg .^ tg
```

NOTE

- You cannot specify a constant on the right side of the operator. Only variable names are allowed there.
- If you have not specified *variable* the *value* will stay on the stack.

READ FIELD FROM RECORD**3.1.2**

`record|number .^ variable`

Provided *record* is divided for fields, separated by the vertical bar sign, each field of *record*, can be accessed by field *number* and copied into *variable*. The field number must be not smaller than **1**, and not bigger than the count of all fields.

The value copied from the *record* will be saved in textual format, i.e. *variable* will become a string. It can be converted into numerical format by the `.#` operator.

EXAMPLE

```
rcd|2 .^ sn  
rcd|i .^ sa
```

NOTE

- There is space neither before nor after the vertical bar sign.
- If *number* has not been used yet, it will carry value **0** and the phrase `record|number` will be equivalent to `record|` (count fields in *record*).

WRITE FIELD TO RECORD**3.1.3**

value **.^** record|number

Provided *record* is divided for fields, separated by the vertical bar sign, each field of *record* can be accessed by field *number* and modified. The **.^** operator will overwrite the old value of the field. The field *number* must be not smaller than **1**, and not bigger than the count of all fields.

Value should be presented in textual format, otherwise it will be converted to the textual format, as by the **.\$** operator.

Information stored in *record* is kept in the variable length format. If a new *value* of a field has different length than the old value, *record* will be reformatted.

EXAMPLE

```
"John Smith" .^ rcd|2  
2 .^ rcd|n
```

NOTE

- Variable *record* must be initiated. It can be loaded from the file or defined by the **.def\$** operator or the number of fields changed by the **.^ record|** method.

COUNT FIELDS**3.1.4**

`record| .^ variable`

The fields of *record* are counted and the result assigned to *variable*. If *record* consists of **n** vertical bars, the result of counting will be equal to **n+1**, as you may have one field after the last bar. An empty record has one field, as explained above.

Therefore the result of counting will be always bigger than **0**. The `.^` operator will change the type of *variable* to numerical.

EXAMPLE

```
rcd| .^ n
```

NOTE

- It is important to know the number of fields in a record, as the fields are later accessed by a number from **1** to the count of fields.

CHANGE NUMBER OF FIELDS**3.1.5**

`value .^ record|`

The number of fields in *record* will be changed to the specified *value*. If *value* is bigger than the current number of fields, a proper number of vertical bars will be added at the end of *record*. If *value* is smaller than the current number of fields, the exceeding fields will be truncated.

EXAMPLE

```
12 .^ rcd|
```

NOTE

- You can get the same result by applying the `.def$` operator.

READ RECORD FROM FILE**3.1.6**

file%number .^ record

Provided *file* is divided for records separated by the vertical bar sign, each record can be accessed by record *number* and copied into the string variable *record*. *Number* must be not smaller than **1**, and not bigger than the count of records. If *number* is pointing non existing record or *file* does not exist, the operator will return an empty string in *record*.

EXAMPLE

```
TENANT%3 .^ rcd
```

NOTE

- Please make a difference between file names and variable names.
You may use uppercase characters only while specifying file names.

WRITE RECORD TO FILE**3.1.7**

record .^ file%number

Provided *file* is divided for records separated by the carriage return character, each record of *file* can be accessed by record *number* and modified. The .^ operator will overwrite the old value of the record. The record *number* must be not smaller than **1**, and not bigger than the count of all records.

The *record* should be presented in textual format, otherwise it will be converted to the textual format, as by the .\$ operator.

EXAMPLE

```
rcd .^ TENANT%3
```

NOTE

- Please make a difference between file names and variable names.
You may use uppercase characters only while specifying file names.

COUNT RECORDS**3.1.8**

`file% .^ variable`

The records of *file* are counted and the result assigned to *variable*. If *file* consists of **n** lines (carriage return characters), the result of counting will be equal to **n+1**, as you may have one record after the last carriage return. An empty file has one record, as explained above.

The result of counting will be **0** if *file* does not exist – you may use that fact to check the presence of *file*. The `.^` operator will change the type of the *variable* to numerical.

EXAMPLE

```
TENANT% .^ nr
```

NOTE

- It is important to know the number of records in a file, as the records are later accessed by a number from **1** to the count of records.

CHANGE NUMBER OF RECORDS**3.1.9**

`value .^ file%`

The number of records in *file* will be changed to the specified *value*. If *value* is bigger than the current number of records, a proper number of carriage return characters will be added at the end of *file*. If *value* is smaller than the current number of records, the exceeding records will be removed.

EXAMPLE

```
5 .^ TENANT%
```

NOTE

- If the number of records has been increased, the new records must be initiated, i.e. assigned with a default value.

DELETE FILE**3.1.10****0** .^ file%

The operator will delete *file* and all information stored there. Any operation on data files, creates backup files. If *file* has been deleted accidentally, you will have an option to recover it.

EXAMPLE

0 .^ **TENANT%**

NOTE

- In your programme you should perhaps ask a user for a confirmation before deleting *file*.

CREATE FILE**3.1.11****1** .^ file%

The operator .^ will create a new file, providing *file* does not exist. *File* will be created and saved empty. It is the starting point of the data file building. In the second step you must change the number of records, and write the initial format of records.

EXAMPLE

1 .^ **TENANT%**
5 .^ **TENANT%**
rcd .^ **TENANT%1**

NOTE

- To check if *file* exists, ask about the number of records.

CONVERT TO NUMBER**3.1.12**value **.#** variable

Value will be converted to the numerical format. *Variable* will become a number. The operator calculates the value of the initial digits at the beginning of the operand.

EXAMPLE

```
sg .# ng  
"-1234" .# n  
x .# x
```

NOTE

- You may use the same variable name at the both sides of the operator.

CONVERT TO STRING**3.1.13**variable **.\$** variable

Value will be converted to the textual format. *Variable* will become a string. The number will be converted into the decimal format with the sign at the front.

EXAMPLE

```
ng .$ sg  
123 .$ str  
x .$ x
```

NOTE

- You may use the same variable name at the both sides of the operator.

BINARY OPERATORS**3.2****BITWISE CONJUNCTION****3.2.1**

$$nf \ ng \ .and \ nh$$

The both operands nf , ng should be numbers. If not, they will be converted to the numerical format. Then, the logical operation of conjunction is performed on each couple of corresponding bits of the binary representation of the numbers. The result of the operation is saved in nh in the numerical format.

The following table shows, how the logical conjunction is calculated.

p	q	p & q
---	---	---
0	0	0
0	1	0
1	0	0
1	1	1

For example $nf = 37$, $ng = 25$. Convert numbers to binary representation: $nf = 100101$, $ng = 11001$. Write the binary numbers with the right alignment, and calculate in accordance to the table.

```

100101
011001
-----
000001

```

EXAMPLE

```
37 25 .and z
```

NOTE

- The bitwise conjunction operation can be utilised in construction of complex conditions, and building conditional execution structures.

BITWISE DISJUNCTION**3.2.2**

$$nf \ ng \ .or \ nh$$

The both operands nf , ng should be numbers. If not, they will be converted to the numerical format. Then, the logical operation of disjunction is performed on each couple of corresponding bits of the binary representation of the numbers. The result of the operation is saved in nh in the numerical format.

The following table shows, how the logical disjunction is calculated.

p	q	p q
---	---	-----
0	0	0
0	1	1
1	0	1
1	1	1

For example $nf = 37$, $ng = 25$. Convert numbers to binary representation: $nf = 100101$, $ng = 11001$. Write the binary numbers with the right alignment, and calculate in accordance to the table.

$$\begin{array}{r}
 100101 \\
 011001 \\
 \hline
 111101
 \end{array}$$

EXAMPLE

$$37 \ 25 \ .or \ z$$

NOTE

- ♦ The bitwise disjunction operation can be utilised in construction of complex conditions, and building conditional execution structures.

BITWISE EXCLUSIVE-OR**3.2.3**

nf ng .xor nh

The both operands *nf*, *ng* should be numbers. If not, they will be converted to the numerical format. Then, the logical operation of exclusive-or is performed on each couple of corresponding bits of the binary representation of the numbers. The result of the operation is saved in *nh* in the numerical format.

The following table shows, how the logical exclusive-or is calculated.

<i>p</i>	<i>q</i>	<i>p ^ q</i>
0	0	0
0	1	1
1	0	1
1	1	0

For example *nf = 37*, *ng = 25*. Convert numbers to binary representation: *nf = 100101*, *ng = 11001*. Write the binary numbers with the right alignment, and calculate in accordance to the table.

```

100101
011001
-----
111100

```

EXAMPLE

```
37 25 .xor z
```

NOTE

- The bitwise exclusive-or operation can be utilised in construction of complex conditions, and building conditional execution structures.

BITWISE NEGATION**3.2.4**

`ng .not nh`

Operand *ng* should be a number. If not, it will be converted to the numerical format. Then, the logical operation of negation is performed on each bit of the binary representation of the number. The result of the operation is saved in *nh* in the numerical format.

The following table shows, how the logical negation is calculated.

<u>p</u>	<u>~p</u>
0	1
1	0

For example `ng = 25`. Convert the number to binary representation: `ng = 11001`. Calculate in accordance to the table with each bit of the number.

```
011001
-----
100110
```

EXAMPLE

```
25 .not z
```

NOTE

- The bitwise negation operation can be utilised in construction of complex conditions, and building conditional execution structures.

VALUE TRUE**3.2.5**

.true variable

Operator **.true** assigns value **-1** to *variable*. Number **-1** has binary representation **1**₁₆ (16 digits **1**).

EXAMPLE

```
.true x  
@ x
```

NOTE

- Bitwise negation of **-1** equals **0**.

VALUE FALSE**3.2.6**

.false variable

Operator **.false** assigns value **0** to *variable*. Number **0** has binary representation **0**₁₆ (16 digits **0**).

EXAMPLE

```
? k = 0D  
.false x
```

NOTE

- Bitwise negation of **0** equals **-1**.

RELATIONAL OPERATORS

3.3

RELATION SMALLER THAN

3.3.1

 $xf\ xg\ .<\ nh$

Operator `.<` (smaller than) compares two numbers or two strings xf and xg . A result of the comparison, either true denoted by `-1` or false denoted by `0`, will be assigned to the integer variable nh specified on the right side of the operator.

In case of numerical operands, the operator compares their numerical values. In case of textual operands, the operator compares their lexical values, i.e. positions in the alphabetical order.

Relational operator `.<` will give the result true, if the first operand is smaller than the second operand. Otherwise, the result will equal false.

EXAMPLE

```
m 5 .< c1
n 4 .< c2
c1 c2 .and cnd
? cnd \
```

NOTE

- If operands are of different types, both will be converted to strings, and the lexical comparison will be performed.

RELATION GREATER THAN**3.3.2** $xf\ xg\ .>\ nh$

Operator `.>` (greater than) compares two numbers or two strings xf and xg . A result of the comparison, either true denoted by `-1` or false denoted by `0`, will be assigned to the integer variable nh specified on the right side of the operator.

In case of numerical operands, the operator compares their numerical values. In case of textual operands, the operator compares their lexical values, i.e. positions in the alphabetical order.

Relational operator `.>` will give the result true, if the first operand is greater than the second operand. Otherwise, the result will equal false.

EXAMPLE

```
m 5 .> c1
n 4 .> c2
c1 c2 .and cnd
? cnd \
```

NOTE

- If operands are of different types, both will be converted to strings, and the lexical comparison will be performed.

RELATION NOT GREATER THAN**3.3.3** $xf \ xg \ .<= \ nh$

Operator `.<=` (not greater than) compares two numbers or two strings xf and xg . A result of the comparison, either true denoted by `-1` or false denoted by `0`, will be assigned to the integer variable nh specified on the right side of the operator.

In case of numerical operands, the operator compares their numerical values. In case of textual operands, the operator compares their lexical values, i.e. positions in the alphabetical order.

Relational operator `.<=` will give the result true, if the first operand is smaller than or equal to the second operand.

EXAMPLE

```
m 5 .<= c1
n 4 .<= c2
c1 c2 .and cnd
? cnd \
```

NOTE

- If operands are of different types, both will be converted to strings, and the lexical comparison will be performed.

RELATION NOT SMALLER THAN**3.3.4** $xf\ xg\ .>= \ nh$

Operator `.<=` (not smaller than) compares two numbers or two strings xf and xg . A result of the comparison, either true denoted by `-1` or false denoted by `0`, will be assigned to the integer variable nh specified on the right side of the operator.

In case of numerical operands, the operator compares their numerical values. In case of textual operands, the operator compares their lexical values, i.e. positions in the alphabetical order.

Relational operator `.>=` will give the result true, if the first operand is greater than or equal to the second operand.

EXAMPLE

```
m 5 .>= c1
n 4 .>= c2
c1 c2 .or cnd
? cnd \
```

NOTE

- If operands are of different types, both will be converted to strings, and the lexical comparison will be performed.

RELATION EQUAL TO**3.3.5**
$$xf \ xg \ . = \ nh$$

Operator `. =` (equal to) compares two numbers or two strings xf and xg . A result of the comparison, either true denoted by `-1` or false denoted by `0`, will be assigned to the integer variable nh specified on the right side of the operator.

In case of numerical operands, the operator compares their numerical values. In case of textual operands, the operator compares their lexical values, i.e. positions in the alphabetical order.

Relational operator `. =` will give the result true, if the first operand is equal to the second operand.

EXAMPLE

```
m 5 . = c1
n 4 . = c2
c1 c2 .or cnd
? cnd \
```

NOTE

- If operands are of different types, both will be converted to strings, and the lexical comparison will be performed.

RELATION NOT EQUAL TO**3.3.5** $xf \ xg \ .<> \ nh$

Operator `.<>` (not equal to) compares two numbers or two strings xf and xg . A result of the comparison, either true denoted by `-1` or false denoted by `0`, will be assigned to the integer variable nh specified on the right side of the operator.

In case of numerical operands, the operator compares their numerical values. In case of textual operands, the operator compares their lexical values, i.e. positions in the alphabetical order.

Relational operator `.<>` will give the result true, if the first operand is not equal to the second operand.

EXAMPLE

```
m 5 .<> c1
n 4 .<> c2
c1 c2 .or cnd
? cnd \
```

NOTE

- If operands are of different types, both will be converted to strings, and the lexical comparison will be performed.

ARITHMETICAL OPERATORS

3.4

ADD NUMBERS

3.4.1

 $nf\ ng\ .+\ nh$

The operands nf and ng of the $.+$ operator should be numbers, if not, they will be converted to the numerical format, in the same way as by the $.\#$ operator. Then, the standard operation of adding numbers will be done. The result of the operation will be assigned to the numerical variable nh , specified on the right side of the operator.

If there is no variable after the operator, the result of the operation will be pushed on the operational stack, for the usage as an operand of the next operator. It allows you to combine several operations into one-line expression.

 $a\ b\ c\ d\ .+\ .+\ .+\ e$

will result in $e = a + (b + (c + d)) = a + b + c + d$.

EXAMPLE

```
5 .^ m
# 3
m 4 .+ m
'#end
```

NOTE

- Could you calculate on paper, what is the value of variable **m** after running the above-listed USM programme?

SUBTRACT NUMBERS**3.4.2**

`nf ng .- nh`

The operands *nf* and *ng* of the `.-` operator should be numbers, if not, they will be converted to the numerical format, in the same way as by the `.#` operator. Then, the standard operation of subtracting numbers will be done. The result of the operation will be assigned to the numerical variable *nh*, specified on the right side of the operator.

If there is no variable after the operator, the result of the operation will be pushed on the operational stack, for the usage as an operand of the next operator. It allows you to combine several operations into one-line expression.

```
a b c d .- .- .- e
```

will result in $e = a - (b - (c - d)) = a - b + c - d$.

EXAMPLE

```
25 .^ m
# 5 . i
m i .- m
'#end
```

NOTE

- Could you calculate on paper, what is the value of variable **m** after running the above-listed USM programme?

MULTIPLY NUMBERS**3.4.3**

nf ng . nh*

The operands *nf* and *ng* of the *.** operator should be numbers, if not, they will be converted to the numerical format, in the same way as by the *.#* operator. Then, the standard operation of multiplying numbers will be done. The result of the operation will be assigned to the numerical variable *nh*, specified on the right side of the operator.

If there is no variable after the operator, the result of the operation will be pushed on the operational stack, for the usage as an operand of the next operator. It allows you to combine several operations into one-line expression.

```
a b .*  
c d .*  
.+ e
```

will result in $e = (a * b) + (c * d)$.

EXAMPLE

```
5 .^ m  
# 2  
m m .* m  
'#end
```

NOTE

- Could you calculate on paper, what is the value of variable **m** after running the above-listed USM programme?

DIVIDE NUMBERS**3.4.4**

nf ng ./ nh

The operands *nf* and *ng* of the *./* operator should be numbers, if not, they will be converted to the numerical format, in the same way as by the *.#* operator. Moreover, the second number must be not equal to **0**. Then, the operation of dividing numbers and truncating to integer will be done. The result of the operation will be assigned to the numerical variable *nh*, specified on the right side of the operator.

If there is no variable after the operator, the result of the operation will be pushed on the operational stack, for the usage as an operand of the next operator. It allows you to combine several operations into one-line expression.

```
a b ./  
c d ./  
.- e
```

will result in $e = (a / b) - (c / d)$.

EXAMPLE

```
17 .^ m  
# 2  
m 3 ./ m  
'#end
```

NOTE

- Could you calculate on paper, what is the value of variable **m** after running the above-listed USM programme?

CALCULATE REMINDER**3.4.5**

nf ng **.mod** *nh*

The operands *nf* and *ng* of the **.mod** operator should be numbers, if not, they will be converted to the numerical format, in the same way as by the **.#** operator. Moreover, the second number must be not equal to **0**. Then, the operation of dividing numbers and calculating a remainder will be done. The result of the operation will be assigned to the numerical variable *nh*, specified on the right side of the operator.

If there is no variable after the operator, the result of the operation will be pushed on the operational stack, for the usage as an operand of the next operator. It allows you to combine several operations into one-line expression.

```
a b .mod
c d .mod
.+ e
```

will result in $e = (a \bmod b) + (c \bmod d)$.

EXAMPLE

```
678 .^ m
"" .^ sm
# 3
m 10 .mod d
d sm .& sm 'operation of linking strings
m 10 ./ m
'#end
```

NOTE

- Could you calculate on paper, what is the value of variable **sm** after running the above-listed USM programme?

STRING OPERATORS

3.5

LINK STRINGS**3.5.1**`sf sg .& sh`

The operands *sf* and *sg* of the `.&` operator should be strings, if not, they will be converted to textual format, in the same way as by the `.$` operator. Then, the strings will be linked, i.e. string *sf* will be directly followed by string *sg*, to create a new string *sh*.

Several parts (e.g. words) can be combined into one string (e.g. sentence) through the operational stack. All parts are placed on the stack, the stack is counted and the operator of linking strings will be repeated as follows.

EXAMPLE

```
"Hello " "Africa "  
"tell " "me " "how "  
"you " "doing."  
.count n  
n 1 .-  
# n  
. &  
. ^ sh
```

NOTE

- In the above-presented sample separating spaces are included into parts. Otherwise, the resulting string will have no spaces between words.

INITIATE STRING**3.5.2**`sf ng .def$ sh`

A new string *sh* will be defined by repeating phrase *sf ng* times. Phrase *sf* may have one character or more characters. You could obtain the same result by applying the `.&` operator.

EXAMPLE

```
" |" 15 .def$ rcd
```

NOTE

- The `.def$` operator must be utilised to create an empty record with proper number of fields.

STRING LENGTH**3.5.3**`sg .$len nh`

If operand *sg* is not a string, it will be converted to the textual format. Then, characters of *sg* will be counted and result copied to variable *nh*.

EXAMPLE

```
sn .$len n  
? n > 0  
  "Not empty" .TestPrint  
'?end
```

NOTE

- The `.def$` operator must be utilised to create an empty record with proper number of fields.

POSITION IN STRING**3.5.4**

sf sg **.\$pos** *nh*

Operator **.\$pos** searches for appearance of string *sf* in string *sg*. If the operation is successful, variable *nh* will show position of *sf* in *sg*, where characters are counted starting from **1**. If the operation is not successful, variable *nh* will be set to **0**.

The operation of searching is case sensitive.

If the first operand is empty, the result will be equal to **1**, even when the second operand is empty as well.

Operand *sf* may be longer than **1**, position of *sf* in *sg* shows character number of the first matching character in *sg*.

EXAMPLE

```
"Hello Africa" .^ sa
" " sa .$pos n
? n = 0
"One word" .TestDisplay
? n > 0
"Many words" .TestDisplay
'?end
```

NOTE

- ✦ The first operand should be shorter than the second one.

DELETE SUBSTRING**3.5.5**`sf nf ng .del$ sh`

A part of string *sf* will be removed, and the resulting string copied to variable *sh*. The part to be removed is defined by starting position *nf* and the length of the substring (value *ng*).

EXAMPLE

```
str 4 3 .del$ str
```

NOTE

- The **.del\$** operator does not return the substring, which was deleted. You must firstly get the substring by applying the **.get\$** operator.

INSERT SUBSTRING**3.5.6**`sf nf ng .ins$ sh`

Spaces will be inserted to string *sf*, and the resulting string copied to variable *sh*. The position, where spaces must be inserted is defined by *nf*, and the number of the inserted spaces by *ng*.

EXAMPLE

```
str 4 3 .ins$
```

NOTE

- The **.ins\$** operator does not define the inserted substring (it will be filled by spaces). You must then put a new substring by applying the **.put\$** operator.

GET SUBSTRING**3.5.7**

sf *nf* *ng* **.get\$** *sh*

A part of string *sf* will be extracted, and the substring copied to variable *sh*. The part to be extracted is defined by starting position *nf* and the length of the substring (value *ng*).

EXAMPLE

```
str 4 3 .get$ sg
```

NOTE

- The **.get\$** operator does not delete the substring, which was taken. If necessary, you must then delete the substring by applying the **.del\$** operator.

PUT SUBSTRING**3.5.8**

sf *nf* *sg* **.put\$** *sh*

String *sg* will be inserted to string *sf*, and the resulting string copied to *sh*. The position, where the substring will be inserted is defined by *nf*.

EXAMPLE

```
" extra " .^ sg  
str 4 sg .put$ str
```

NOTE

- The **.put\$** operator does not create space for the substring. If necessary, you must insert spaces by applying the **.ins\$** operator.

STACK OPERATORS**3.6****PUSH STACK****3.6.1**

value

To place *value* on the stack, list it in a line without any operator. You may list more than one value in the line, and you can use more than one line.

Term *value* can be substituted by: numbers, strings, variables, records, fields. Prior to the placing value on the stack, it will be converted to constant, e.g. name of a variable will be replaced by its value. Consequently, the stack contains constants only (numerical or textual).

EXAMPLE

```
-1 20H "COLLECTION SUMMARY"  
# 4  
-1 0 ""
```

POP STACK**3.6.2**

.^ variable

To pop a constant from the stack, use operator assign but do not specify a value. The topmost element of the stack will be assigned to variable, and removed from the stack.

EXAMPLE

```
.^ str  
str .TestDisp
```

COUNT STACK**3.6.3****.count** variable

Elements of the stack will be counted and the result of counting assigned to the specified *variable*.

EXAMPLE

```
.count n  
# n  
.^ s
```

NOTE

- Operator **.count** will push the result to stack, if *variable* is not present on the right side of the operator.

CLEAR STACK**3.6.4****.clear**

The stack will be cleared, it means all elements of the stack removed definitely. Some of the interpreter specific procedures require the stack containing their operands only. Those procedures allow to use variable number of input elements.

EXAMPLE

```
.clear  
-1 20H "COLLECTION SUMMARY"
```

NOTE

- Operator **.clear** may be used either before or after procedures calls.

4. INTERPRETER SPECIFIC PROCEDURES

Mostly, they have common syntax throughout interpreters. Few of them are unique for a particular interpreter. What must be stated is: each of the interpreter specific procedures has to be written from scratch, while a new interpreter is built.

The procedures are servicing external devices of a small or big computer. They talk to the operating system, to the API (Application Platform Interface), or SDK (Software Development Kit). Depending on the language in which the interpreter for the computer is built, the interpreter specific procedures are constructed differently.

In this chapter few of the procedures are described. The chapter presents guidelines of how to create new procedures. It is beyond the scope of this book, to describe all of the procedures. In the book the USM language is defined, its syntax and semantics described, and philosophy of going forward with new methods creation – touched.

The chapter describes procedures, grouped in five points: keyboard, screen, printer, card reader, communication channels.

KEYBOARD**4.1**

lines_of_text
.AnyKey variable

Lines_of_text are displayed on screen, placed at the bottom of the screen. The number of the lines is variable, but not bigger than the size of the screen. Because elements of the stack are counted, the stack must be cleared prior to the procedure call.

Then, the procedure is waiting until a key has been pressed. The code of the key will be copied to *variable* on return from the procedure.

EXAMPLE

```
.clear  
"Press any key"  
"to continue.."  
.AnyKey key  
? key = 0C \
```

NOTE

- Different devices have different code tables. For example, the "1" number key of JadeAire terminal returns code **1**, when the same key of a PC returns **49**.

SCREEN**4.2**

value **.TestDisp**

Operand *value* will be displayed on screen to help you in debugging of your new programme. It will allow you to check your variables, records, fields, or to display messages showing the process of execution.

EXAMPLE

```
"I am here" .TestDisp  
n .TestDisp  
rcd|3 .TestDisp
```

NOTE

- Alternatively, you could use operator **.TestPrint**.

lines_of_text
.Screen

Lines_of_text are displayed on screen, starting from the top of the screen. The number of the lines is variable, but not bigger than the size of the screen. Because elements of the stack are counted, the stack must be cleared prior to the procedure call.

Procedure **.Screen** is similar to procedure **.AnyKey**, but there are two differences. Firstly, *lines_of_text* are placed in differently. Secondly, procedure **.Screen** does not wait for a key to be pressed.

EXAMPLE

```
{ descr .ErrorMessage
# 3
# 7
"""
descr .Screen
60 .Wait
# 8
"""
.Screen
40 .Wait
}
```

NOTE

- In the above-listed sample, string *descr* is displayed in the bottom line of the screen in the blinking mode. A new function **.ErrorMessage** has been created. To call the function, write the following statement.

```
"Data File Empty" .ErrorMessage
```

- Procedure **.Screen** cleans the screen, and then displays text. In the sample, you may remove two lines and have exactly the same result.

```
#8
"""
```

lines_of_text
item **.Menu** choice

Lines_of_text are displayed on screen, starting from the top of the screen. The number of the lines is variable, and it can be bigger than the size of the screen. The first line is regarded as the title of the menu. The following lines are options, and will be scrolled by the up and down keys. You will select one of the options, your choice will be returned by the **.Menu** procedure in variable *choice*.

Operand *item* decides about position of a cursor at the beginning of the selection process. The cursor can be moved down and up, gradually with the arrows keys, or quickly with the numerical keys. Eventually, the enter key should be pressed to show the choice. You could also press the cancel key, and value **-1** will be returned by the procedure.

After return from the procedure, variable *choice* will be used in the select case structure to perform right action. A menu system has usually a hierarchal construction with nested select case structures end several **.Menu** procedures.

EXAMPLE

```
1.^ iC1
@ iC1 > 0
"COLLECTIONS"
"[1] Download Info"
"[2] Select Tenant"
"[3] Upload Info"
iC1 .Menu iC1
! iC1
```

NOTE

- Procedure **.Menu** measures time of screen being in idle state, and compares to the *timeout* parameter. The procedure will press cancel by itself, if *timeout* has been passed.

lines_of_text
.Form output_variables

Lines_of_text are displayed on screen, placed at the bottom of the screen. The number of lines is variable, but not bigger than the size of the screen. They must correspond to *output_variables*, listed after the procedure name. *Lines_of_text* and *output_variables* will be aligned to the bottom, and procedure **.Form** will allow you to give answers next to *lines_of_text*, i.e. questions.

Starting from the top of the form, the old value of the next variable will be displayed, and the underscore cursor will show, where the editing process is active. The enter key will move the cursor to the next line, or finish the procedure, if pressed in the last line. You may interrupt the procedure by pressing the cancel key.

Edited values will return from the procedure in *output_variables*. Each variable will change format to textual (use **.#** to convert to numbers).

EXAMPLE

```
.clear  
"IDENTIFY TENANT"  
"Name |"  
"Address |"  
.Form sn sa
```

NOTE

- You could omit the dialog title line and leave active lines only. On the contrary, the dialog title can occupy more than one line.

PRINTER**4.3**

value **.TestPrint**

Operand *value* will be printed on printer to help you in debugging of your new programme. It will allow you to check your variables, records, fields, or to display messages showing the process of execution.

EXAMPLE

```
"Label A" .TestPrint  
str .TestPrint  
# 4  
"" .TestPrint
```

NOTE

- Alternatively, you could use operator **.TestDisp**.

{ align font string }*

.Receipt

Procedure **.Receipt** formats lines of text and sends them to printer. You may use the procedure to print: receipts, messages, reports, vouchers, etc.

Clear the stack, and write three-element statements, as many as it is required. The first element *align* describes the alignment: **-1** – left, **0** – centre, **1** – right. The second element *font* may have four values: **0** – normal, **10H** – double width, **20H** – double height, **30H** – double width and height. String can be declared as: variable, textual constant, result of linking with the **.&** operator. Please study the example below.

The procedure clears the stack after finishing and does not return any value.

EXAMPLE

```
.clear
0 30H "WIERDA PROPERTY"
0 0 sn
0 0 sa
0 0 ""
-1 20H "COLLECTION SUMMARY"
-1 0 "Date: " .Date .&
-1 0 "Time: " .Time .&
-1 0 " By: " .User .&
-1 0 "Rent: " st|4 .&
-1 0 "Paid: " st|5 .&
-1 0 ""
1 0 "Electricity on " st|7 .&
1 0 "Meter Reading " st|6 .&
1 0 "Water on " st|9 .&
1 0 "Meter Reading " st|8 .&
# 4
-1 0 ""
.Receipt
```

NOTE

- Any mistake in specifying the input values will result in wrong printout. Make sure, each line contains three elements, as described above.

CARD READER**4.4**

message **.Track12** ir trk1 trk2

Procedure **.Track12** reads information from a magnetic card. It prompts a user to swipe the card displaying *message*, passed to the procedure as the input parameter.

The procedure will return three values: status of the operation in *ir*, track 1 information in *trk1*, track 2 information in *trk2*. Variable *ir* may have the following values:

-1 – cancelled, 0 – nothing done, 1 – track 1, 2 – track 2, 3 – track 1 and 2.

Both *trk1* and *trk2* are strigs, you must measure them by **.\$len**, to know what was received during swiping.

Procedure **.Track12** is presented to show, that each step of a financial transaction is a separate procedure – easy to call and performing exactly what is needed.

EXAMPLE

```
"Swipe a card" .Track12 i sg1 sg2
? i > 0
  sg2 "=" .$pos j
  j 1 .- j
  ? j > 0
    sg2 1 j .get$ sg2
"Card number" .TestPrint
sg2 .TestPrint
```

NOTE

- Procedure **.Track12** contains built-in timeout mechanism. Return value **0** means that nothing was done.

card type **.Transact** *ir*

Procedure **.Transact** shows global approach to the transacting. Instead of doing partial steps, one complete procedure performs everything. You specify two input parameters: type of card and type of transaction.

<i>card</i>	meaning
10	Saving Account
20	Cheque Account
30	Credit Card
40	Combo Card
90	Loyalty Card
91	Cash Payment
92	Cheque Payment

<i>type</i>	meaning
0	purchase
1	cash withdrawal
9	goods and cash
20	refund
30	balance

The procedure will perform several operations, which can be shortly described as follows:

- ask to swipe a card;
- allow to make a manual entry of a card number;
- check the last four digits of the card number;
- prompt to enter an amount;
- optionally ask for a tip, a table, an invoice number, budget period, etc.
- if required, ask for PIN and perform encryption;
- connect to a host;
- build a message and send;
- wait for a response and interpret it;
- print merchant's receipt;
- allow to perform reversal;
- print client's receipt.

If the above described operation has been successfully accomplished, procedure **.Transact** will return value **0** in *ir*. If the operation fails, procedure will return an error

message (non-zero number). It can be either bank error code (positive number) or procedure error code (negative number).

EXAMPLE

```
1 .^ iC3
@ iC3 > 0
sna
"[1] Credit Card"
"[2] Debit Card (Sav)"
"[3] Debit Card (Chq)"
"[4] Personal Cheque"
"[5] Company Cheque"
"[6] Cash Payment"
iC3 .Menu iC3
! iC3
: 1
  30 0 .Transact ir
: 2
  10 0 .Transact ir
: 3
  20 0 .Transact ir
: 4
  3 .Traderef ir
  ? ir = 0
  92 0 .Transact ir
: 5
  4 .Traderef ir
  ? ir = 0
  92 0 .Transact ir
: 6
  91 0 .Transact ir
'!end
```

NOTE

- The same result can be obtained by writing your own procedure "Transaction", where each step is performed by a statement of the USM language.

COMMUNICATION CHANNELS**4.5**

filename **.DnLoadPC**

Provided your terminal is connected via RS-232 port to a PC with a special communication application, you could download files and save them in the file system of the terminal (if exists). You must specify *filename* only.

Procedure **.DnLoadPC** will send to the PC message: "Download *filename*". In response, the PC will transfer the first chunk of the file. The terminal will then send the empty message (ACK) and get the next chunk – until the chunk will be empty.

EXAMPLE

"TENANT" .DnLoadPC

filename **.UpLoadPC**

Provided your terminal is connected via RS-232 port to a PC with a special communication application, you could upload files from the file system of the terminal (if exists). You must specify *filename* only.

Procedure **.UpLoadPC** will send to the PC message: "Upload *filename*". In response, the PC will send an empty message (ACK). The terminal will then transfer chunks of the file, and an empty message after reaching the end of file.

EXAMPLE

"TENANT" .UpLoadPC

5. PROGRAMMING SAMPLE

```
{ n .Traderef
  n .^ tref|2
  ! n
  : 3
  "PERSONAL CHEQUE"
  "ID No |"
  "Chq No|"
  "Br No |"
  "Acc No|"
  "Date |"
  "Amount|"
  .Form s3 s4 s5 s6 s7 s8
  s3 .^ tref|3
  s4 .^ tref|4
  s5 .^ tref|5
  s6 .^ tref|6
  s7 .^ tref|7
  s8 .^ tref|8
  : 4
  "COMPANY CHEQUE"
  "Reg No|"
  "Chq No|"
  "Br No |"
  "Acc No|"
  "Date |"
  "Amount|"
  .Form s3 s4 s5 s6 s7 s8
  s3 .^ tref|3
  s4 .^ tref|4
  s5 .^ tref|5
  s6 .^ tref|6
  s7 .^ tref|7
  s8 .^ tref|8
  tref .TalkToTref iret
} iret
```

```

'Main Module
1 .^ iC1
@ iC1 > 0
"    COLLECTIONS"
"[1] Download Info"
"[2] Identify Tenant"
"[3] Upload Info"
iC1 .Menu iC1
! iC1
: 1
"Check RS-232"
"and Launch JadePC"
"======"
"Cancel | Enter" ""
.AnyKey k
? k = 0D
"TENANT" .DnLoadPC
"Download OK"
.AnyKey kk
? k <> 0D
"Download Cancelled"
.AnyKey kk
'?end
: 2
"" .^ sn
"" .^ sa
0 .^ ir
@ ir = 0
"IDENTIFY TENANT"
"Name   |"
"Address|"
.Form sn sa
"TENANT" sn 2 sa 3
"TEN" .Select2 ir
1 .^ iC2
"    SELECT TENANT"
"TEN" iC2 .FileMenu iC2
? iC2 > 0
TEN%iC2 .^ st
st|2 .^ sn

```

```

st|3 .^ sa
"      " sn .& sna
sna ": " .& sna
sna sa .& sna
1 .^ iC2
@ iC2 > 0
sna
"[1] Levy Payment"
"[2] Electricity Meter"
"[3] Water Meter"
"[4] Summary"
iC2 .Menu iC2
! iC2
: 1
1 .^ iC3
@ iC3 > 0
sna
"[1] Credit Card"
"[2] Debit Card (Sav)"
"[3] Debit Card (Chq)"
"[4] Personal Cheque"
"[5] Company Cheque"
"[6] Cash Payment"
iC3 .Menu iC3
! iC3
: 1
30 0 .Transact ir
: 2
10 0 .Transact ir
: 3
20 0 .Transact ir
: 4
3 .Traderef ir
: 5
4 .Traderef ir
: 6
91 0 .Transact ir
'!end
? ir = 0
.Date .^ st|5

```

```

        st|1 .^ n
        st .^ TENANT%n
    '?end
    '@
: 2
st|6 .^ sr
st|7 .^ sd
"" .^ sc
"    ELECTRICITY"
sna
"Last|" sd .&
"Read|" sr .&
"Curr|" .Form sc
? sc <> ""
    sc .^ st|6
    .Date .^ st|7
    st|1 .^ n
    st .^ TENANT%n
'?end
: 3
st|8 .^ sr
st|9 .^ sd
"" .^ sc
"    WATER"
sna
"Last|" sd .&
"Read|" sr .&
"Curr|" .Form sc
? sc <> ""
    sc .^ st|8
    .Date .^ st|9
    st|1 .^ n
    st .^ TENANT%n
'?end
: 4
0 30H "SHOPPING  CENTRE"
0 0 sn
0 0 sa
-1 0 ""
-1 20H "COLLECTION  SUMMARY"

```

```
        -1 0 "Date: " .Date .&
        -1 0 "Time: " .Time .&
        -1 0 " By: " .User .&
        -1 0 "Rent: " st|4 .&
        -1 0 "Paid: " st|5 .&
        -1 0 ""
        1 0 "Electricity on " st|7 .&
        1 0 "Meter Reading " st|6 .&
        1 0 "Water on " st|9 .&
        1 0 "Meter Reading " st|8 .&
        # 4
        -1 0 ""
        .Receipt
    '!end
  '@end
'?end
: 3
  "Check RS-232"
  "and Launch JadePC"
  "======"
  "Cancel | Enter" ""
  .AnyKey k
  ? k = 0D
    "TENANT" .UpLoadPC
    "Upload OK"
    .AnyKey kk
  ? k <> 0D
    "Upload Cancelled"
    .AnyKey kk
'?end
'!end
'@end
"Thank you"
.AnyKey k
```

INDEX

! (select case)	2.2.1	19
" (string)	2.1.3	13
# (repeat)	2.2.4	24
% (record)	2.1.5	15
' (comment)	2.1	10
. (method call)	2.2.5	26
.# (convert to number)	3.1.12	43
.\$ (convert to string)	3.1.13	43
.\$len (string length)	3.5.3	61
.\$pos (position in string)	3.5.4	62
.& (link strings)	3.5.1	39
.* (multiply numbers)	3.4.3	57
.+ (add numbers)	3.4.1	55
.- (subtract numbers)	3.4.2	56
./ (divide numbers)	3.4.4	58
.< (relation smaller than)	3.3.1	49
.<= (relation not greater than)	3.3.3	51
.<> (relation not equal to)	3.3.6	54
.= (relation equal to)	3.3.5	53
.> (relation greater than)	3.3.2	50
.>= (relation not smaller than)	3.3.4	52
.and (bitwise conjunction)	3.2.1	44
.AnyKey	4.1	68
.clear (clear stack)	3.6.4	66
.count (count stack)	3.6.3	66
.def\$ (initiate string)	3.5.2	61
.del\$ (delete substring)	3.5.5	63
.DnLoadPC	4.5	78
.false (value false)	3.2.6	48
.Form	4.2	72
.get\$ (get substring)	3.5.7	64
.ins\$ (insert substring)	3.5.6	63
.Menu	4.2	71
.mod (calculate remainder)	3.4.5	60
.not (bitwise negation)	3.2.4	47
.or (bitwise disjunction)	3.2.2	45

.put\$ (put substring)	3.5.8	64
.Receipt	4.3	74
.Screen	4.2	70
.TestDisp	4.2	69
.TestPrint	4.3	73
.Track12	4.4	75
.Transact	4.4	76
.true (value true)	3.2.5	48
.UpLoadPC	4.5	78
.xor (bitwise exclusive-or)	3.2.3	46
: (case)	2.2.2	21
< (condition)	2.2.1	19
<= (condition)	2.2.1	19
<> (condition)	2.2.1	19
= (condition)	2.2.1	19
> (condition)	2.2.1	19
>= (condition)	2.2.1	19
? (conditional execution)	2.2.1	19
@ (while loop)	2.2.3	23
Add numbers (.+)	3.4.1	55
Assign value to variable (.^)	3.1.1	36
Bitwise conjunction (.and)	3.2.1	44
Bitwise disjunction (.or)	3.2.2	45
Bitwise exclusive-or (.xor)	3.2.3	46
Bitwise negation (.not)	3.2.4	47
Break (\)	2.2.3	23
Calculate remainder (.mod)	3.4.5	59
Case (:)	2.2.2	21
Change number of fields	3.1.5	39
Change number of records	3.1.9	41
Clear stack (.clear)	3.6.4	66
Comment (')	2.1	10
Conditional execution (?)	2.2.1	19
Convert to number (.#)	3.1.12	43
Convert to string (.\$)	3.1.13	43
Count fields	3.1.4	39
Count records	3.1.8	41
Count stack (.count)	3.6.3	66

Create file	3.1.11	42
Delete file	3.1.10	42
Delete substring (.del\$)	3.5.5	63
Divide numbers (./)	3.4.4	58
Field ()	2.1.6	16
Function definition ({})	2.2.6	28
Get substring (.get\$)	3.5.7	64
Initiate string (.def\$)	3.5.2	61
Insert substring (.ins\$)	3.5.6	63
Link strings (.&)	3.5.1	60
Method call (.)	2.2.5	26
Multiply numbers (.*)	3.4.3	57
Position in string (\$pos)	3.5.4	62
Put substring (.put\$)	3.5.8	64
Read field from record	3.1.2	37
Read record from file	3.1.6	40
Record (%)	2.1.5	15
Relation equal to (.=)	3.3.5	53
Relation greater than (.>)	3.3.2	50
Relation not equal to (.<>)	3.3.6	54
Relation not greater than (.<=)	3.3.3	51
Relation not smaller than (.>=)	3.3.4	52
Relation smaller than (.<)	3.3.1	49
Repeat (#)	2.2.4	24
Return (})	2.2.6	28
Select case (!)	2.2.2	21
String (\$)	2.1.3	13
String length (.\$len)	3.5.3	61
Subtract numbers (.-)	3.4.2	56
Value false (.false)	3.2.6	48
Value true (.true)	3.2.5	48
While loop (@)	2.2.3	23
Write field to record	3.1.3	38
Write record to file	3.1.7	40
\\ (break)	2.2.3	23
{ (function definition)	2.2.6	28
(field)	2.1.6	16
} (return)	2.2.6	28