FORTH Basics

Tektronix Berlin GmbH & Co. KG
modified by K.S.Lee (kyosoon@hanafos.com)

K1297 Training FORTH Basics

Target

Knowledge of fundamental

FORTH

to get the best out of the

K1297 protocol tester

Contents (1)

How to get results ...

- FORTH Arithmetics
- Stack Concept
- Stack Manipulations

You are in command ...

- Increasing the Display Lines
- Reentering a command

Some things change and other do not ...

- Variables
- Constants

Contents (2)

The FORTH engine ...

- Dictionary with it's content
- useful Words
- defining your own Words

Address Manipulation

- Strings
- Changing address contents

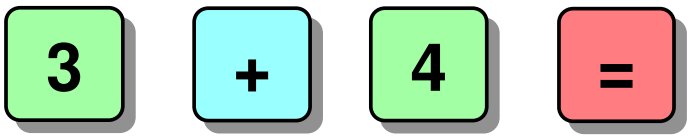
Decisions must be taken ...

- Conditional Phrase
- Comparison and Logical Operators

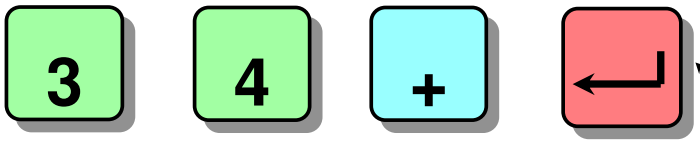
How to get results ...

Reverse Polish Notation and FORTH

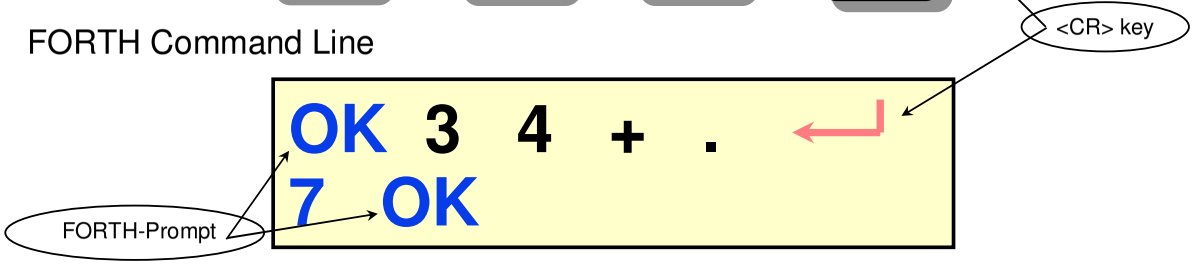
Standard Mathematical Notation



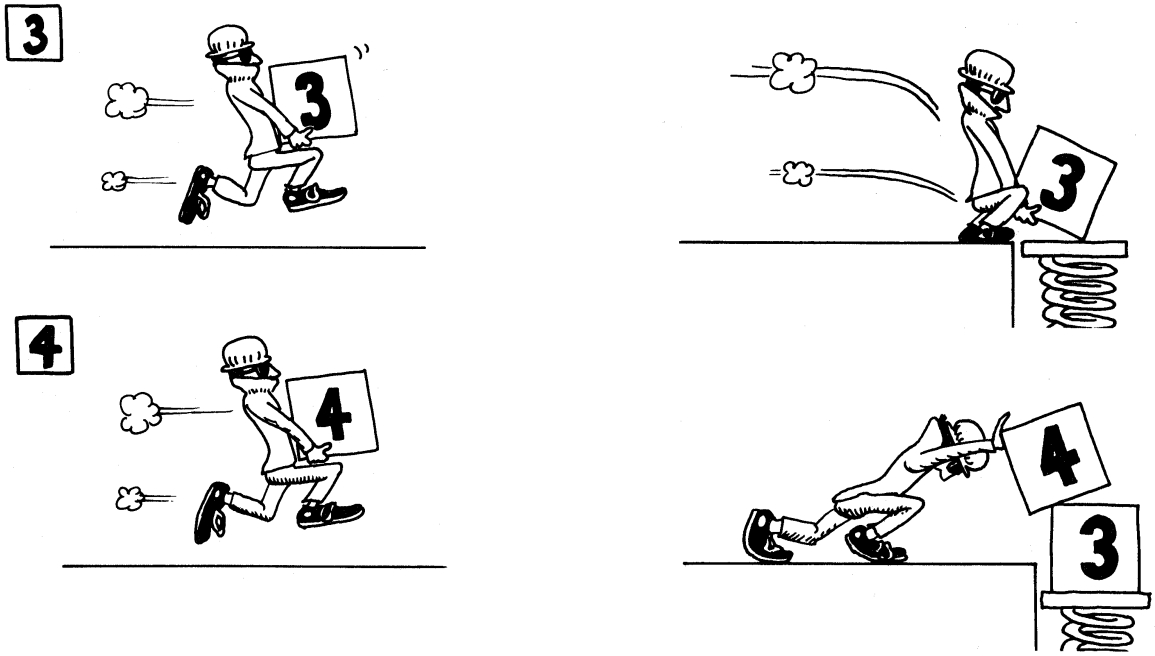
Reverse Polish Notation



FORTH Command Line

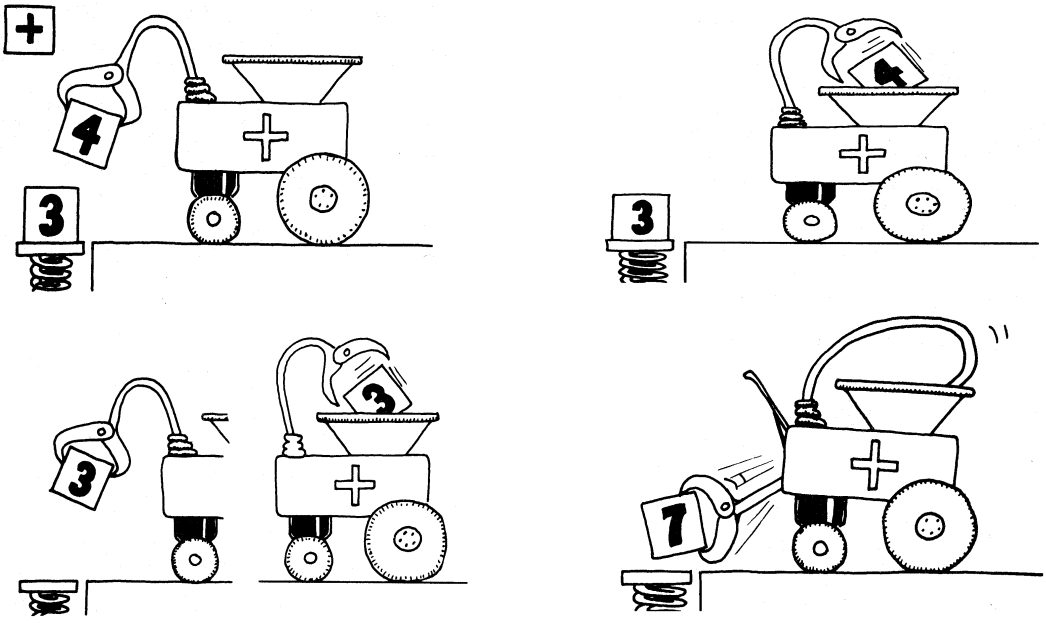


3 and 4 are put on STACK



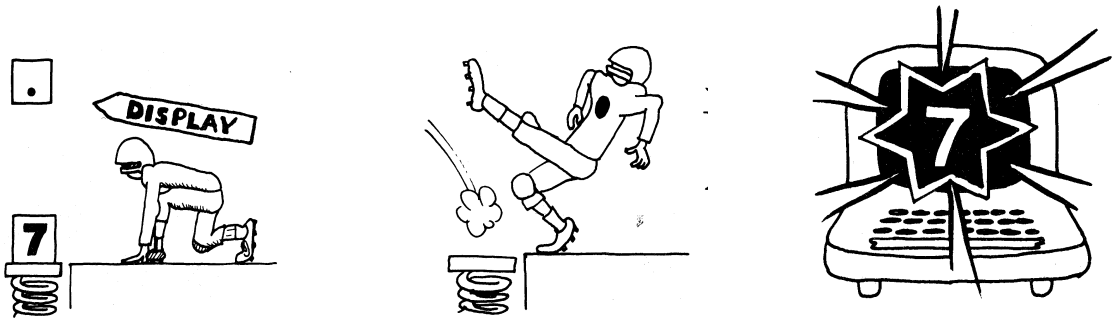
Tektronix®

+ consumes the argument numbers and leaves the result sum on STACK



Tektronix®

. (DOT) removes and prints the **T**op **O**f **S**tack

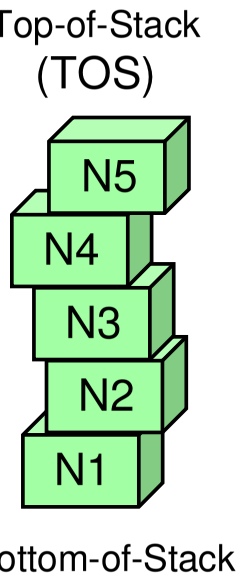


FORTH Arithmetic

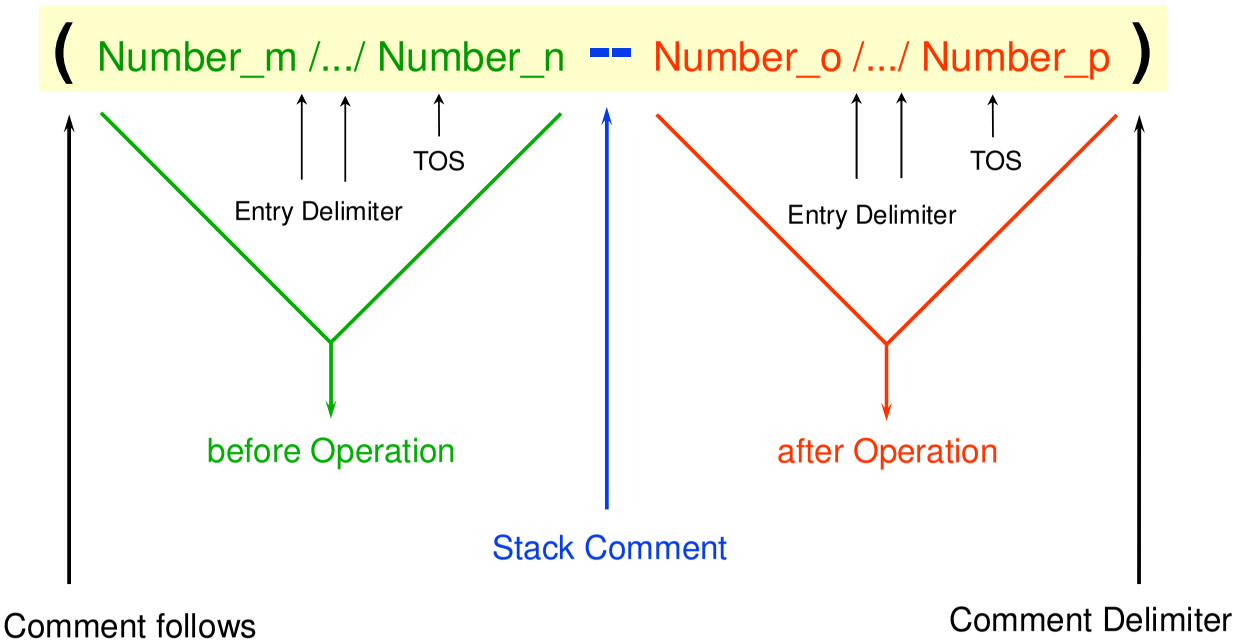
+	Addition	(N1/N2 -- “N1+N2”)
-	Subtraction	(N1/N2 -- “N1-N2”)
*	Multiplication	(N1/N2 -- N1*N2)
/	Integer Division	(N1/N2 -- “N1:N2”)

Parameter or Operational STACK

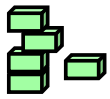
- **L**ast **I**n **F**irst **O**ut Buffer
- **Only Numbers** are a valid Stack content
  - **Values**
  - **Addresses**
- **Commands** may
  - **consume** value(s) from Stack
  - **leave** result(s) on Stack
  - **manipulate** the value(s) on Stack



Stack Effect Notation

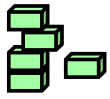


Stack Effects



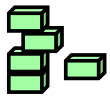
No Stack effect	(--)
Consuming effect	(N --)
Return effect	(-- N)
Consuming and Returning	(N1 / N2 -- N3)
Manipulation effect	(N1 / N2 -- N2 / N1)

Keeping track of the Stack



.	DOT print and removes the TOS	(N --)
S.	prints the TOS without removing	(--)
.S	prints the top 4 entries	(--)
STACK	prints the complete Stack	(--)
SP!	erases all values on Stack and resets the Stack Pointer	(--)

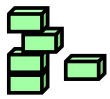
Stack Manipulations on a single entry



DROP	Delete TOS	(N --)
DUP	Copy TOS	(N -- N/N)
SWAP	Exchange top 2 entries	(N1/N2 -- N2/N1)
OVER	Copy 2nd entry on TOS	(N1/N2 -- N1/N2/N1)
ROT	Move 3rd entry on TOS	(N1/N2/N3 -- N2/N3/N1)
n PICK	Copy n+1st entry on TOS	(Nn+1/.../N1 -- Nn+1/ .../N1/Nn+1)

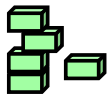
Blank

Stack manipulations on a pair of entries



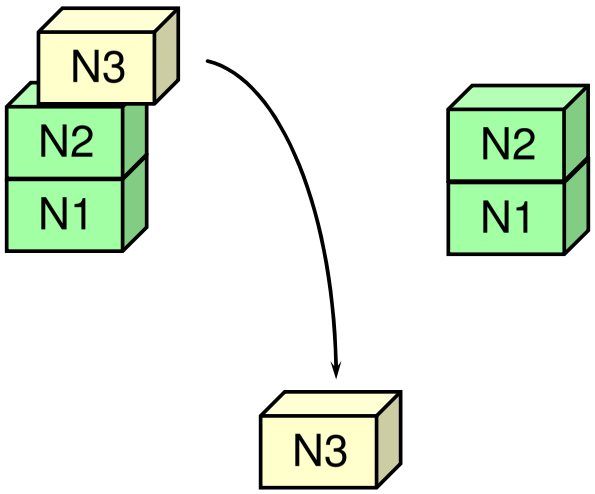
2DROP	Delete top 2 entries	(N1/N2 --)
2DUP	Copy top 2 entries	(N1/N2 -- N1/N2/N1/N2)
2SWAP	Exchange top 2 pair of values	(N1/N2/N3/N4 -- N3/N4/N2/N1)
2OVER	Copy the 2nd pair of entries on TOS	(N1/N2/N3/N4 -- N1/N2/N3/N4/N1/N2)

Stack manipulation - DROP

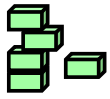


Operation: Deletes the TOS

Stack effect: (N3 --)

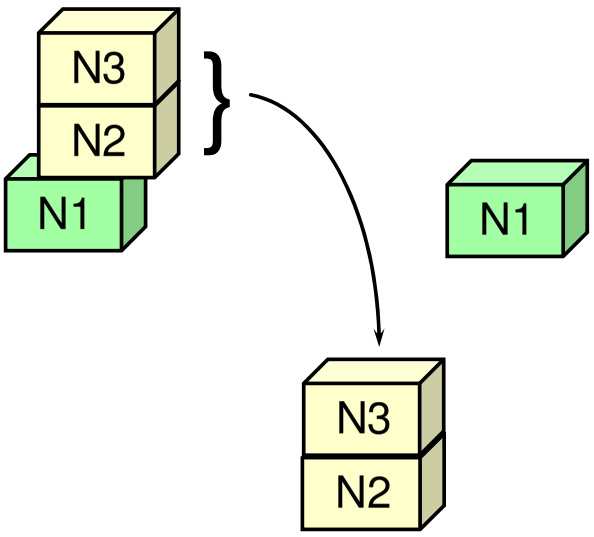


Stack manipulation - 2DROP

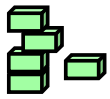


Operation: Deletes the top pair of entries on Stack

Stack effect: (N2/N3 --)

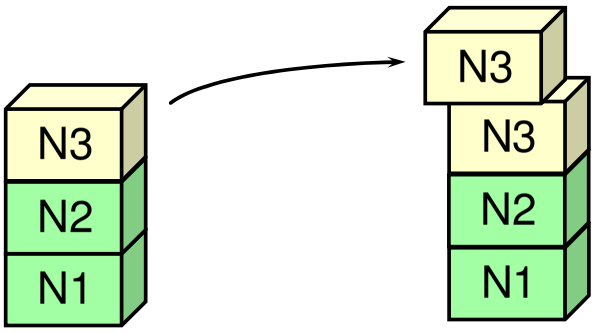


Stack manipulation - DUP

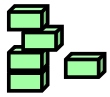


Operation: Copy the TOS

Stack effect: (N3 -- N3/N3)

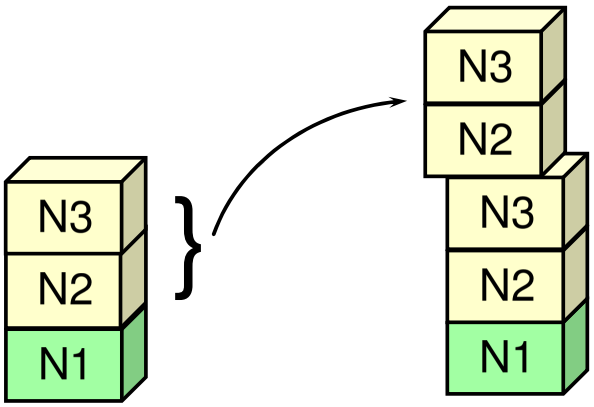


Stack manipulation - 2DUP

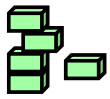


Operation: Copy the top pair of entries

Stack effect: (N2/N3 -- N2/N3/N2/N3)

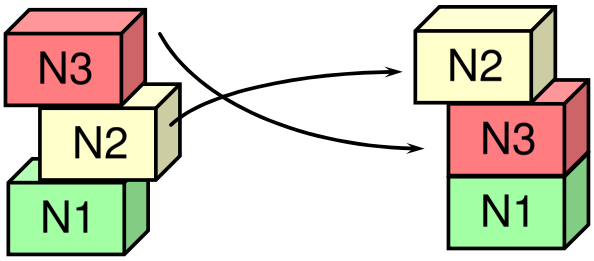


Stack manipulations - SWAP

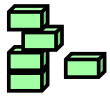


Operation: change the order of the first 2 entries

Stack effect: (N2/N3 -- N3/N2)

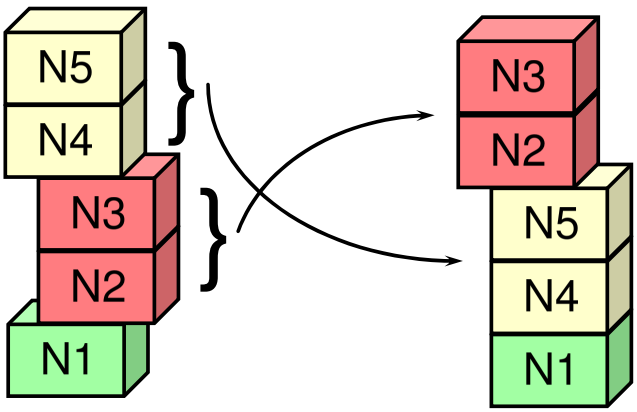


Stack manipulations - 2SWAP



Operation: Change the order of the first two pairs of entries

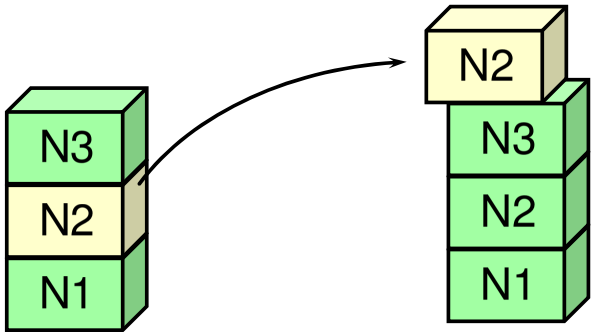
Stack effect: (N2/N3/N4/N5 -- N4/N5/N2/N3)



Stack manipulations - OVER

Operation: Copy the second entry to TOS

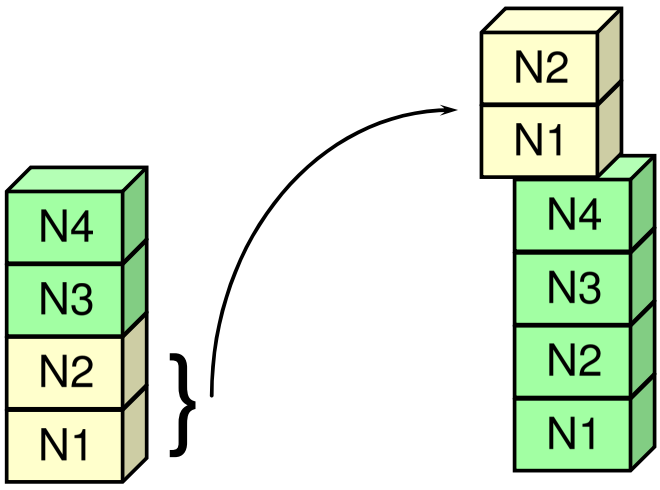
Stack effect: (N2/N3 -- N2/N3/N2)



Stack manipulations - 2OVER

Operation: Copy the second pair of entries to TOS

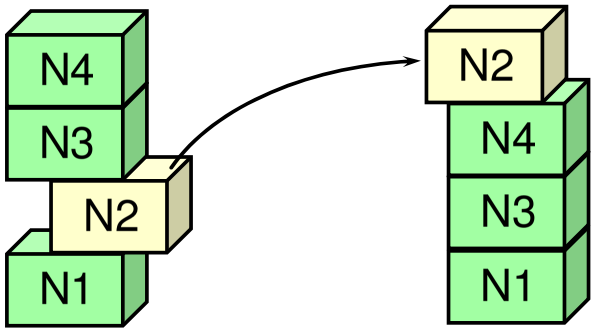
Stack effect: (N1/N2/N3/N4 -- N1/N2/N3/N4/N1/N2)



Stack manipulations - ROT

Operation: Change order of first 3 Stack entries

Stack effect: (N2/N3/N4 -- N3/N4/N2)

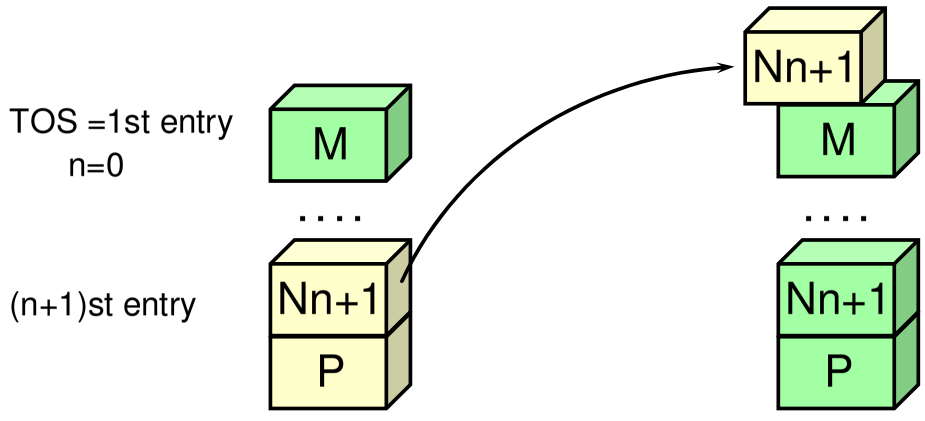


Stack manipulations - n□PICK

Blank

Operation: Copies n+1st entry on TOS

Stack effect: (Nn+1/.../M -- Nn+1/.../M/Nn+1) where Nn+1 is the n+1st entry



You are in Command ...

Keys to increase the number of command lines

CTRL → **Increase by one line**

CTRL ← **Reduce by one line**

CTRL Shift → **Increase by 10 lines**

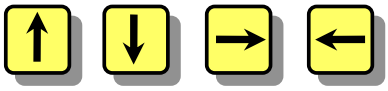
CTRL Shift ← **Reduce by 10 lines**

A maximized command window

Reentering a command

To avoid retyping a command still visible in an upper line use:

You then can step through the command screen with:



Using the key combination:

deletes a character and

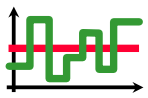
inserts a blank

Useful FORTH Word - 1

SET_WD	**set WD to a specific directory (directory_name --)**
	WD1 " D:/V1.10/USR" SET_WD
" ABCD"	**String(ASCII Code) for ABCD**
	" ABCD" 10 DUMP (if you want to check)
." ABCD"	**Display ABCD to FORTH Command window**
CR	**Carriage Return for FORTH Command Window**
	." HELLO K1297" CR
T." ABCD"	**Display ABCD to trace-window**
TCR	**Carriage Return for trace-window**
	T." HELLO K1297" TCR

Some things change other do not ...

Constant



Definition of a constant:

(--)

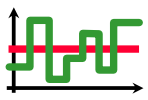
After definition of a constant the value
can be retrieved with it's:

Name (-- Value of the Constant)

The result is the Value of the Constant Name on TOS.

K1297 Training FORTH Basics

Constant - Example



Session

```
OK  10  CONSTANT  MY_CONST
OK  MY_CONST
OK  .
10  OK
```

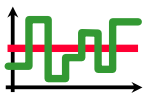
Comments

Definition of MY_CONST

(-- Value of MY_CONST)

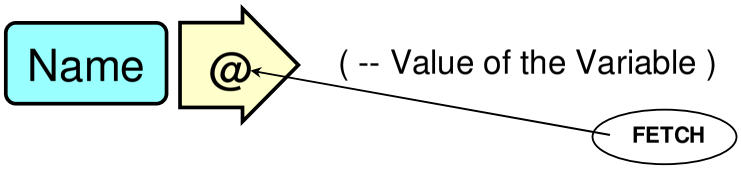
Print TOS (N --)

Variable

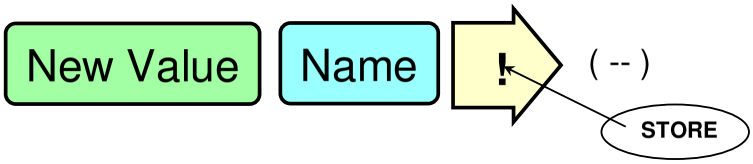


Definition of a variable:

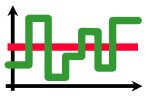
After definition the actual value of variable can be retrieved with:



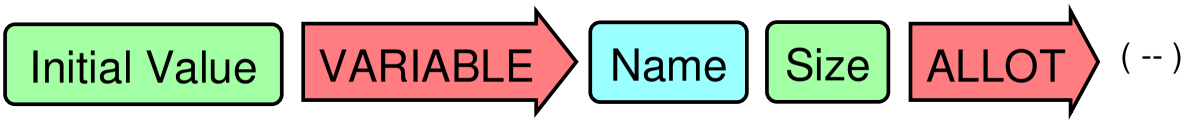
A new value is assigned to a defined variable with:



Variable (Array)

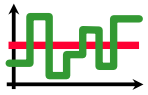


Definition of an array: used for string



- ALLOT: Byte
- ALLOTW: Word (2 Bytes)
- ALLOTL: 4 Bytes

Operating with Variables



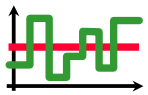
Name Puts Memory Address of Variable on TOS (-- Address)

@ <FETCH> returns the Value (32 bits) of specified Address (Address -- Value)

! <STORE> writes the Value (32 bits) to the specified Address (Value/Address --)

+! <Plus-STORE> adds Value to the content of the given Address (Value/Address --)

Variable - Example



Session

```
OK 10 VARIABLE MY_VAR
OK MY_VAR
OK .
146597 OK MY_VAR
OK @
OK .
10 OK 5
OK MY_VAR
OK !
OK MY_VAR @ .
5 OK
```

Comments

Definition of MY_VAR
(-- Address of MY_VAR)
Print TOS (N --)
(-- Address of MY_VAR)
FETCH (Address -- Value)
Print TOS (N --)
(-- 5)
(-- 5 / Address)
STORE (Value / Address --)
FETCH and PRINT TOS (--)

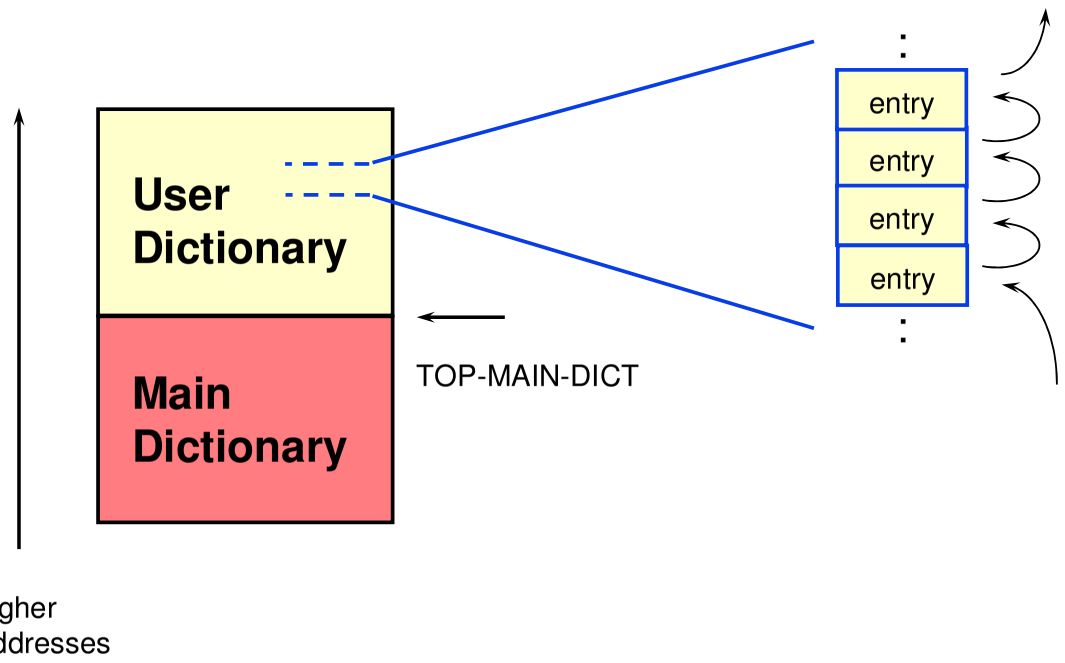
The FORTH Engine ...

The Dictionary ...

- ... has a base called the main or system dictionary
- ... is a linked list of entries

- Entries are linked in the order they were defined, not in lexical order
- User defined entries are added on top of the dictionary
- Entries on top are more complex than entries lower in the dictionary

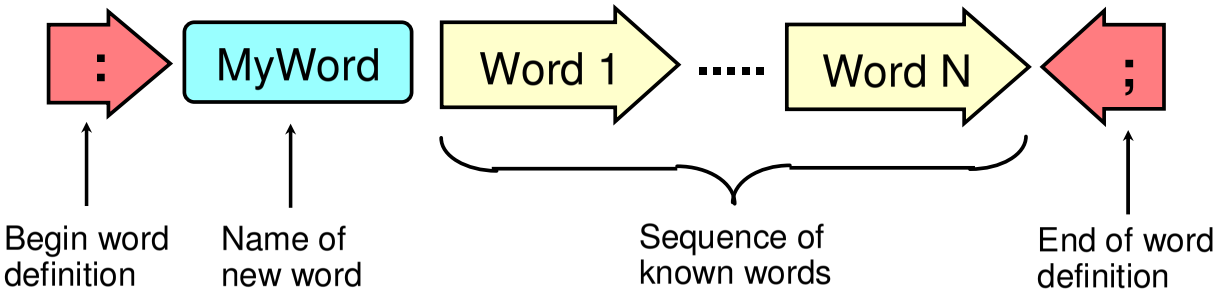
View of the dictionary



Dictionary Entries ...

- ... are commands that can be executed
- ... are called **WORDS**
- ... are created by defining new words using existing words

Definition of new words



Sample Session

```
OK : SQARE DUP * ;
OK 10 SQARE
OK .
100 OK
```

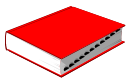

Useful words around the dictionary

Searches and prints out all the words in the dictionary which name contain the *Searchstring*

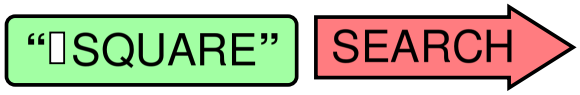
DECOMP Word — Shows the definition of *Word*

FORGET Word — Removes *Word* and all later entries from the dictionary

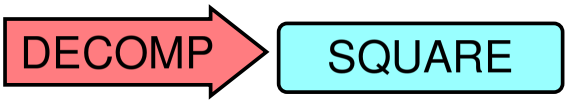
Useful words : Hands On



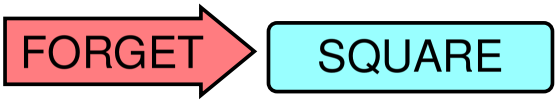
좌측에 보이는 명령어를 차례대로 수행하면서 동작 결과를 본다.



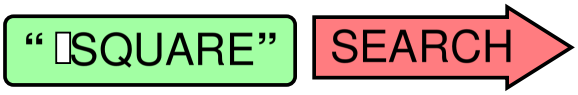
SQUARE가 딕셔너리에 있으면 화면에 출력한다.



워드 내용을 화면에 보여 준다.



워드를 딕셔너리에서 제거한다



워드가 지워졌으므로 화면에는 아무것도 보여지지 않을 것이다.

Programming & Compiling

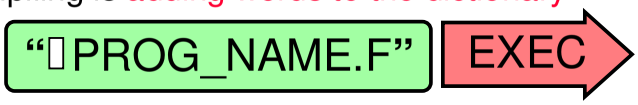
A FORTH program is a text file containing definitions of words

```
: Word 1
   <sequence of words>
;
....
: WORDm <sequence of words> ;

: Word n
   <sequence of words>
   ( may include Word1 through WORDm )
;
```

Usually FORTH programs are saved under the name <PROG_NAME.F>.

Compiling is adding words to the dictionary



Address Manipulation

Address Manipulation (1)

C!	Store byte at address	(b/addr --)
C@	Fetch for bytes	(addr -- b)
W!	Store 16 bits of n at address	(n/addr --)
W@	16 bit fetch	(addr -- n)

Address Manipulation (2)

CMOVE Moves specified count of bytes beginning at address addr1 to addr2
(addr1/addr2/count --)

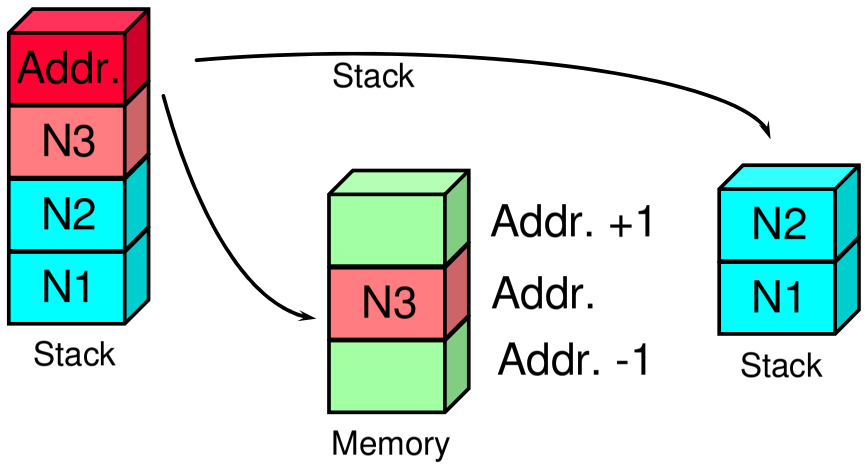
COUNT Leaves the byte address addr2 and byte count n of a text string beginning at addr1
(addr1 -- addr2/n)

TYPE Transmits count characters from addr2 to screen
(addr2/n --)

!STRING Copies the string to the variable
(string var --)

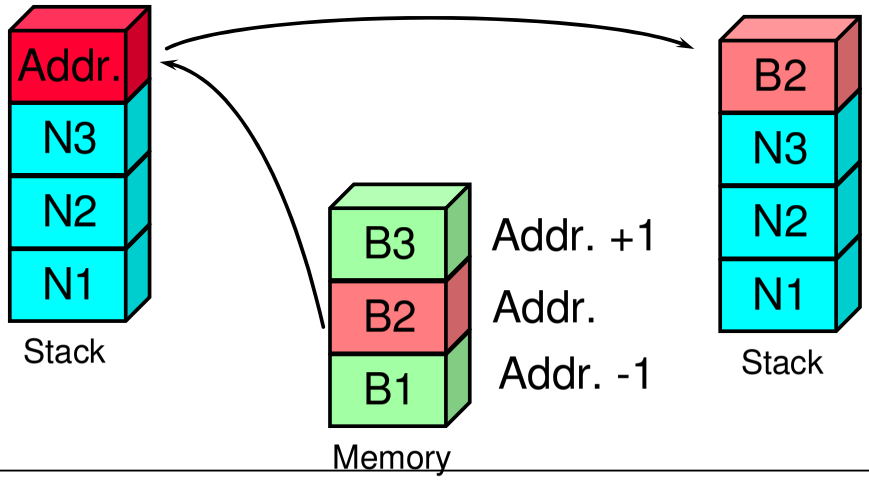
Address Manipulation - C!

- C! (b/addr --)
- Stores 8-bits at address



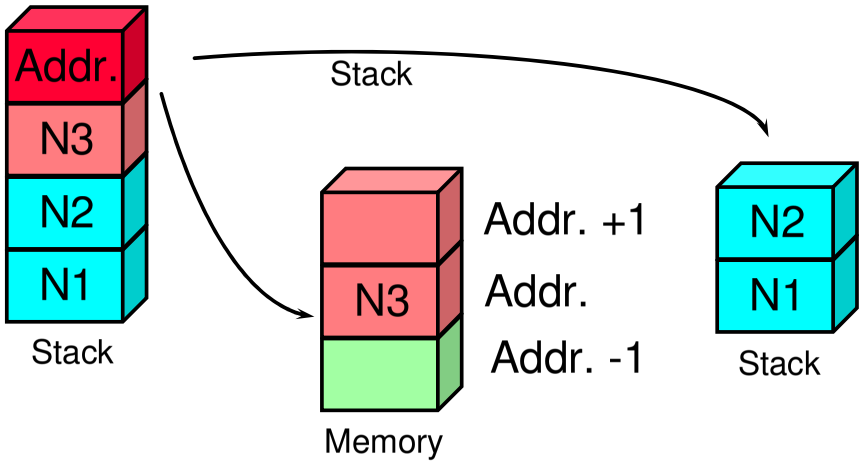
Address Manipulation - C@

- C@ (addr -- b)
- Leaves the 8-bit contents of a memory address



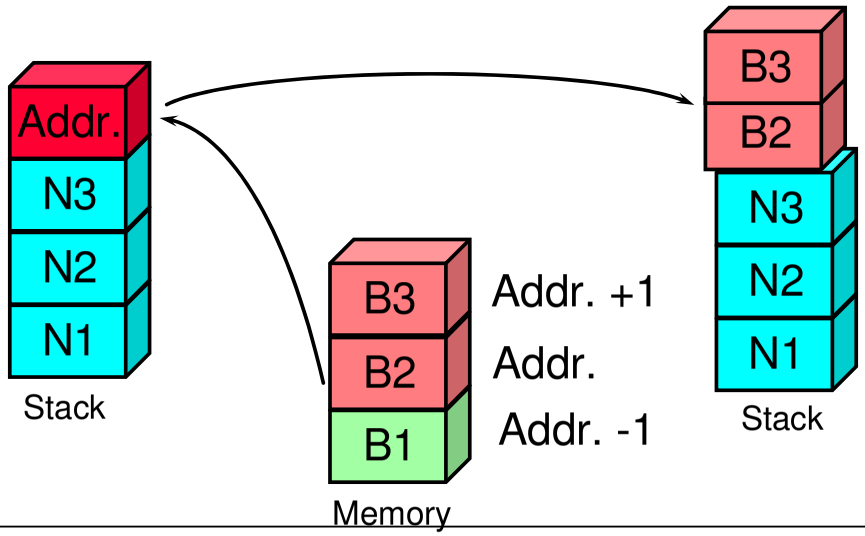
Address Manipulation - W!

- W! (n/addr --)
- Stores 16-bits at address



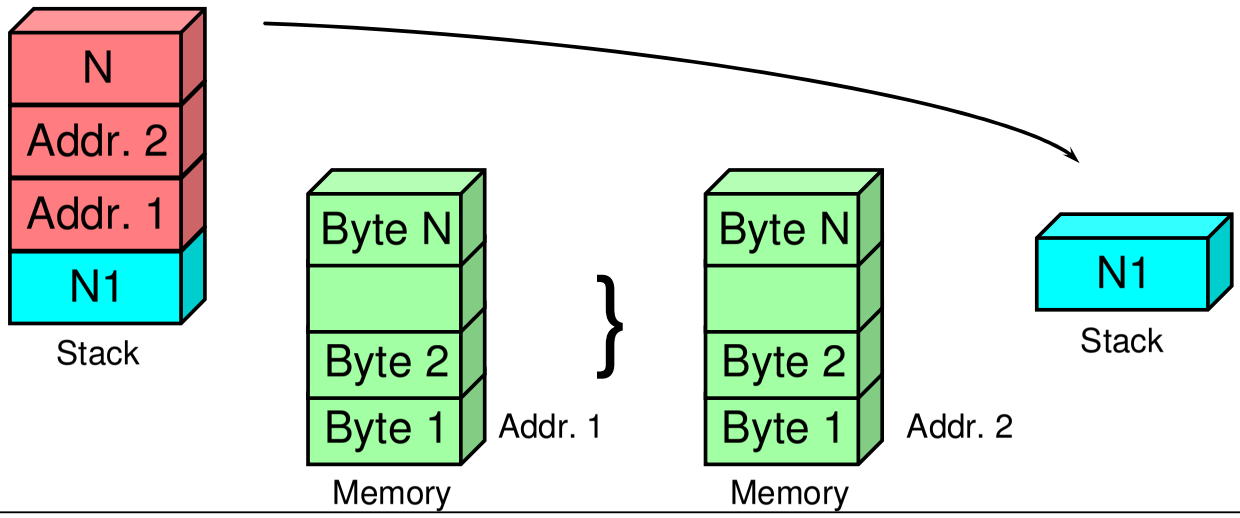
Address Manipulation - W@

- W@ (addr -- n)
- Leaves the 16-bit contents of addr. and addr.+1



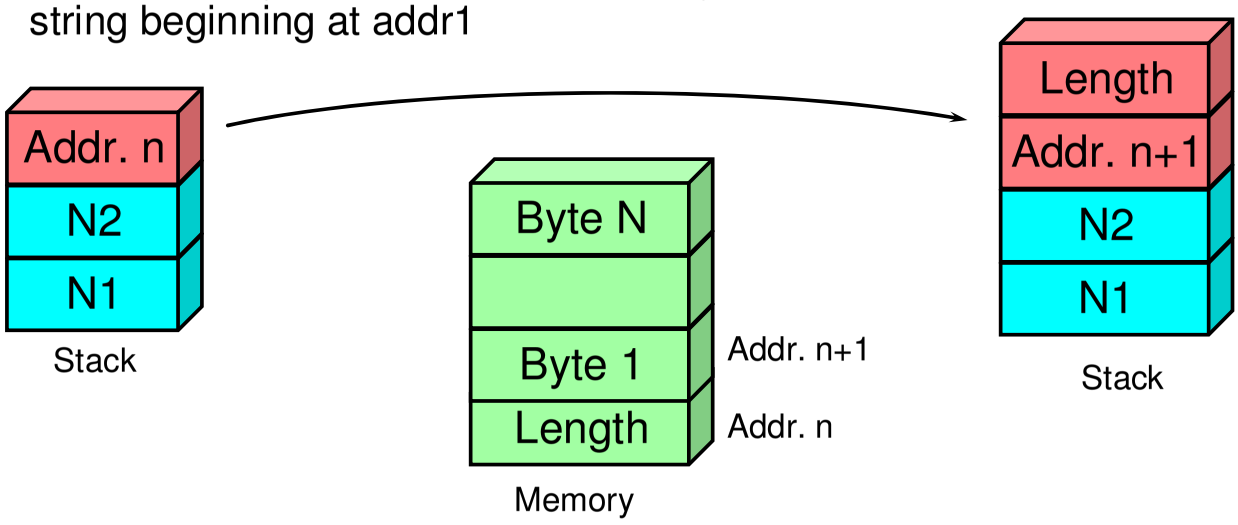
Address Manipulation - CMOVE

- CMOVE (addr1/addr2/count --)
- Moves specified count of bytes beginning at address addr1 to addr2



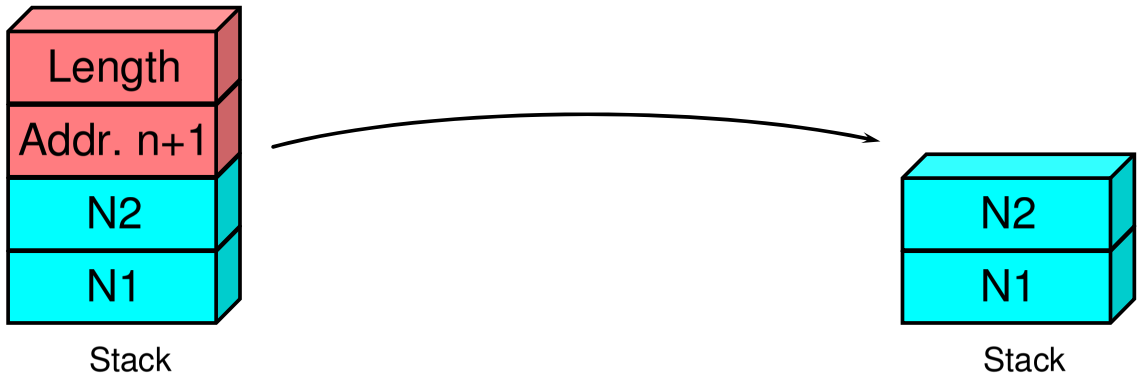
Address Manipulation - COUNT

- COUNT (addr1 -- addr2/n)
- Leaves the byte address addr2 and byte count n of a text string beginning at addr1



Address Manipulation - TYPE

- TYPE (addr2/n --)
- Transmits n characters beginning from “addr2” to the Forth line
- Usually used after the command COUNT
- TTYPE: Like TYPE - Transmits characters inside the screen



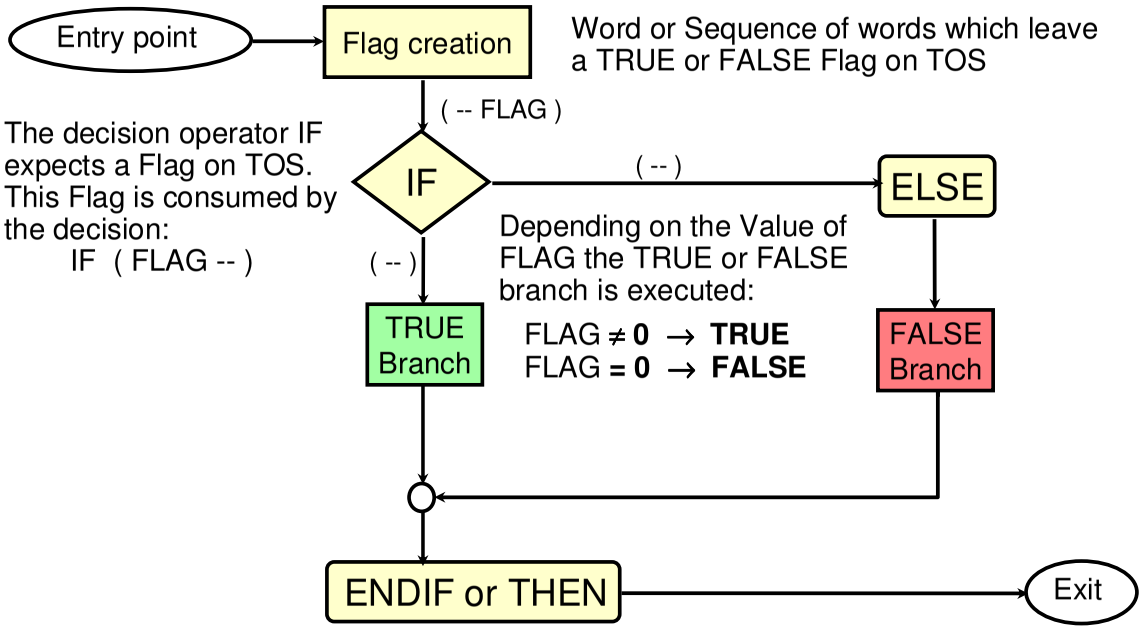
Address Manipulation - !STRING

- !STRING (string var --)
- Copies the string to the variable

```
OK  DECOMP !STRING
OVER   C@   1+   CMOVE   EXIT
OK
OK 0  VARIABLE   MY_STRING
OK " HELLO WORLD"   MY_STRING   @   !STRING
OK MY_STRING   @   COUNT   TYPE
HELLO WORLD OK
```


Decisions must be taken ...

IF ... ELSE ... ENDIF



IF ... ELSE ... ENDIF - Example-1

Session

```
OK  :  TEST ( Flag -- )
->  IF
->      ." True ( Flag <> 0)" CR
->  ELSE
->      ." False ( Flag = 0)" CR
->  ENDIF
->  ;
OK  10  TEST
True ( Flag <> 0)
OK  0 TEST
False ( Flag = 0)
OK
```

Comments

<**: TEST**> calls the inner interpreter

IF consumes the Flag

TRUE Branch

ELSE

FALSE Branch

End of conditional branch

<;> leaves the inner interpreter

RUN-Time Execution

IF ... ELSE ... ENDIF - Example-2

Session

```
( NUMBER -- )
: TEST2
  DUP 0 >   ( FLAG CREATED)
  IF
    DROP .“ POSITIVE “
  ELSE
    0=
    IF      ( NEW FLAG CREATED)
      .” EQUAL TO ZERO”
    ELSE
      .” NEGATIVE”
    THEN
  THEN
;

OK 10 TEST2
POSITIVE
OK 0 TEST
EQUAL TO ZERO
OK
```

Comments

<: **TEST2**> calls the inner interpreter

IF consumes the Flag

TRUE Branch

ELSE

FALSE Branch

End of conditional branch

<;> leaves the inner interpreter

RUN-Time Execution

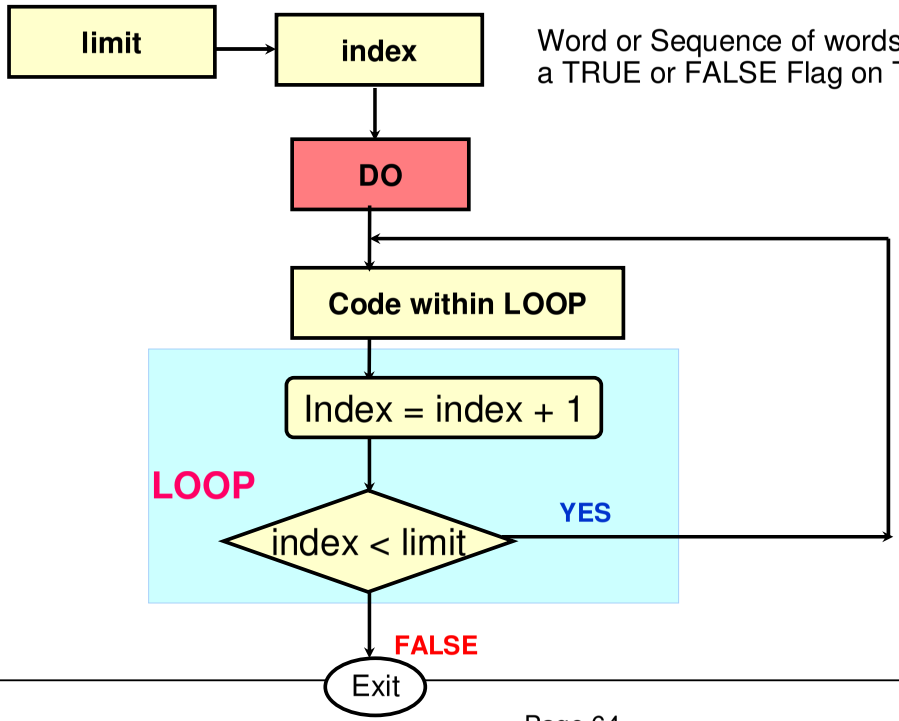
Comparison Operators to create Flags

=	Is M equal N ?	(M/N -- Flag)
>	Is M bigger N ?	(M/N -- Flag)
<	Is M smaller N ?	(M/N -- Flag)
0=	Is N equal 0 ?	(N -- Flag)

Logical Operators to combine Flags

0=	Is Flag1 FALSE ?	(Flag1 -- Flag2)
OR	Is Flag1 or Flag2 TRUE ?	(Flag1/Flag2 -- Flag3)
AND	Are Flag1 and Flag2 TRUE ?	(Flag1/Flag2 -- Flag3)

DO . . . LOOP



DO . . . LOOP - Example

Session

```
0 VARIABLE VAR
: TEST3 ( -- )
  10              ( LIMIT )
  0               ( INDEX )
  DO
     VAR @ .      ( OUTPUT VAR )
     3 VAR +!     ( INCREASE VAR BY 3 )
  LOOP
;

OK   TEST3
0 3 6 9 12 15 18 21 24 27
OK
```

Comments

<: **TEST3**> calls the inner interpreter

Begin of DO construct

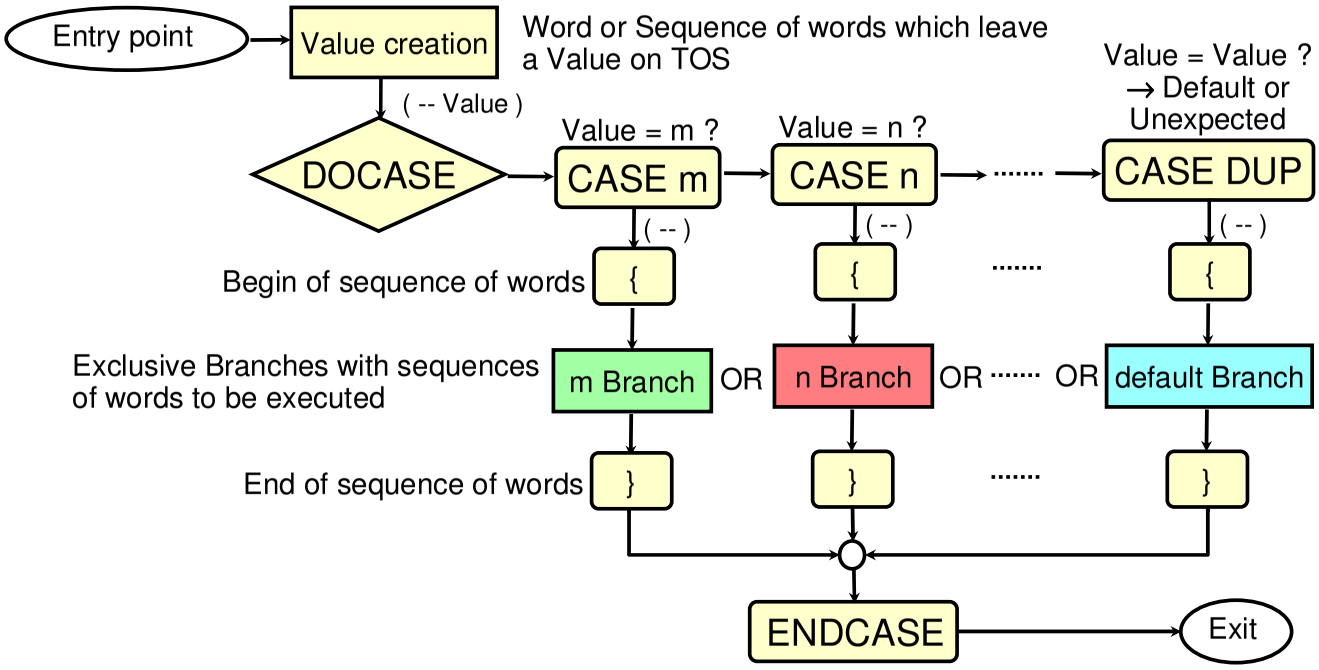
Code within LOOP

End of LOOP construct

<;> leaves the inner interpreter

RUN-Time Execution

DOCASE ... CASE m ... CASE n ... ENDCASE



DOCASE ... ENDCASE - Example

Session

```
OK  :  TEST ( Value -- )
->  DOCASE
->      ORCASE 0
->      CASE 1 { ." Val=0 or 1" CR }
->      CASE 2 { ." Val=2" CR }
->      CASE DUP { ." ?Val" CR }
->  ENCASE
->  ;
OK  2  TEST
Val=2
OK  5 TEST
?Val
OK
```

Comments

<: **TEST**> calls the inner interpreter

Begin of CASE construct

Value = 1 Branch

Value = 2 Branch

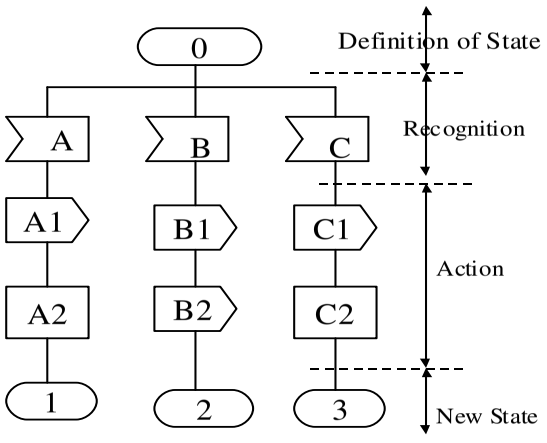
Default Branch

End of CASE construct

<;> leaves the inner interpreter

RUN-Time Execution

SDL Machine Language




```
0  STATE_INIT{

 }STATE_INIT

0 STATE{
  A ACTION{
     A1
     A2
     1 NEW_STATE
  }ACTION

  B ACTION{
     B1
     B2
     2 NEW_STATE
   }ACTION

  C ACTION{
     C1
     C2
     3 NEW_STATE
   }ACTION
}STATE
```

SDL Machine Language: Event

- Function Key
- Timer
- Message
- Event
- IPC

SDL Machine Language: Function Key Event

```
0 STATE_INIT{
   TM_DEF
   UF1   " TEST"   =TM_KEY
   UF2   " BEEP"   =TM_KEY
 }STATE_INIT

0 STATE{
   ?FK_1 ACTION{
    ." TEST: FK1 Key pressed !"  CR
   }ACTION

   ?FK_2 ACTION{
      BEEP
   }ACTION
}STATE
```

SDL Machine Language: Timer Event

- START_TM_TIMER (TIMER_NO\DURATION --)
  - Timer-No : 0 ~ 127
  - Duration : Any-Value >= 1 (Time Unit = 100msec)
    Duration 10 => 1000 msec
- STOP_TM_TIMER (TIMER_NO --)
- ?TM_TIMER (TIMER_NO -- Flag)

SDL Machine Language: Hands On

```
0 STATE_INIT{
  UF1 " GO TO 1" =TM_KEY
  UF2 "         " =TM_KEY
}STATE_INIT

0 STATE{
  ?FK_1 ACTION{
    T." GO TO STATE-1" TCR
    T." Press FK-2 in 3 seconds" TCR
    1 3000 TM_TIMER_START
    1 NEW_STATE
  }ACTION
}STATE
```

ST_EX.F

```
1 STATE_INIT{
  UF1 "         " =TM_KEY
  UF2 " go to 0" =TM_KEY
}STATE_INIT

1 STATE{
  ?FK_2 ACTION{
    T." FK-2 was pressed " TCR
    T." Cancel timer & Go to state-0" TCR
    1 TM_TIMER_STOP
    0 NEW_STATE
  }ACTION
  1 ?TM_TIMER ACTION{
    T." Timer Expired without FK-2 press"
    T." go to state-0" TCR
    0 NEW_STATE
  }ACTION
}STATE
```

```
#IFDEF LOADER
   FORGET LOADER
#ENDIF

: LOADER
  GLOB_TCLR
  TM0  " ST_EX.F"  EXECF
  GLOB_TM_RUN
  TM0
;

LOADER
```

ST_EX.L

SDL Machine Language: Message Event

- **<USER_PART_ID> <MSG_ID> ?RX_SU**

```
#GSM_SCCP  #UDTS  ?RX_SU ACTION{
   T." UNIT_DATA Received !!" TCR
   2 NEW_STATE
}ACTION

#WB_ISUP #WI_ACM  ?RX_SU ACTION{
   T." ISUP-ACM Received !!" TCR
}ACTION
```

- **FORTH word for receiving Message is different dependant on the protocol.**

SDL Machine Language: Message Send

- SEND_MSG (string address --)
- SEND_SCCP_TCAP (string address --)
- FORTH word for sending the message will be different dependant on the protocol
- Example

  " MSG_NAME" SEND_MSG
  or
  " MSG_NAME" EDIT_MSG
  4 #SLS P!
  SEND_MSG