

```

; Defines area:
tks = 177560 ; keyboard status.
tkb = 177562 ; keyboard buffer.
tps = 177564 ; printer status.
tpb = 177566 ; printer buffer.
lcs = 177546 ; clock status.

WALL=1 ; implies to wall in Warehouse.
BOX='# ; define BOX to be the ascii of '#, witch implies box in Warehouse array.
TARGET=1 ; implies to target place in Target array.
EMPTY_CELL=0 ; cell in Warehouse that does'nt have neither box nor wall.
NEW_PAGE=30. ; how many new lines should be printed to have new page.
SPACEBAR=32. ; used to replace letter that should be deleted, on the output screen.
STR_MAX_LEN=20. ; the max size of strings that program receive from the user.
CURSOR='?' ; this symbol will used as cursor when receiving input from the user.
BACK_SPACE=8. ; the ascii of Backspace key, when it printed, it backwards the cursor one
; place whitout deleting on screen chars.

; symbols that printed on the board on the output screen.
sWALL='=' ; the symbol of wall from Warehouse array.
sPLAYER='^' ; the symbol of the player on the output board.
sBOX=' #' ; the symbol of the box on the output board.
sBOX_ON_TARGET='$' ; the symbol of box setting on target place.
sEMPTY_CELL=32. ; ascii of SPACEBAR, used to imply to empty place on the board.
sTARGET='o' ; implies target place on the board.
LF=10. ; Line feed ascii.
CR=13. ; Carriage return ascii.

```

```

; ~~~~~~ main ~~~~~~
; The main function of the program , it initializes the stack, calls function to initialize interrupt vectors,
; and enable them.
; it also does two main steps, the first one is to call getWelcomeInput function which gets the user input.
; the sconded thing is: after printing the board for the first time, it starts infinite loop that ends only when the
; time ends, or
; when the game is finished.

```

```

main:      . =torg+700 ; write the following code in the place: [program starting address in memory]+7000
mov pc,sp ; stack initialization.
tst -(sp)
jsr pc,initInterrupts ; enable inturrupts and initialize there vectors.
jsr pc,getWelcomeInput ; get welcome input: player name, max time , max moves.
mov @#maxMoves,r5 ; initialize r5 as steps counter, if it reaches max steps the game
; will finish.
mov #inp2, @#60 ; change keyboard interrupts handler to be inp2,
mov #200, @#62 ; and give it priority 4 .
mov #clock2, @#100 ; change clock interrupts handler to be clock2.
mov #101, @#tks ; enable interrupts from keyboard.

mov r5,-(sp) ; pass the remaining moves to printInterfaceData.
jsr pc,printInterfaceData ; this is the first time board print.
tst (sp)+ ; clear the stack from passed parameters.

; the main loop, loops until time ends, or external factors come from interrupts.
mLoop:    cmp #1,@#finishProg ; checks if external factor told to finish.
beq mEnd ; if yes , go to main end.

```

```

        tst @#timeLeft          ; check the remaining time.
        ble timeEnd            ; if time end , jump to timeEnd label.
        br mLoop               ; keep looping.

timeEnd:  mov #0, @#tkb         ; disable reading and interrupts from keyboard.
        mov #timeExcded,-(sp)   ; time finished, user lost, print suitable message (pass by stack),
        jsr pc,printStatistics  ; and print the statistics.
        tst (sp)+               ; clear the stack from passed parameters.

mEnd:     halt                  ; are you sure you want to exit this nice dxd9 game ? :P
;-----
;~~~~~ initInterrupts ~~~~~
; enable interrupts and initialize there vectors.

initInterrupts: mov #iot_putc, @#20 ; address of iot trap handler.
               mov #300, @#22      ; that will operate at priority 6. to pause the timer (clock
                                   ; handler) while printing.

               mov #inp1, @#60     ; address of keyboard interrupt handler.
               mov #300, @#62      ; priority 6 , to not be interrupted by the cursor (clock1).
               mov #clock1, @#100  ; put clock interrupts handler to be clock1.
               mov #300, @#102     ; priority 6
               mov #101, @#tkb     ; enable interrupts from keyboard.
               mov #100, @#lcs     ; enable interrupts from the clock.
               rts pc              ; will return to 'main'.
;-----
;~~~~~ interrupt vectors ~~~~~
;inp1:
; this is the keyboard interrupt handler for the first section of the home work, wich responsible to get user input.
; its main work is to get input from the user , controlling the use of 'backspace' and 'enter'. Also
; controls cursor remains.
; it gets in the input pionter to string, the final output after several interrupts is a received string from the user.

inp1:        mov r0,-(sp)         ; backup r0. will used to handle the input char.
               ; r1               ; pointer to the target string.(param passed by register)
               mov r2,-(sp)       ; backup r2.
               mov #cursorStatus,r2 ; r2 pointer to cursorStatus, if it less than 'rate/2' this mean the
                                   ; cursor is visible, else if it is equal or begger then it is hidden.

               movb @#tkb,r0      ; get the input char from the key board to r0.
               mov #101, @#tkb    ; read enable + IE enable.
               cmpb r0,#CR        ; is user pressed 'enter'.
               bne ges2           ; if not , continue proccessing this input char.
               mov @#rate,r0      ; if it is, the the user finished this input, so before go to the next
               asr r0              ; line, clear this line (on the terminal) from the cursor(if it still showed).
               cmp (r2),r0        ; r0 now have 'rate/2' , cmapre cursorStatus to it.
               bge gesEnd         ; if it not still visible on the screen go to gesEnd.
               jsr pc,removeCurrent ; else remove it,
               br gesEnd          ; and go to gesEnd , that means this input string finished.

ges2:        cmpb r0,#BACK_SPACE ; is user pressed 'backspace'.
               bne ges3           ; if not , continue proccessing this input char.
               mov @#rate,r0      ; if it is, then delete the last char from the screen and the array.
               asr r0              ; but first handle the cursor like the previous paragraph.
               cmp (r2),r0
               bge ges22
               jsr pc,removeCurrent
               ; [ why the labels name ges* ?! I don't remeber :( .... ]

```

```

ges22:    movb #BACK_SPACE,-(sp) ; now there is no cursor issue, so move cursor one step back,
        iot                    ; by printing 'backspace' to the printer using traps.
        tst (sp)+              ; clear the stack from passed parameters.
        jsr pc,removeCurrent ; remove the current place char. If there was no chars on the screen,
                                ; this won't do thing. now we finished from the screen, let remove one
        mov #strLen,r2         ; from the array. so make r2 pointer to strLen ( the array length),
        tst (r2)               ; check: is it empty array?
        beq gesRet             ; if yes , return from this interrupt by jumping to 'gesRet'.
        dec r1                 ; if not , point to the last gotten char.
        movb #0,(r1)           ; delete it.
        dec (r2)               ; strLen = strLen - 1
        br gesRet              ; will return from this interrupt by going to gesRet.

ges3:    ; ges3 maens the input is normal character, and it will handle it.
        mov #strLen,r2         ; make r2 pointer to strLen ( the array length).
        movb r0,-(sp)          ; echo.
        iot
        tst (sp)+              ; clear the stack from passed parameters.
        movb r0,(r1)+          ; save it in the suitable place in the array.
        inc (r2)               ; increase the length.
        cmp (r2),#STR_MAX_LEN ; did the array reached max size of arrays.
        blt gesRet             ; if not return from this interrupt. else:
gesEnd:  jsr pc,printNewLine    ; gesEnd is to say that this input string finished.
        mov #0,(r2)            ; clear strLen for next use.
        mov #1,@#finishWait    ; told that this string ready to use.
gesRet:  mov (sp)+,r2           ; restore r2.
        mov (sp)+,r0           ; restore r0.
        rti                    ; return from interrupt.
; -----

```

;inp2:

; this is the keyboard interrupt handler for the second section of the home work, wich responsible
; to get one letter
; from the user each time. this letter means the direction the user want the player move to.
; so , after getting the letter it checks if it correct (one of the defined letters for this game), then
; it done the move.
; if the move is not legal , the step won't be done.
; then it prints the board and the interface data, checks the status (success/max moves reached) and
; deals whith it.
; all of this done by using this functions: isCorrect , doStep, printInterfaceData, updateNumOfCovrd
; and isSuccess.
; its parameters is: remaining moves (passed by register r5).

```

inp2:    movb @#tkb, r0         ; get direction letter from the keyboard.
        mov #101, @#tkb       ; read enable + IE enable.

        tst -(sp)              ; allocate place in the stack to be used by "isCorrect" to return result.
        jsr pc,isCorrect        ; call "isCorrect". direction passed using r0 as parameter.
        cmp #1,(sp)+           ; test the result. 1 = correct , else = non correct.
        beq i2                 ; if it is correct, continue proccessing.
        rti                    ; if the letter in not correct, return from interrupt.
i2:      tst -(sp)              ; allocate place in the stack to be used by "doStep" to return result.
        jsr pc,doStep           ; call "doStep" function. direction passed using r0 as parameter.
        cmp #1,(sp)+           ; test the result. if it 1 then the step was legal and it done,
        beq i3                 ; so continue. else :
        rti                    ; if the step is illegal, return from interrupt.

```

```

i3:      dec r5          ; this step was done fine, so decrease remaining moves by 1.
        mov r5,-(sp)    ; pass remaining moves as parameter to printInterfaceData, by stack.
        jsr pc,printInterfaceData ; print remaining moves, remaining time , and the board.
        tst (sp)+       ; clear the stack from passed parameters.
        jsr pc,updateNumOfCovrd ; update the value of the variable trgtsCovered.
        mov r1,-(sp)    ; back up r1 , cuz it will be used by isSuccess.
        jsr pc,isSuccess ; call for "isSuccess" to check after this move result.
        cmp #1,r1       ; check the result of "isSuccess" function. 1 success.
        beq success     ; if it is success jump to 'success'.
        mov (sp)+,r1    ; esle restore r1.
        tst r5          ; check remaining moves, if it is zero and it was not success status
        beq movesExceeded ; jump to ' movesExceeded' to finish the game in the suitable way.
        rti             ; else return from interrupt.

```

```

success:      mov #0, @#tks      ; disable reading and interrupts from keyboard.
        jsr pc,printNewLine    ; print new line,
        mov #name,-(sp)       ; the print the user name,
        jsr pc, printStr ; using printStr.
        tst (sp)+             ; clear the stack from passed parameters.
        mov #msgYouWon,-(sp) ; then print the suitable message for the win status.
        jsr pc, printStr      ; print message passed by stack.
        tst (sp)+             ; clear the stack from passed parameters.
        mov #1,@#finishProg ; told the program that the game is finished.
        rti                   ; return from interrupt.

```

```

movesExceeded:      mov #0, @#tks      ; disable reading and interrupts from keyboard.
        mov #mvsExcded,-(sp) ; print the suitable message for the win status.
        jsr pc,printStatistics ; print message passed by stack.
        tst (sp)+             ; clear the stack from passed parameters.
        mov #1,@#finishProg ; told the program that the game is finished.
        rti                   ; return from interrupt.

```

```

;inp3:
; this is the keyboard interrupt handler used for " press any key to continue"
; so the input value will be ignored.

```

```

inp3:      mov r0,-(sp) ; backup r0.
        movb @#tkb, r0 ; get the input from the keyboard.
        mov #101, @#tks ; read enable + IE enable.
        mov (sp)+,r0 ; restore r0. ( the input lost here)
        mov #1,@#finishWait ; told the program that the user pressed key;
        rti             ; return from interrupt.

```

```

; -----
; clock1:
; this is the clock interrupt handler used for the first section of the home work, and it is responsible
; to show and hide the cursor.
; show time = 0.5 second
; hide time= 0.5 second
; the time calculated comparing to 'rate'. every 'rate/2' ticks means 0.5 second.

clock1:      mov r0,-(sp)          ; backup r0.
             mov r1,-(sp)          ; backup r1.
             mov #cursorStatus,r1 ; make r1 pointer to 'cursorStatus', if it is less than 'rate/2' this mean
                                     ; cursor should be visible, else if it is equal or bigger, then it is hidden.
             mov @#rate,r0         ; make r1 pointer to 'rate'.
             inc (r1)               ; every clock interrupt increase cursorStatus by 1.
             cmp (r1),r0            ; compare it with rate
             blt clc1               ; if it less jump to clc1.
             mov #0,(r1)           ; else reset it to 0.
clc1:        cmp #0,(r1)            ; if cursorStatus = 0 then
             beq curShow           ; show it , until it reaches 'rate/2' ( = 0.5 second).
             asr r0                 ; else make r0 = ' rate/2'.
             cmp (r1),r0            ; compare cursorStatus with ' rate/2',
             beq curHide           ; if equal hide it, else leave it as it.
             br clck1Ret           ; jump to clck1Ret to return.

curShow:     movb #CURSOR,-(sp)     ; print the cursor on the screen.
             iot
             tst (sp)+
             movb #BACK_SPACE,-(sp) ; move terminal cursor one step back.
             iot                   ; by printing backspace using traps.
             tst (sp)+
             br clck1Ret

curHide:     jsr pc,removeCurrent ; remove the cursor from the screen.
clck1Ret:    mov (sp)+,r1           ; restore r1.
             mov (sp)+,r0           ; restore r0.
             rti

; -----
; clock2:
; this is the clock interrupt handler used for the second section of the home work, and it is
; responsible to calculate
; time.
; simply it decrease 1 from 'timeLeft' every clock tick. this means every 'rate' ticks , timeLeft actually
; decreased by one second.

clock2:      tst @#timeLeft; if timeLeft =0
             beq clk2Ret           ; don't decrease it any more, keep it 0.
             dec @#timeLeft        ; decrease timeLeft by 1.
clk2Ret:     rti                   ; return from interrupt.

; -----
; clock3:
; this is the clock interrupt handler used in the stage " press any key to continue", so we neither
; see the cursor of clock1 nor we lose time of clock2.

clock3:      rti                  ; do nothing. just return.
; -----

```

```

; -----
; the iot trap handler. its work to print char to screen without losing game tim ( because output takes
; long time reletively).
; it gets the char as parameter passed using stack.

```

```

iot_putc:      tstb @#tps          ; printing using busy wait.
               bpl iot_putc
               movb 4(sp), @#tpb   ; send the char to the screen.
               rti

```

```

; -----
; ~~~~~~ getWelcomeInput ~~~~~~

```

```

; getWelcomeInput:
; this function considered one of the main functions, it handling the first section of the home work.
; it asks for input by printing suitable messages, then it gets the input, proccesses it and saves it to the
; right place.
; input from the user.
; output: updating globals - name, timeLeft, maxMoves
; usage:

```

```

; -----
; jsr pc,getWelcomeInput
; -----

```

```

getWelcomeInput:  mov r1,-(sp)          ; backup r1. will be used as pointer to strings that
                                   ; will get input.

                 ; get player name.
                 mov #msgEntrName,-(sp) ; pass as param the suitable message to ask for player name.
                 mov #name,r1          ; get the name into 'name' string.
                 jsr pc,askAndGetInput  ; call 'askAndGetInput' function.
                 tst (sp)+              ; clear the stack from passed parameters.

                 ; get max seconds.
                 mov #msgMaxTime,-(sp) ; pass the as param suitable message to ask for max time.
                 jsr pc,clrTmpStr        ; clear TmpStr from previous uses.
                 mov #tmpStr,r1         ; get the max time as string into it by
                 jsr pc,askAndGetInput  ; calling 'askAndGetInput' function.
                 tst (sp)+              ; clear the stack from passed parameters.
                 tst -(sp)              ; allocate place in the stack, strToNum will use it to the result.
                 mov #tmpStr,-(sp)      ; pass the string to it.
                 jsr pc,strToNum         ; convet the string to number and return the result by stack.
                 tst (sp)+              ; clear the stack from passed parameters.
                 mov (sp)+,r1           ; put the result (max time) into r1.
                 mul @#rate,r1          ; multiply it with 'rate'.
                 mov r1,@#timeLeft      ; copy it to 'timeLeft'. so now 'timeLeft' contianing max time with
                                   ; clock ticks units. every 'rate' ticks = 1 second.

                 ; get max moves.
                 mov #msgMaxMoves,-(sp) ; pass the as param suitable message to ask for max moves.
                 jsr pc,clrTmpStr        ; clear TmpStr from previous uses.
                 mov #tmpStr,r1         ; get the max moves number as string into it by
                 jsr pc,askAndGetInput  ; calling 'askAndGetInput' function.
                 tst (sp)+              ; clear the stack from passed parameters.
                 tst -(sp)              ; allocate place in the stack, strToNum will use it to the result.
                 mov #tmpStr,-(sp)      ; pass the string to it.
                 jsr pc,strToNum         ; convet the string to number and return the result by stack.
                 tst (sp)+              ; clear the stack from passed parameters.
                 mov (sp)+,@#maxMoves   ; copy the result to 'maxMoves'.

```

```

; this peice of code implements: Press any key to continue.
mov #inp3, @#60      ; change keyboard interrupt handler to inp3. it ignores the input data.
mov #clock3, @#100    ; no '?' cursor anymore.
jsr pc,newPage        ; get new page.
mov #name,-(sp)       ; print player name.
jsr pc, printStr
tst (sp)+
mov #msgPressKey,-(sp) ; pass suitable message.
jsr pc,askAndGetInput ; since inp3 don't get input, this line won't change anything.
tst (sp)+

mov (sp)+,r1          ; restore r1.
rts pc

```

```

; prints message passed to it then gets input into string(passed into r1)
askAndGetInput: mov 2(sp),-(sp) ; pass the message to be printed.
jsr pc, printStr
tst (sp)+
jsr pc, printNewLine
jsr pc,getInp1ToStr      ; get input to the str pointed by r1.
jsr pc, printNewLine
rts pc

```

```

; get input to the str pointed by r1. it counts on iterrupts.
; it will stay in loop writing to the string by inp1 kb interrupt , until it gets an
; sign to (finishWait = 1).
; this sign is received if one of following two events happen: 1- the user pressed 'enter'
; 2- the string length reaches MAX_STR_LEN.
getInp1ToStr: mov #101, @#tks ; enable keyboard reading and iterrupts. becuase it is input time
mov #0,@#strLen
mov #0,@#finishWait
gtLoop: cmp #1,@#finishWait ; instead of "wait".
bne gtLoop
mov #0, @#tks ; disable reading from keyboard (because out of this function
rts pc ; should not receive input)

```

```

; -----
; ~~~~~~ getRltvPlace ~~~~~~

```

```

; getRltvPlace:
; convertes the player place from coordinates to relative place from begining of array.
; input : array adress , row , column (by stack)
; output: relative place ( by stack).
; usage:
; -----
; tst -(sp) ; will contain the result
; mov [the array address],-(sp)
; mov row ,-(sp)
; mov column,-(sp)
; jsr pc,getRltvPlace
; add #6,sp
; mov (sp)+,result
; -----

```

```

getRltvPlace: mov r1,-(sp) ; backup r1.
mov r3,-(sp) ; backup r3.
;12(sp) = array address ; 10(sp)=row ; 6(sp) = column
mov 12(sp),r1 ; make r1 as apointer to the beginning of the array.

```

```

mov 10(sp),r3 ; the next three lines calculates the absolute place with the following ...
mul nCols,r3  ; formula: [absolute place]=[row]*[the number of columns] + [column]
add 6(sp),r3
add r3,r1      ; [relative place]=[the beginning of the array address]+[absolute place]
mov r1,14(sp) ; 14(sp)=place to return the result
mov (sp)+,r3  ; restore r3.
mov (sp)+,r1  ; restore r1.
rts pc        ; return.

```

```

; -----
; ~~~~~ updateNumOfCovrd ~~~~~
; updateNumOfCovrd:
; updates this two variables:
; numOfTrgts - number of target places in the game. (actually it needs to be updated just once
; in the game beginning)
; trgtsCovered - the number of target places on the board, that covered with box. it should be updated
; after every player legal move.
; it works by scanning Warehouse, Targets arrays.
; input : Warehouse, Targets, nRows, nCols (globals)
; output: updating trgtsCovered, numOfTrgts
; usage:
; -----
; jsr pc,updateNumOfCovrd
; -----

```

```

updateNumOfCovrd: mov r0,-(sp)  ; backup r0.
                  mov r1,-(sp)  ; backup r1.
                  mov r2,-(sp)  ; backup r2.
                  mov r3,-(sp)  ; backup r3.
                  mov #Warehouse,r0 ; make r0 pointer to the beginning of Warehouse.
                  mov #Targets,r1  ; make r1 pointer to the beginning of Targets.
                  mov @#nRows,r3   ; put in r3 the number of rows (nRows).
                  mul @#nCols,r3   ; r3 = [number of culomns] * [number of rows], now r3
                                   ; contain the number of the cells so
                                   ; r3 will be a counter that will be used to scan the Targets array

                  mov #0,@#numOfTrgts ; reset to zero.
                  mov #0,@#trgtsCovered ; reset to zero.
uNOC1:  tst r3      ; as long as r3 not zero, the scan of Warehouse and Targets didn't finished , so continue
                  beq uNOC3      ; if r3 = 0 the the scan have finished. and update finished.
                  dec r3         ; decreasing the counter
                  cmpb (r1),#TARGET; check the current place of Targets, is it target cell?
                  bne uNOC2      ; if not, continue scanning, else
                  inc @#numOfTrgts ; increase by 'numOfTrgts' 1
                  cmpb (r0),#BOX  ; does the current place at Warehouse have a box?
                  bne uNOC2      ; if not means this target place not covered, continue scanning.
                  inc @#trgtsCovered ; else, it is coveered target, count it.
uNOC2:  inc r0      ; forward r0 to the next place in Warehouse.
                  inc r1      ; forward r1 to the next place in Targets.
                  br uNOC1     ; and go back to uNOC1 loop , which continue scanning.

uNOC3:  mov (sp)+,r3 ; restore r3.
                  mov (sp)+,r2 ; restore r2.
                  mov (sp)+,r1 ; restore r1.
                  mov (sp)+,r0 ; restore r0.
                  rts pc      ; return.

```

```

;-----
;~~~~~ isSuccess ~~~~~
;isSuccess:
; checks the current status of the game, is it success case, in other words , are eachone of the
; target palces covered with box. if yes it is success, else not.
; woking idea very simple : is the number of covered targets 'trgtsCovered' equals the number
; of all the targets 'numOfTrgts' . if yes it is success, else not.
; input : numOfTrgts, trgtsCovered (globals)
; output: if *success returns 1 (by register r1)
;         *not success returns 0 (by register r1)
; usage example:
;-----
; mov r1,-(sp) ; back up r1.
; .....
; jsr pc,isSuccess ; call for "isSuccess" to check after this move result.
; cmp #1,r1 ; check the result of "isSuccess" function. 1 success.
; beq [success]
; .....
; mov (sp)+,r1 ; restore r1.
;-----

isSuccess:    cmp @#numOfTrgts,@#trgtsCovered ; is the number of covered targets equals the number
              bne notSuccess ; of all the targets. if yes it is success, else not.
              mov #1,r1 ; it is Success case, so return 1 by using r1
              rts pc ; return.
notSuccess:  mov #0,r1 ; it is not success case, so return 0 by using r1
              rts pc ; return.
;-----
;~~~~~ input/output functions ~~~~~
;-----
; printStr:
; prints string char by char , using iot trap, until reaches the end of the string (0).
; input: string address (by stack)
; output: printing the string to the screen.
; usage :
;-----
; mov [string address],-(sp)
; jsr pc, printStr
; tst (sp)+
;-----

printStr:    mov r3,-(sp) ; backup r3.
              mov 4(sp),r3 ; make r3 pointer to the string.
pS1:         tstb (r3) ; test if the current byte is 0,
              bne .+6 ; if not, continue.
              mov (sp)+,r3 ; if yes, means we reached the end of the string, so restore r3
              rts pc ; and return.
              movb (r3)+,-(sp) ; pass the current byte (char) to 'iot', and foreword the pointer.
              iot ; print it.
              tst (sp)+ ; clear the stack from passed parameters.
              br pS1 ; jump pS1 to continue.
;-----

```

```

;-----
; removeCurrent:
; removes from the screen the char in the current place on the screen.
; input: nothing
; output: replacing the current char with whitespace (on screen).
; usage :
;-----
; jsr pc,removeCurrent
;-----

removeCurrent:  movb #SPACEBAR,-(sp) ; replacing the char with whitespace.
                iot
                tst (sp)+
                movb #BACK_SPACE,-(sp) ; move cursor one step back to return to it previous position.
                iot
                tst (sp)+
                rts pc
;-----

; printNewLine:
; prints new line to the screen by printing CR and LF.
; input: nothing
; output: prints new line (on screen).
; usage :
;-----
; jsr pc,printNewLine
;-----

printNewLine:  mov #CR,-(sp)          ; print the ascii of CR
                mov #LF,-(sp)        ; print the ascii of LF
                iot
                tst (sp)+
                iot
                tst (sp)+
                rts pc
;-----

; newPage:
; printing NEW_PAGE (default = 30) new lines.
; input: nothing
; output: starts new page (on screen).
; usage :
;-----
; jsr pc,newPage
;-----

newPage:      mov r0,-(sp)            ; backup r0.
                mov #NEW_PAGE,r0    ; put number of new lines to be printed in r0 to be used as counter.
np1:          tst r0                  ; test the counter,
                beq np2              ; if = 0 then the requested amount of newlines is printed, go to finish.
                jsr pc,printNewLine  ; else print new line,
                dec r0               ; decrease the counter,
                br np1              ; continue.
np2:          mov (sp)+,r0            ; restore r0.
                rts pc              ; return.
;-----

```

```

;-----
; printStatistics:
; at the end of the game (if the player lost) it prints a suitable message to explain why he lose,
; and then prints: "Only X out of Y targets are covered.", where X and Y replaced with the suitable values.
; input :    * fail message address (by stack)
;          *trgtsCovered, numOfTrgts , name, msgStatistics ( globals)
; output: printed data to the screen.
; usage:
;-----
; mov [message address],-(sp)
; jsr pc,printStatistics
; tst (sp)+      ; clear the stack.
;-----

```

```

printStatistics: mov r0,-(sp)  ; backup r0.
                jsr pc,printNewLine
                mov #name,-(sp)  ; print the name of the player.
                jsr pc, printStr
                tst (sp)+
                mov 4(sp),-(sp)  ; prints the suitable message to explain why he lose.
                jsr pc, printStr
                tst (sp)+
                jsr pc,printNewLine
                mov #msgStatistics,r0 ; make r0 points to the start of string "Only X out of Y ..."
pSt1:           cmpb (r0),#X      ; print until X.
                beq pSt2         ; if it reached 'X' jump to pSt2.
                movb (r0)+,-(sp) ; else print the current letter.
                iot
                tst (sp)+
                br pSt1          ; continue to the next letter .

                ; print the number of covered targets instead of X.
                ; to do this we need to convert the number to string.
pSt2:           jsr pc,clrTmpStr  ; clear tmpStr to put the converted number in it.
                mov @#trgtsCovered,-(sp) ; pass the number to 'numToStr' by stack.
                mov #tmpStr,-(sp) ; pass the tmpStr to 'numToStr' by stack.
                jsr pc,numToStr   ; convert the number to string and return it into 'tmpStr'.
                tst (sp)+        ; clear the stack from passed parameters.
                tst (sp)+        ; clear the stack from passed parameters.
                mov #tmpStr,-(sp) ; print the number of covered targets to screen.
                jsr pc, printStr
                tst (sp)+
                inc r0            ; point to next char after 'X'.
pSt3:           cmpb (r0),#Y      ; continue printing until 'Y'.
                beq pSt4         ; if it reached 'Y' jump to pSt4.
                movb (r0)+,-(sp) ; else print the current letter.
                iot
                tst (sp)+
                br pSt3          ; continue to the next letter .

                ; print the number of targets instead of 'Y'.
                ; to do this we need to convert the number to string.
pSt4:           jsr pc,clrTmpStr  ; clear tmpStr to put the converted number in it.
                mov @#numOfTrgts,-(sp) ; pass the number to 'numToStr' by stack.
                mov #tmpStr,-(sp) ; pass the tmpStr to 'numToStr' by stack.
                jsr pc,numToStr   ; convert the number to string and return it into 'tmpStr'.

```

```

tst (sp)+          ; clear the stack from passed parameters.
tst (sp)+          ; clear the stack from passed parameters.
mov #tmpStr,-(sp)  ; print the number of covered targets to screen.
jsr pc, printStr
tst (sp)+
inc r0             ; point to next char after 'Y'. continue printing to the end by
mov r0,-(sp)       ; passing r0 to 'printStr'.
jsr pc, printStr
tst (sp)+          ; clear the stack from passed parameters.
mov (sp)+,r0       ; restore r0.
rts pc            ; return.

```

```

;-----
;~~~~~ printInterfaceData ~~~~~
; printInterfaceData:
; updates the the game data on the screen.
; it prints the remaining time and moves, then it prints the board (using 'printBoard' function).
; it is also responsible to convert remaining time from "clock ticks" unit to seconds uint.
; input: * remaining moves ( by stack).
;        * timeLeft, msgMovesLeft, msgTimeLeft, rate (globals)
; output: print to screen in the follow form:
;         Moves left: [remaining moves]
;         Seconds left: [remaining time in seconds]
;         [prints the board]
;
; usage:
;-----
; mov [remaining moves],-(sp)
; jsr pc,printInterfaceData
; tst (sp)+          ; clear the stack.
;-----

```

```

printInterfaceData:  mov #0, @#tks          ; disable reading and interrupts from keyboard.
                    jsr pc,newPage          ; start new page.
                    mov #msgMovesLeft,-(sp) ; print 'msgMovesLeft' message .
                    jsr pc, printStr
                    tst (sp)+
                    jsr pc,clrTmpStr        ; clear tmpStr.
                    mov 2(sp),-(sp)         ; 2(sp) contains the remaining moves.
                    mov #tmpStr,-(sp)       ; convert the remaining moves to string.
                    jsr pc,numToStr
                    tst (sp)+
                    tst (sp)+
                    mov #tmpStr,-(sp)       ; print the remaining moves.
                    jsr pc,printStr
                    tst (sp)+
                    jsr pc,printNewLine
                    mov #msgTimeLeft,-(sp)  ; print 'msgTimeLeft' message .
                    jsr pc, printStr
                    tst (sp)+

; convert remaining time from "clock ticks" unit to seconds uint.
mov r0,-(sp)        ; backup r0;
mov r1,-(sp)        ; backup r1;
mov @#timeLeft,r1   ; put remaining time ( "clock ticks" unit) in r1.
mov #0,r0            ; used by div, so div will proccess it and r1.
div @#rate,r0        ; devide remaining time by rate , so we get remaining seconds

```

```

inc r0                ; round up
mov r0,-(sp)          ; r0 contains the remaining time in seconds rounded up.
jsr pc,clrTmpStr      ; convert it to string to print it.
mov #tmpStr,-(sp)
jsr pc,numToStr
tst (sp)+
tst (sp)+
mov (sp)+,r1          ; restore r1.
mov (sp)+,r0          ; restore r0.
mov #tmpStr,-(sp)     ; print the remaining time in seconds.
jsr pc,printStr
tst (sp)+
jsr pc,printNewLine
jsr pc,printNewLine
jsr pc,printBoard     ; print the board.
jsr pc,printNewLine
mov #101, @#tks       ; re-enable reading and interrupts from keyboard.
rts pc

```

```

; -----
; ~~~~~~ printBoard ~~~~~~
; printBoard:
; prints the board of the game, it means to print the current player, walls, boxes and
; targets place on the screen.
; it use the defined symbols to print each element on the screen.
; input: nRows, nCols, Warehouse,Targets, PlayerRow, PlayerCol (globals)
; output: prints the board to the screen.
; usage:
; -----
; jsr pc,printBoard
; -----

```

```

printBoard:  mov r1,-(sp)      ; backup r1.
              mov #0,r1       ; r1 will used as row counter.
              jsr pc,printLimitWall ; print the the upper edge of the board.
              jsr pc,printNewLine

```

```

; print board lines , line by line.
pBloop:      cmp r1,@#nRows    ; if r1 = nRows this maens that all board row were printed,
              beq pBend        ; so jump to to pBend.
              mov #sWALL,-(sp) ; else print next board line, first print its left side edge.
              iot
              tst (sp)+
              mov r1,-(sp)     ; then pass the number of current row to 'printBline'
              jsr pc,printBLine ; to print the suitable Board Line.
              tst (sp)+
              mov #sWALL,-(sp) ; finally print this line side right edge.
              iot
              tst (sp)+
              jsr pc,printNewLine
              inc r1           ; move to the next row.
              br pBloop

```

```

pBend:       jsr pc,printLimitWall ; at the end print the the bottom edge of the board.
              mov (sp)+,r1        ; restore r1.
              rts pc              ; return.

```

```

; used to print upper and bottom edge of the board.
printLimitWall: mov r1,-(sp)      ; backup r1.
                 mov @#nCols,r1  ; put the number of columns of the board into r1,
                 add #2,r1       ; add 2 for two walls extra for sides.
pLWloop:  tst r1                  ; starting loop for printing suitable number of walls.
                 bne .+6
                 mov (sp)+,r1     ; when finish, restore r1,
                 rts pc           ; return.
                 mov #sWALL,-(sp) ; print one wall.
                 iot
                 tst (sp)+
                 dec r1           ; decrease the counter and
                 br pLWloop       ; continue..

; printBoardLine
printBLine: mov r0,-(sp)          ; backup r0.
             mov r1,-(sp)          ; backup r1.
             mov r2,-(sp)          ; backup r2.
             mov r3,-(sp)          ; backup r3.
             mov 12(sp),r0         ; r0 have the number of the row to print.

             tst -(sp)            ; will contain the result of getRltvPlace.
             mov #Warehouse,-(sp) ; ptr to Warehouse
             mov r0,-(sp)          ; r1= row
             mov #0,-(sp)         ; column=0
             jsr pc,getRltvPlace   ; call "getRltvPlace".
             add #6,sp            ; clear the stack from the parameters were passed.
             mov (sp)+,r1         ; put result into r1. then r1 now points to the first cell
                                 ; of the row, in Warehouse.

             mov r1,r2
             sub #Warehouse,r2
             add #Targets,r2      ; then r2 now point to facing cell on the Targets array.
             mov #0,r3            ; r3 will used as counter for the number of printed cells,
pBL1:      cmp r3,@#nCols         ; when it reaches the number of columns(nCols) means,
             bne pBL2            ; it means all the current line cell printed, so return. else
             mov (sp)+,r3         ; not finished, the jump to pBL2.
             mov (sp)+,r2         ; restore registers, and
             mov (sp)+,r1
             mov (sp)+,r0
             rts pc              ; return.

; pBL2 pass the suitable parameters for 'getSuitableSympol'
; that returns what symbol should be printed for the current cell,
; then pBL2 prints the symbol, and forewords the pointers, and
; returns to pBL1.
pBL2:      tst -(sp)             ; alloc place in stack to get the result.
             mov r1,-(sp)         ; pointer to Warehouse element.
             mov r2,-(sp)         ; pointer to Targets element.
             mov r3,-(sp)         ; column
             mov r0,-(sp)         ; row
             jsr pc,getSuitableSympol ; call 'getSuitableSympol'.
             add #10,sp          ; clear stack from passed parameters.
             iot                 ; print the result (symbol) that is in the stack.
             tst (sp)+           ; clear the stack.
             inc r1              ; forewords the pointers.

```

```

inc r2
inc r3
br pBL1          ; returns to pBL1.

```

```

; this function determines what symbol should be printed for a specific cell.
; it receives what is found at specific cell
; in Warehouse , and what in the facing cell in Targets
; the coodenates of this cell.

```

```

getSuitableSymbol:  cmp 2(sp),@#PlayerRow ; first it compare the coordinates with the player place
                   bne gss1
                   cmp 4(sp),@#PlayerCol
                   bne gss1
                   mov #sPLAYER,12(sp)    ; if they identical, the symbol should be printed for this cell
                   rts pc                  ; is the symbol of player. return it.
gss1:               cmpb @6(sp),#TARGET    ; else if it has no player check if it target cell,
                   bne gss2               ; if not , go to gss2, else if it is , make the default
                   mov #sTARGET,12(sp)    ; result to be symbol of target, but continue the check, so if it
                                           ; was also have a box then return , symbol of covered target.
                                           ; so lelated with warehouse contents.
gss2:               cmpb @10(sp),#WALL     ; it is wall , if yes return wall.
                   bne gss3
                   mov #sWALL,12(sp)
                   rts pc
gss3:               cmpb @10(sp),#EMPTY_CELL ; is it empty cell , if yes
                   bne gss4
                   cmpb @6(sp),#TARGET    ; check what is in Targets array, if target return target, else
                   beq gssEnd
                   mov #sEMPTY_CELL,12(sp) ; empty cell.
                   rts pc
gss4:               cmpb @10(sp),#BOX      ; if it have box without target,
                   bne gssEnd
                   cmpb @6(sp),#TARGET
                   beq gss5
                   mov #sBOX,12(sp)       ; return box.
                   rts pc
gss5:               mov #sBOX_ON_TARGET,12(sp) ; else (box + target) , return coverd target.
gssEnd:             rts pc

```

```

; -----

```

```

;-----
;~~~~~ isCorrect ~~~~~
; isCorrect:
; gets a letter , checks it , if it is one of the defined game letters (l,r,u,d,L,R,U,D) returns 1.
; if it is not one of them returns 0.
; it uses "whichDirection" function to determine if it correct or not .
; input: letter by register r0
; output: result by the stack.
; it have sided symptoms, r0 will be influenced by "whichDirection" and have its result.
; usage:
;-----
; [register r0 has a letter]
; tst -(sp) ; allocate place in the stack to return result.
; jsr pc,isCorrect
; cmp #1,(sp)+ ; test the result. 1 = correct , else = not correct.
;-----

isCorrect:    jsr pc,whichDirection ; call "whichDirection".
              cmp #0,r0             ; is the result 0?
              bne correct           ; if it is this mean not a correct letter.
              mov #0,2(sp)          ; when program get here, then it is not known side, so don't return 1.
              rts pc                ; return.
correct:      mov #1,2(sp)          ; it is correct letter, return 1 by using stack.
              rts pc                ; return.

;-----
;~~~~~ doStep ~~~~~
; doStep:
; this function identifies the direction of the move that should be done by call whichDirection, then
; calling the suitable procedure to implement the move (of the player and the boxes as well).
; this procedures are: lift, right , up , down
; if the step is illegal it doesn't change thing.
; input:      -direction letter using r0 register
;             - Warehouse, PlayerRow,PlayerCol, nCols, nRow (globals)
; output: * if step is legal : editing Warehouse,PlayerCol, PlayerRow
;         * legal or not (by stack)
; usage:
;-----
; [register r0 has a letter]
; tst -(sp) ; allocate place in the stack to return if step was legal or not.
; jsr pc,doStep
; cmp #1,(sp)+ ; test the result. if it 1 then the step was legal , else it was illegal step.
;-----

doStep:      mov r1,-(sp) ; backup r1

              ; these will be used in case was legal step ( else they have no influence):
              mov r2,-(sp) ; backup r2 . save new plr row.
              mov r3,-(sp) ; backup r3 . save new plr column.
              mov r4,-(sp) ; backup r4. will save player new place. so put in it 0.
                          ; (to remove box from it if it have)
              mov r5,-(sp) ; backup r5. will save box new place. so put in it #.
              mov #0,r5    ; (if there was no box move will be 0)
              mov @#PlayerRow,r2 ; init it.
              mov @#PlayerCol,r3 ; init it.

```

jsr pc,whichDirection ; result in capital letter returned to r0.

; the following subroutine simulate what will happen if player moved left ,
; if the step was legal , it jumps to legal , where the move done in real by the saved values
; from this subroutine. (AWESOME *,*)

```
left: cmpb #'L',r0      ;check if the letter is 'L'.
      bne right      ; if not, try right.
      mov PlayerCol,r1 ; move by value PlayerCol to r1.
      sub #1,r1       ; move(imagenary) the player left by subtracting 1 from his column
      cmp nCols,r1    ; check if it is in the limits of the board.
      ble notLegal    ; if not , the it is illegal, go to notLegal.
      bcs notLegal

      mov r1,r3       ; save player new column.
      tst -(sp)       ; will contain the result (relative place)
      mov #Warehouse,-(sp)
      mov PlayerRow,-(sp)
      mov r1,-(sp)
      jsr pc,getRltvPlace ; get the player new relative place.
      add #6,sp       ; clear stack from passed parameters.
      mov (sp)+,r4    ; save player new relative place in r4.

      tstb (r4)       ; if r4's address content is 0 so the new place neither have box nor wall.
      beq legal      ; so jump to legal.
      cmpb (r4),#WALL ; else check if it have wall,
      beq notLegal   ; if yes, then not legal. if no , continue.
      cmpb (r4),#BOX ; check if it have box, if not then
      bne notLegal   ; return not legal because it has strange value.
      sub #1,r1      ; if yes, move(imagenary) the box left by subtracting 1 from his column
      cmp nCols,r1   ; check if it is in the limits of the board.
      ble notLegal   ;
      bcs notLegal

      tst -(sp)       ; will contain the result
      mov #Warehouse,-(sp)
      mov PlayerRow,-(sp)
      mov r1,-(sp)
      jsr pc,getRltvPlace ;get the box new relative place.
      add #6,sp       ; clear stack.
      mov (sp)+,r5    ; save player new relative place in r5.
      tstb (r5)       ; if r4's address content is 0 then the new place neither have box nor wall,
      bne notLegal    ; then it is legal, else not legal.
      br legal
```

; the following subroutine simulate what will happen if player moved right ,
; if the step was legal , it jumps to legal , where the move done in real by the saved values
; from this subroutine.

```
right: cmpb #'R',r0      ;check if the letter is 'R'.
       bne up          ; if not, try up.
       mov PlayerCol,r1 ;make r1 as apointer to PlayerCol.
       add #1,r1        ;move(imagenary) the player right by adding 1 to his column

       cmp nCols,r1    ; check if it is in the limits of the board.
       ble notLegal
       bcs notLegal
```

```

mov r1,r3          ; save player new column.
tst -(sp)          ; will contain the result (relative place)
mov #Warehouse,-(sp)
mov PlayerRow,-(sp)
mov r1,-(sp)
jsr pc,getRltvPlace ;get the player new relative place
add #6,sp          ; clear stack from passed parameters.
mov (sp)+,r4       ; save player new relative place in r4.

tstb (r4)          ; if r1 contain 0, the new place neither have box nor wall.
beq legal          ; so jump to legal.
cmpb (r4),#WALL    ; if new place is wall
beq notLegal       ; then it is not legal
cmpb (r4),#BOX     ; check if it have box,
bne notLegal
add #1,r1          ; if yes move(imagenary) the box right by adding 1 from his column
cmp nCols,r1       ; check if it is in the limits of the board.
ble notLegal
bcs notLegal

```

```

tst -(sp)          ; will contain the result
mov #Warehouse,-(sp)
mov PlayerRow,-(sp)
mov r1,-(sp)
jsr pc,getRltvPlace ;get the box new relative place
add #6,sp          ; clear stack from passed parameters.
mov (sp)+,r5       ; save box new relative place in r5.
tstb (r5)          ; if r1's content is 0 so the new place neither have box nor wall.
bne notLegal
br legal

```

```

; -----
notLegal: mov (sp)+,r5 ; restore r5
          mov (sp)+,r4 ; restore r4
          mov (sp)+,r3 ; restore r3
          mov (sp)+,r2 ; restore r2
          mov (sp)+,r1 ; restore r1
          mov #0,2(sp)
          rts pc

```

```

; if the simulation result was postive, apply its saved value on real.
legal:  mov r2,@#PlayerRow; update player row.
        mov r3,@#PlayerCol ; update player column.
        movb #EMPTY_CELL,(r4) ; sign the previous place of the player as empty cell.
        tst r5 ; check if there was moved box.
        beq leg2

```

```

leg2:  movb #BOX,(r5) ; if yes put it in the suitable place.
        mov (sp)+,r5 ; restore r5
        mov (sp)+,r4 ; restore r4
        mov (sp)+,r3 ; restore r3
        mov (sp)+,r2 ; restore r2
        mov (sp)+,r1 ; restore r1
        mov #1,2(sp)
        rts pc
; -----

```

; the following subroutine simulate what will happen if player moved up ,
 ; if the step was legal , it jumps to legal , where the move done in real by the saved values
 ; from this subroutine.

```
up:    cmpb #'U',r0      ;check if the letter is 'U'.
      bne down         ; if not, try down.
      mov PlayerRow,r1  ; move by value PlayerCol to r1.
      sub #1,r1         ; move the dummy player left by subtracting 1 from his column
      cmp nRows,r1      ; check if it is still in the limits of the board.
      ble notLegal
      bcs notLegal

      mov r1,r2         ; save player new row.
      tst -(sp)         ; will contain the result (relative place)
      mov #Warehouse,-(sp)
      mov r1,-(sp)
      mov PlayerCol,-(sp)
      jsr pc,getRltvPlace ;get the player new relative place in r1 by calling getPlace procedure
      add #6,sp         ; clear stack from passed parameters.
      mov (sp)+,r4      ; save player new relative place in r4.

      tstb (r4)         ; if r1's content is 0 so the new place neither have box nor wall.
      beq legal         ; so jump to legal.
      cmpb (r4),#WALL
      beq notLegal
      cmpb (r4),#BOX    ; check if it have box,
      bne notLegal
      sub #1,r1         ; if yes move(imagenary) the box left by subtracting 1 from his column
      cmp nRows,r1
      ble notLegal
      bcs notLegal

      tst -(sp)         ; will contain the result
      mov #Warehouse,-(sp)
      mov r1,-(sp)
      mov PlayerCol,-(sp)
      jsr pc,getRltvPlace ;get the player new relative place
      add #6,sp         ; clear stack from passed parameters.
      mov (sp)+,r5      ; save box new relative place in r4.
      tstb (r5)         ; if r1's content is 0 so the new place neither have box nor wall.
      bne notLegal
      br legal
```

; the following subroutine simulate what will happen if player moved down ,
 ; if the step was legal , it jumps to legal , where the move done in real by the saved values
 ; from this subroutine.

```
down:  cmpb #'D',r0      ; check if the letter is 'D'.
      bne notLegal      ; if not , it is not legal
      mov PlayerRow,r1  ; move by value PlayerCol to r1.
      add #1,r1         ; move the dummy player left by subtracting 1 from his column
      cmp nRows,r1      ; check if it is still in the limits of the board.
      ble notLegal
      bcs notLegal

      mov r1,r2         ; save player new row.
      tst -(sp)         ; will contain the result (relative place)
```

```

mov #Warehouse,-(sp)
mov r1,-(sp)
mov PlayerCol,-(sp)
jsr pc,getRltvPlace    ;get the player new relative place.
add #6,sp              ; clear stack from passed parameters.
mov (sp)+,r4           ; save player new relative place in r4.

tstb (r4)              ; if r1's content is 0 so the new place neither have box nor wall.
beq legal              ; so jump to legal.
cmpb (r4),#1
beq notLegal
cmpb (r4),#BOX         ; check if it have box,
bne notLegal
add #1,r1              ; if yes move(imagenary) the box left by subtracting 1 from his column
cmp nRows,r1           ; check if it is still in the limits of the board.
ble notLegal
bcs notLegal

tst -(sp)              ; will contain the result (relative place)
mov #Warehouse,-(sp)
mov r1,-(sp)
mov PlayerCol,-(sp)
jsr pc,getRltvPlace    ;get the box new relative place.
add #6,sp              ; clear stack from passed parameters.
mov (sp)+,r5           ; save player new relative place in r5.
tstb (r5)              ; if r1's content is 0 so the new place neither have box nor wall.
bne powerJump         ; will jump to notLegal.
br legal

```

```

powerJump: jmp notLegal ; Spinach meal for 'bne'.
; -----

```

```

;-----
; ~~~~~ whichDirection ~~~~~
; whichDirection:
; This function returns the suitable direction (in capital letter) of input letter.
; example: if letter is u/U it returns U
; if the letter is not suitable, it returns 0.
; this function uses Hassan's trick to convert the letter to capital letter.
; input: direction letter using r0 register.
; output: * the capital letter of the letter (by r0 register)
;         * 0 if the letter is not one of the defined game letters. (by r0 register)
; usage:
;-----
; [register r0 has a letter]
; jsr pc,whichDirection
; [register r0 updated]
;-----

whichDirection: bic #40,r0          ; Hassan's trick to convert the letter to capital letter.
                cmpb #'L',r0      ; check if the current letter in Moves array is 'L'.
                bne R             ; if not, go to check if it 'R'.
                rts pc            ; return.
R:              cmpb #'R',r0      ; check if the current letter in Moves array is 'R'.
                bne U             ; if not, go to check if it 'U'.
                rts pc            ; return.
U:              cmpb #'U',r0      ; check if the current letter in Moves array is 'U'.
                bne D             ; if not, go to check if it 'D'.
                rts pc            ; return.
D:              cmpb #'D',r0      ; check if the current letter in Moves array is 'D'.
                bne non           ; if not, then it is non of defined game letters.
                rts pc            ; return.
non:            mov #0,r0         ; if the program gets here, then it is not known side, so return 0.
                rts pc            ; return.
;-----
; ~~~~~ number <-> string ~~~~~
; strToNum:
; converts string number to normal number.
; input: string address (by stack)
; output: the number (by stack)
; usage:
;-----
; tst -(sp)      ; allocate place in the stack, strToNum will put into it the result.
; mov [string address],-(sp)
; jsr pc,strToNum
; tst (sp)+      ; clear the stack from passed parameters.
; mov (sp)+,[result]
;-----

strToNum:       mov r0,-(sp)      ; backup r0.
                mov r1,-(sp)      ; backup r1.
                mov r2,-(sp)      ; backup r2.
                mov r3,-(sp)      ; backup r3.
                mov #0,r2         ; will contain the result.
                mov 12(sp),r0     ; make r0 pointer to the string.
                mov #0,r3         ; will be used to calculate the length.
sTN1:           tstb (r0)+
                beq sTN2

```

```

        inc r3          ; length++.
        br sTN1
sTN2:   mov 12(sp),r0
sTN3:   tstb (r0)
        beq sTN6
        movb (r0)+,r1
        bic #177760,r1    ; get its numeric value.
        mov r3,-(sp)
        dec r3          ; to get 100 you mul by 10 2 times not 3
sTN4:   tst r3
        beq sTN5
        mul #10.,r1    ; so get its value by multipling it by its weight.
        dec r3
        br sTN4
sTN5:   add r1,r2
        mov (sp)+,r3
        dec r3
        br sTN3
sTN6:   mov r2,14(sp) ; return the result by stack.
        mov (sp)+,r3 ; restore r3.
        mov (sp)+,r2 ; restore r2.
        mov (sp)+,r1 ; restore r1.
        mov (sp)+,r0 ; restore r0.
        rts pc
; -----
; numToStr:
; puts number into string.
; input:      * number (by stack)
;            * target string address ( by stack)
; output : editing the string.
; usage:
; -----
; mov [number],-(sp)
; mov [string address],-(sp)
; jsr pc,numToStr
; tst (sp)+      ; clear the stack from passed parameters.
; tst (sp)+      ; clear the stack from passed parameters.
; -----

numToStr:      mov r0,-(sp)  ; backup r0.
               mov r1,-(sp) ; backup r1.
               mov r2,-(sp) ; backup r2.
               mov #0,r0    ; will used by div.
               mov 12(sp),r1 ; the number
               mov 10(sp),r2 ; pointer to the output string.
               br nTS1
nTS0:          tst r1
               beq nTS2
nTS1:          div #10.,r0   ; every loop divide the number by 10, take the remaint
               add #0,r1    ; convert it to its ascii value
               movb r1,(r2)+ ; save it to array
               sub #0,r1
               mov r0,r1
               mov #0,r0
               br nTS0
nTS2:          mov 10(sp),r1

```

```

nTS3:      decb r2
           cmp r1,r2          ; finally the array should be to oppsite direction
           bge nTS4           ; so do on it swap.
           movb (r1),r0
           movb (r2),(r1)
           movb r0,(r2)
           incb r1
           decb r2
           br nTS3

nTS4:      mov (sp)+,r2 ; restore r2.
           mov (sp)+,r1 ; restore r1.
           mov (sp)+,r0 ; restore r0.
           rts pc

```

```

; -----
; clrTmpStr:
; fills tmpStr with 0s, so it clear it from previous uses.
; input: tmpStr ( global)
; usage:
; -----
; jsr pc,clrTmpStr
; -----

```

```

clrTmpStr:  mov r0,-(sp) ; backup r0.
           mov r1,-(sp) ; backup r1.
           mov #tmpStr,r0 ; make r0 pointer to tmpStr.
           mov #STR_MAX_LEN+2,r1 ; r1 contians the len of tmpStr, will used as a counter.

cts1:      tst r1 ; check the counter.
           bne .+10
           mov (sp)+,r1 ; restore r1.
           mov (sp)+,r0 ; restore r0.
           rts pc
           movb #0,(r0)+ ; fill current byte with 0, foreward the pointer.
           dec r1 ; decrease the counter.
           br cts1 ; continue.

```

```

; -----
timeLeft:  .even
           .word 7500. ; the remaining time in clock ticks units.
maxMoves:  .word 10 ; the max moves player can do. its value will be gotten from the user.
numOfTrgts: .word 0 ; number of target places in the game. will be
                  ; calculated by "updateNumOfCovrd".
trgtsCovered: .word 0 ; the number of target places on the board, that coverd with box. it is
                  ; updated after every player legal move by "updateNumOfCovrd".
cursorStatus: .word 0 ; value in clock ticks units, to determine to show the cursor, or not.
finishWait:  .word 0 ; when it 1 , it means don't wait anymore for input.
finishProg:  .word 0 ; when it 1 , it means that the program done its job.
strLen:      .word 0 ; used to calc str len by interrupt handlers.

```

; Messages that will be printed to the user:

```

msgEntrName: .ascii<Enter the name of the player:>
             .byte 0 ; end of string.
msgMaxTime:  .ascii<Enter the maximum number of seconds allowed:>
             .byte 0
msgMaxMoves: .ascii<Enter the maximum number of moves allowed:>
             .byte 0
msgPressKey: .ascii<, press any key to start the game...>
             .byte 0

```

```
msgMovesLeft:      .ascii<Moves left: >  
                   .byte 0  
msgTimeLeft:       .ascii<Seconds left: >  
                   .byte 0  
name:              .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  
msgYouWon:         .ascii<, you won!>  
                   .byte 0  
timeExcded:        .ascii<, sorry, allowed time exceeded, the game is over.>  
                   .byte 0  
mvsExcded:         .ascii<, sorry, allowed moves exceeded, the game is over.>  
                   .byte 0  
msgStatistics:     .ascii<Only X out of Y targets are covered.>  
                   .byte 0  
tmpStr:            .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; this string is for general use, espicialy  
                                                           ; to convert numbers to strings to print them.  
                   .even    ; if someone add word won't be errors  
;
```