# Chapter 9.2: The emu8086.inc Library

## Microprocessors

The INCLUDE directive · Macros · PUTC · GOTOXY · PRINT · PRINTN · Cursor macros · Procedures · Reading / printing numbers · PTHIS and the stack · Example programs

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/microprocessors/slides/Chapter_09_02_The_emu8086_inc_Library.html

# Learning Outcomes

**INCLUDE**

Explain how a library file is pulled into the source and where the assembler looks for it.

**Macros**

Show what macros such as `PUTC`, `GOTOXY`, `PRINT` and `PRINTN` expand to.

**Procedures**

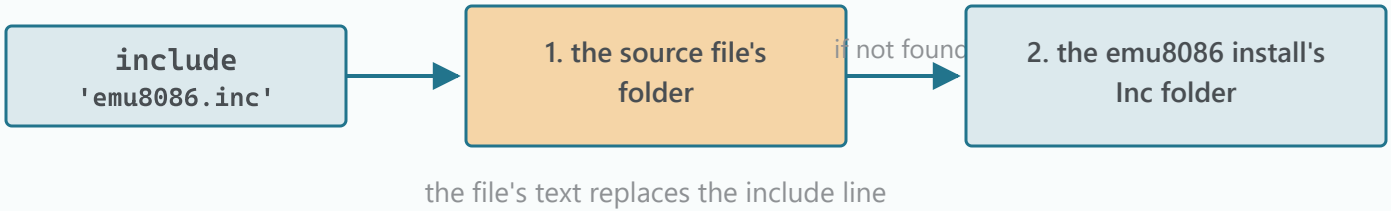
Use the ready-made procedures (number input/output, string printing) with `DEFINE_…` and `CALL`.

**Example programs**

Write programs that read numbers, compute with them and lay out the screen using library macros and procedures.

# A Library of Common Functions

- Frequently used routines come ready-made, so you don't have to write them yourself.
- Routines defined in other files are brought in with the **INCLUDE** directive.
- To use anything in `emu8086.inc`, put `include 'emu8086.inc'` at the top of the source file.
- The library covers the screen, keyboard and number routines most emu8086 programs need.




INCLUDE is a *directive*: it generates no machine code, it just inserts text at assembly time.

# The Source After INCLUDE

```
; in place of the include line:
PUTC MACRO char      ; template: 0 bytes
  ...
ENDM
GOTOXY MACRO col, row
  ...
ENDM
DEFINE_SCAN_NUM MACRO
  ...                ; procedure template
ENDM
; the program itself:
ORG  100h
PUTC 'A'             ; 8 bytes here
RET
```

- The assembler replaces the `include` line with the file's text, then assembles the result as one source.
- A macro definition is only a **template**: an unused macro adds no bytes at all.
- Code appears **where the macro is used**: `PUTC 'A'` expands into 8 bytes of instructions.
- Procedures are wrapped in macros too (`DEFINE_…`): without its DEFINE_ line, a procedure never enters the program.
- Including the whole library costs nothing: program size depends only on the macros you use and the procedures you define.

At run time there is no “emu8086.inc”: memory holds only expanded instructions and the procedures you defined.

# What's in the Library?

### Macros · used directly

| Macro | Purpose |
|---|---|
| `PUTC char` | print one character |
| `GOTOXY col, row` | position the cursor |
| `PRINT string` | print a string |
| `PRINTN string` | print a string + newline |
| `CURSOROFF` | hide the cursor |
| `CURSORON` | show the cursor |

### Procedures · DEFINE_ + CALL

| Procedure | Purpose |
|---|---|
| `PRINT_STRING` | print the string at DS:SI |
| `PTHIS` | print the string after the CALL |
| `GET_STRING` | read a string from the keyboard |
| `CLEAR_SCREEN` | clear the screen |
| `SCAN_NUM` | read a number → CX |
| `PRINT_NUM` | print AX as signed |
| `PRINT_NUM_UNS` | print AX as unsigned |

Macros are **copied** to every place they are used; procedures are defined once and **called** with `CALL`.

# Screen Macros

| Macro | What it does | Under the hood |
|---|---|---|
| `PUTC char` | Prints one ASCII character at the cursor. | `INT 10h`, AH = 0Eh |
| `GOTOXY col, row` | Moves the cursor to the given column and row. | `INT 10h`, AH = 02h |
| `PRINT string` | Prints the given string. | 0Eh, one character at a time |
| `PRINTN string` | Like PRINT, then adds CR + LF (13, 10) to start a new line. | 0Eh, one character at a time |
| `CURSOROFF` | Turns the text cursor off (hides it). | `INT 10h`, AH = 01h |
| `CURSORON` | Turns the text cursor back on. | `INT 10h`, AH = 01h |

Macro arguments can be constants, registers or variables: `PUTC 65`, `PUTC 'B'`, `PUTC AL`.

# What Do PUTC and GOTOXY Expand To?

## PUTC 'A'

```
PUSH AX         ; save AX
MOV  AL, 'A'    ; argument
MOV  AH, 0Eh    ; teletype
INT  10h
POP  AX         ; restore AX
```

5 instructions, **8 bytes**. Registers are preserved; the calling code is unaffected.

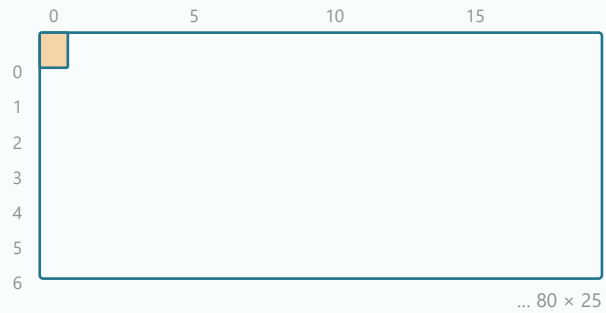
## GOTOXY 10, 5

```
PUSH AX
PUSH BX
PUSH DX
MOV  AH, 02h    ; set cursor
MOV  DH, 5      ; row
MOV  DL, 10     ; column
MOV  BH, 0      ; page 0
INT  10h
POP  DX
POP  BX
POP  AX
```

The argument order is **column, row** (x, y); the BIOS takes the column in **DL** and the row in **DH**. Top left is (0, 0), bottom right (79, 24).

# Code Trace: PRINT, GOTOXY, PUTC

```
▶ include emu8086.inc
2 ORG    100h
3 PRINT 'Hello World!'
4 GOTOXY 10, 5
5 PUTC 65           ; 65 - is an ASCII code for 'A'
6 PUTC 'B'
7 RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`include` : macro definitions are inserted at assembly time; no code is generated.

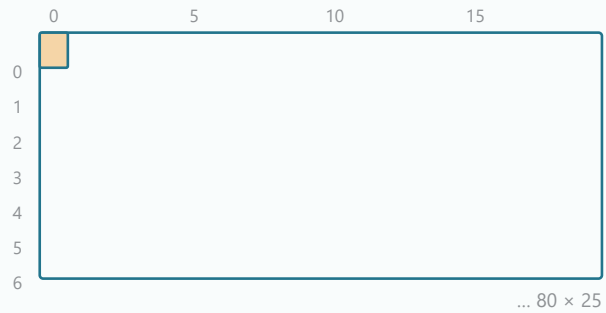
Step 1 / 7

**VARIABLES**

`col = 0` `row = 0`

# Code Trace: PRINT, GOTOXY, PUTC

```
1 include emu8086.inc
2 ORG    100h
3 PRINT 'Hello World!'
4 GOTOXY 10, 5
5 PUTC 65           ; 65 - is an ASCII code for 'A'
6 PUTC 'B'
7 RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`ORG 100h` : a COM program starts at 100h.

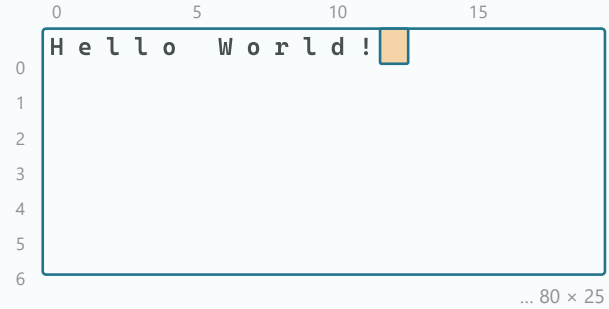
Step 2 / 7

**VARIABLES**

`col = 0` `row = 0`

# Code Trace: PRINT, GOTOXY, PUTC

```
1 include emu8086.inc
2 ORG    100h
▶ PRINT 'Hello World!'
4 GOTOXY 10, 5
5 PUTC 65           ; 65 - is an ASCII code for 'A'
6 PUTC 'B'
7 RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`PRINT` : 12 characters printed (columns 0–11); the cursor is at column 12.

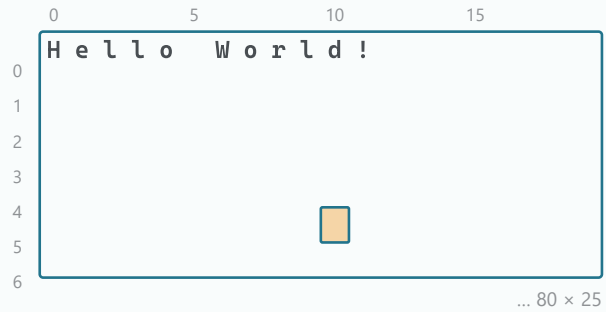
Step 3 / 7

VARIABLES

`col = 12` `row = 0`

# Code Trace: PRINT, GOTOXY, PUTC

```
1 include emu8086.inc
2 ORG    100h
3 PRINT 'Hello World!'
▶ GOTOXY 10, 5
5 PUTC 65           ; 65 - is an ASCII code for 'A'
6 PUTC 'B'
7 RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`GOTOXY 10, 5` : the cursor moves to column 10, row 5. Nothing is printed.

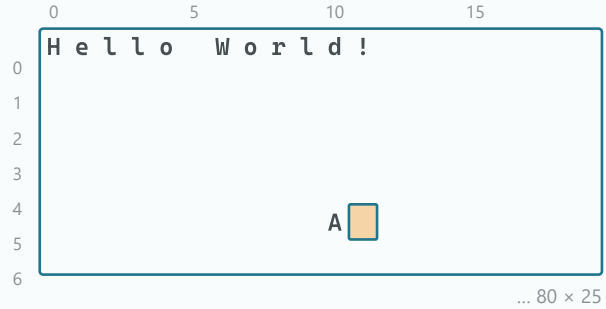
Step 4 / 7

**VARIABLES**

`col = 10` `row = 5`

# Code Trace: PRINT, GOTOXY, PUTC

```
1 include emu8086.inc
2 ORG    100h
3 PRINT 'Hello World!'
4 GOTOXY 10, 5
▶ PUTC 65           ; 65 - is an ASCII code for 'A'
6 PUTC 'B'
7 RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`PUTC 65` : 65 = 41h = 'A' is printed; the cursor moves to column 11.

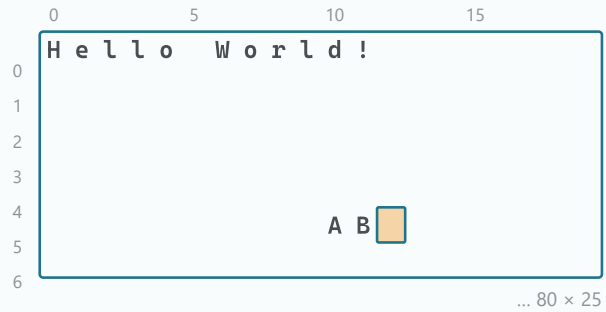
Step 5 / 7

**VARIABLES**

`col = 11` `row = 5`

# Code Trace: PRINT, GOTOXY, PUTC

```
1 include emu8086.inc
2 ORG    100h
3 PRINT 'Hello World!'
4 GOTOXY 10, 5
5 PUTC 65           ; 65 - is an ASCII code for 'A'
▶ PUTC 'B'
7 RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`PUTC 'B'` : a character literal works too.

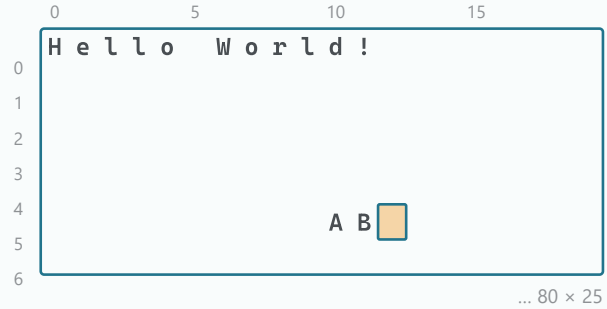
Step 6 / 7

**VARIABLES**

`col = 12` `row = 5`

# Code Trace: PRINT, GOTOXY, PUTC

```
1 include emu8086.inc
2 ORG    100h
3 PRINT 'Hello World!'
4 GOTOXY 10, 5
5 PUTC 65           ; 65 - is an ASCII code for 'A'
6 PUTC 'B'
▶ RET               ; return to operating system.
8 END               ; directive to stop the compiler.
```




`RET` : return to DOS. `END` just tells the assembler the source ends here.

Step 7 / 7

**VARIABLES**

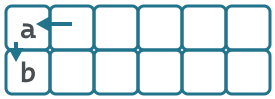
`col = 12` `row = 5`

# PRINTN and the Cursor Macros

### PRINT vs. PRINTN

`PRINT 'a'` then `PRINT 'b'` → `ab` on one line.
`PRINTN 'a'` then `PRINT 'b'` → `a` and, on the next line, `b`.

**CR** (13) moves the cursor to the start of the line, **LF** (10) moves it down a line; a newline needs both.




CR (13): line start

LF (10): next line

### CURSOROFF / CURSORON

They use `INT 10h`, AH = 01h (cursor shape): CH = start, CL = end scan line. Setting bit 5 of CH **hides** the cursor; CURSORON restores the normal shape.

Handy for hiding the blinking cursor on game or menu screens.

```
PRINTN 'Full name:'  ; print, then newline
CURSOROFF
GOTOXY 30, 12
PRINT 'Hello'
CURSORON
```

# Control Characters with PUTC

| Code | Name | PUTC effect |
|---|---|---|
| `07h` | BEL | A short beep from the speaker; the cursor doesn't move. |
| `08h` | BS (backspace) | Cursor steps left; it does **not erase** the character. No effect in column 0. |
| `0Ah` | LF (line feed) | Cursor moves down a row; **the column stays**. On the last row the screen scrolls up one line. |
| `0Dh` | CR (carriage return) | Cursor returns to column 0 of the same row; the text stays put. |

### Newline = CR + LF

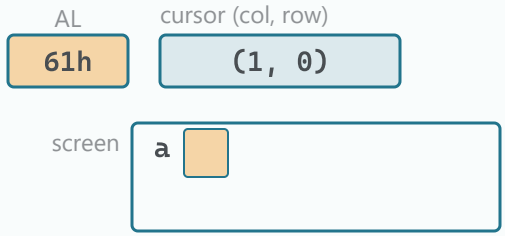
The `13, 10` pair that PRINTN appends combines both moves: line start, then down a line. In reverse order (10, 13) the cursor ends up in the same place.

### Erasing on screen

To erase a character, use the triple `PUTC 8`, `PUTC ' '`, `PUTC 8`: step left, overwrite with a space, step left again. The library procedures that handle backspace fix the screen the same way.

# Code Trace: CR and LF Separately

```
1  ORG  100h
2  MOV  AH, 0Eh      ; teletype
3  MOV  AL, 'a'
4  INT  10h
5  MOV  AL, 'b'
6  INT  10h
7  MOV  AL, 10       ; LF only
8  INT  10h
9  MOV  AL, 'c'
10 INT  10h
11 MOV  AL, 13       ; CR only
12 INT  10h
13 MOV  AL, 'd'
14 INT  10h
15 RET
```




AH = 0Eh is loaded only once here; `INT 10h` prints the 'a' in AL and moves the cursor right to (1, 0).

Step 1 / 6

**VARIABLES**

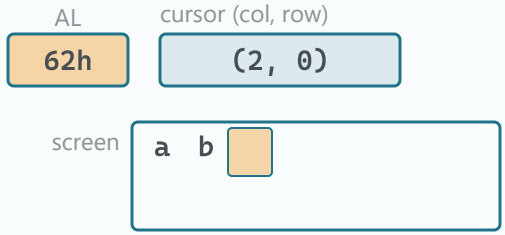
`AL = 61h` `cursor = (1, 0)`

**OUTPUT**

```
a
```

# Code Trace: CR and LF Separately

```
1  ORG  100h
2  MOV  AH, 0Eh      ; teletype
3  MOV  AL, 'a'
4  INT  10h
▶  MOV  AL, 'b'
▶  INT  10h
7  MOV  AL, 10       ; LF only
8  INT  10h
9  MOV  AL, 'c'
10 INT  10h
11 MOV  AL, 13       ; CR only
12 INT  10h
13 MOV  AL, 'd'
14 INT  10h
15 RET
```




'b' is printed; cursor at (2, 0).

Step 2 / 6

**VARIABLES**

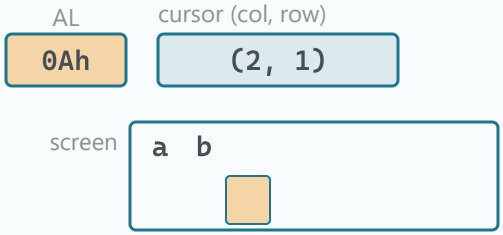
`AL = 62h` `cursor = (2, 0)`

**OUTPUT**

```
ab
```

# Code Trace: CR and LF Separately

```
1  ORG  100h
2  MOV  AH, 0Eh      ; teletype
3  MOV  AL, 'a'
4  INT  10h
5  MOV  AL, 'b'
6  INT  10h
▶  MOV  AL, 10       ; LF only
▶  INT  10h
9  MOV  AL, 'c'
10 INT  10h
11 MOV  AL, 13       ; CR only
12 INT  10h
13 MOV  AL, 'd'
14 INT  10h
15 RET
```




**LF alone** (10): the cursor drops one row but stays in column 2: (2, 1).

Step 3 / 6

VARIABLES

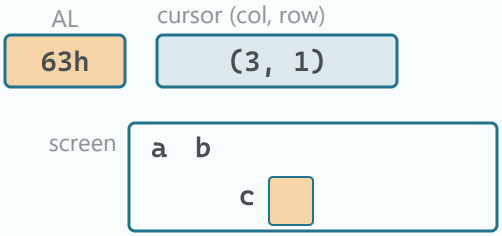
AL = 0Ah  cursor = (2, 1)

OUTPUT

```
ab
```

# Code Trace: CR and LF Separately

```
1  ORG  100h
2  MOV  AH, 0Eh      ; teletype
3  MOV  AL, 'a'
4  INT  10h
5  MOV  AL, 'b'
6  INT  10h
7  MOV  AL, 10       ; LF only
8  INT  10h
▶  MOV  AL, 'c'
▶  INT  10h
11 MOV  AL, 13       ; CR only
12 INT  10h
13 MOV  AL, 'd'
14 INT  10h
15 RET
```




'c' lands in column 2 of the next row: the text “staircases”.

Step 4 / 6

**VARIABLES**

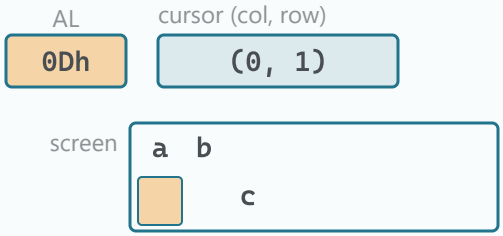
`AL = 63h` `cursor = (3, 1)`

**OUTPUT**

```
ab
  c
```

# Code Trace: CR and LF Separately

```
1  ORG  100h
2  MOV  AH, 0Eh      ; teletype
3  MOV  AL, 'a'
4  INT  10h
5  MOV  AL, 'b'
6  INT  10h
7  MOV  AL, 10       ; LF only
8  INT  10h
9  MOV  AL, 'c'
10 INT  10h
▶ MOV  AL, 13       ; CR only
▶ INT  10h
13 MOV  AL, 'd'
14 INT  10h
15 RET
```




**CR alone** (13): the cursor returns to the start of the same row, (0, 1). The c is not erased.

Step 5 / 6

VARIABLES

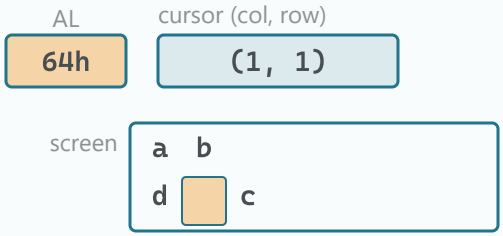
AL = 0Dh  cursor = (0, 1)

OUTPUT

```
ab
  c
```

# Code Trace: CR and LF Separately

```
1  ORG  100h
2  MOV  AH, 0Eh      ; teletype
3  MOV  AL, 'a'
4  INT  10h
5  MOV  AL, 'b'
6  INT  10h
7  MOV  AL, 10       ; LF only
8  INT  10h
9  MOV  AL, 'c'
10 INT  10h
11 MOV  AL, 13       ; CR only
12 INT  10h
▶ MOV  AL, 'd'
▶ INT  10h
15 RET
```




'd' goes into column 0. The screen shows “ab” with “d c” below. A proper newline needs both 13 and 10.

Step 6 / 6

**VARIABLES**

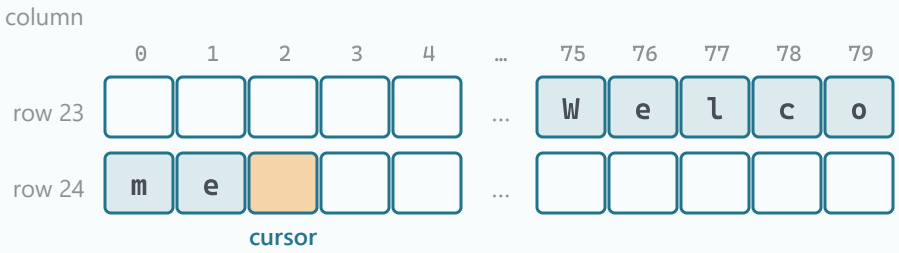
AL = 64h  cursor = (1, 1)

**OUTPUT**

```
ab
d c
```

# Line Wrap and Scrolling

- After writing column 79, service 0Eh moves the cursor to the start of the next row by itself, so PRINT continues a long string without a break.
- If the cursor has to move down from the last row (24), the whole screen **scrolls up one row**: the top row disappears and a blank row appears at the bottom.
- Example: `GOTOXY 75, 24` then `PRINT 'Welcome'`. "Welco" fills columns 75–79; for "me" the screen scrolls. In the end "Welco" is on row 23, "me" at the start of row 24, and the cursor at (2, 24).
- GOTOXY and PUTC do no bounds checking: keep positions within 0–79 (column) and 0–24 (row).



# Demo: The GOTOXY Screen

Click a cell on the 80 × 25 text screen to see the macro that moves the cursor there, the BIOS registers it sets and the matching video memory address.

PUTC character A clear

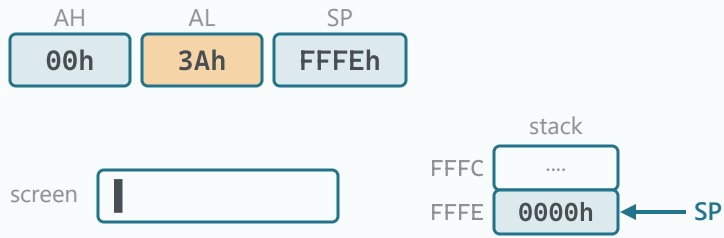


**GOTOXY 0, 0** → MOV DL,0 · MOV DH,0 · AH=02h · INT 10h

video memory: B800:0000h (character), B800:0001h (attribute)

# Code Trace: Two Digits with AAM and PUTC

```
1  ORG  100h
▶  MOV  AL, 58
3  AAM              ; AH = 5, AL = 8
4  ADD  AX, 3030h   ; '5', '8'
5  PUSH AX          ; PUTC AH expanded
6  MOV  AL, AH
7  MOV  AH, 0Eh
8  INT  10h
9  POP  AX          ; AX back: 3538h
10 PUSH AX          ; PUTC AL expanded
11 MOV  AL, AL
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 RET
```




AL = 58 (3Ah). Goal: print it as two decimal digits.

Step 1 / 7

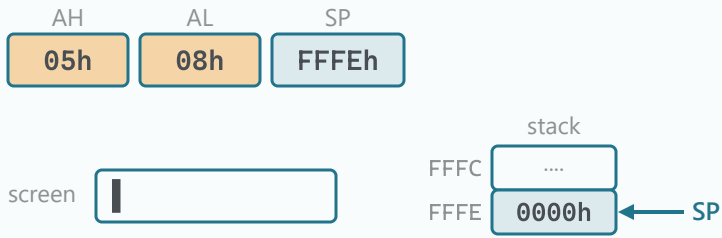
**VARIABLES**

AH = 00h  AL = 3Ah  SP = FFFEh

**OUTPUT**

# Code Trace: Two Digits with AAM and PUTC

```
1  ORG  100h
2  MOV  AL, 58
▶  AAM              ; AH = 5, AL = 8
4  ADD  AX, 3030h   ; '5', '8'
5  PUSH AX          ; PUTC AH expanded
6  MOV  AL, AH
7  MOV  AH, 0Eh
8  INT  10h
9  POP  AX          ; AX back: 3538h
10 PUSH AX          ; PUTC AL expanded
11 MOV  AL, AL
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 RET
```




`AAM` divides AL by 10: AH = 5 (tens), AL = 8 (ones). Only AL is divided, so the number must be 0–99.

Step 2 / 7

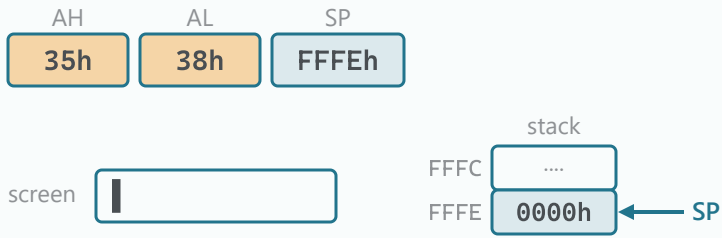
**VARIABLES**

`AH = 05h` `AL = 08h` `SP = FFFEh`

**OUTPUT**

# Code Trace: Two Digits with AAM and PUTC

```
1  ORG  100h
2  MOV  AL, 58
3  AAM              ; AH = 5, AL = 8
▶  ADD  AX, 3030h   ; '5', '8'
5  PUSH AX          ; PUTC AH expanded
6  MOV  AL, AH
7  MOV  AH, 0Eh
8  INT  10h
9  POP  AX          ; AX back: 3538h
10 PUSH AX          ; PUTC AL expanded
11 MOV  AL, AL
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 RET
```




`ADD AX, 3030h` adds 30h to both bytes at once: AH = '5' (35h), AL = '8' (38h).

Step 3 / 7

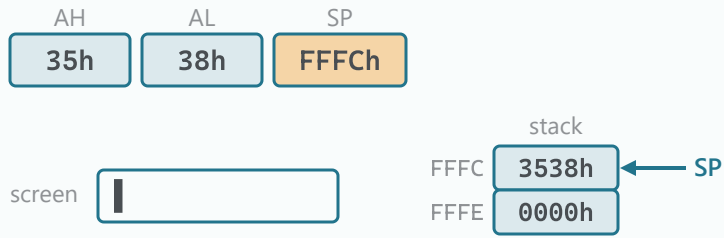
**VARIABLES**

`AH = 35h` `AL = 38h` `SP = FFFEh`

**OUTPUT**

# Code Trace: Two Digits with AAM and PUTC

```
 1 ORG  100h
 2 MOV  AL, 58
 3 AAM              ; AH = 5, AL = 8
 4 ADD  AX, 3030h   ; '5', '8'
 5 PUSH AX          ; PUTC AH expanded
 6 MOV  AL, AH
 7 MOV  AH, 0Eh
 8 INT  10h
 9 POP  AX          ; AX back: 3538h
10 PUSH AX          ; PUTC AL expanded
11 MOV  AL, AL
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 RET
```




`PUTC AH` expansion: first AX (3538h) is pushed, SP = FFFCh.

Step 4 / 7

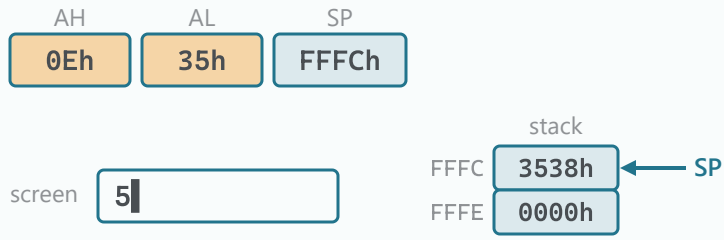
**VARIABLES**

`AH = 35h` `AL = 38h` `SP = FFFCh`

**OUTPUT**

# Code Trace: Two Digits with AAM and PUTC

```
1  ORG  100h
2  MOV  AL, 58
3  AAM              ; AH = 5, AL = 8
4  ADD  AX, 3030h   ; '5', '8'
5  PUSH AX          ; PUTC AH expanded
▶  MOV  AL, AH
▶  MOV  AH, 0Eh
▶  INT  10h
9  POP  AX          ; AX back: 3538h
10 PUSH AX          ; PUTC AL expanded
11 MOV  AL, AL
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 RET
```




`MOV AL, AH` and `MOV AH, 0Eh`: AX = 0E35h; the ones digit is gone from the register. `INT 10h` prints 5.

Step 5 / 7

**VARIABLES**

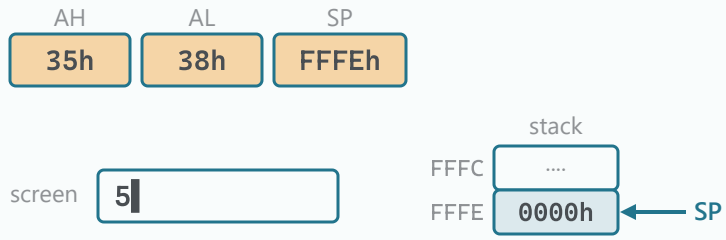
`AH = 0Eh` `AL = 35h` `SP = FFFCh`

**OUTPUT**

```
5
```

# Code Trace: Two Digits with AAM and PUTC

```
1  ORG  100h
2  MOV  AL, 58
3  AAM              ; AH = 5, AL = 8
4  ADD  AX, 3030h   ; '5', '8'
5  PUSH AX          ; PUTC AH expanded
6  MOV  AL, AH
7  MOV  AH, 0Eh
8  INT  10h
▶ POP  AX           ; AX back: 3538h
10 PUSH AX          ; PUTC AL expanded
11 MOV  AL, AL
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 RET
```




`POP AX` : AX = 3538h is back; '8' is in AL again.

Step 6 / 7

**VARIABLES**

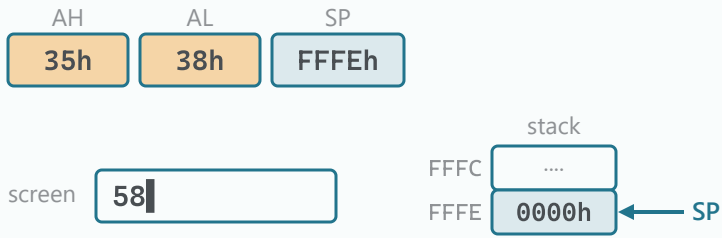
`AH = 35h` `AL = 38h` `SP = FFFEh`

**OUTPUT**

```
5
```

# Code Trace: Two Digits with AAM and PUTC

```
1  ORG  100h
2  MOV  AL, 58
3  AAM              ; AH = 5, AL = 8
4  ADD  AX, 3030h   ; '5', '8'
5  PUSH AX          ; PUTC AH expanded
6  MOV  AL, AH
7  MOV  AH, 0Eh
8  INT  10h
9  POP  AX          ; AX back: 3538h
▶ PUSH AX          ; PUTC AL expanded
▶ MOV  AL, AL
▶ MOV  AH, 0Eh
▶ INT  10h
▶ POP  AX
15 RET
```




`PUTC AL` expands the same way (`MOV AL, AL` is harmless). The screen shows 58; AX is still 3538h. Because PUTC preserves AX, both digits print without copying them to another register.

Step 7 / 7

**VARIABLES**

`AH = 35h` `AL = 38h` `SP = FFFEh`

**OUTPUT**

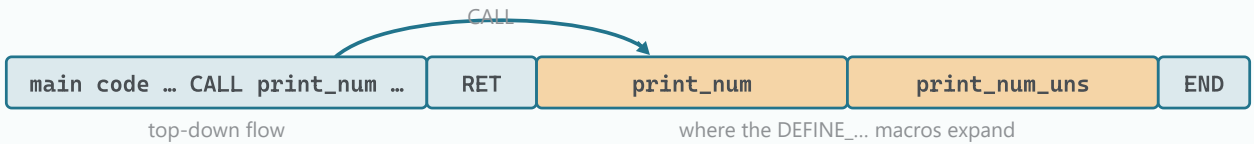
```
58
```

# Using Procedures: DEFINE_ and CALL

```
include 'emu8086.inc'
ORG 100h
    ...
    CALL print_num    ; use
    ...
RET

DEFINE_PRINT_NUM     ; define
DEFINE_PRINT_NUM_UNS
END
```

- Each procedure must be defined **once**, before the `END` directive, with its own `DEFINE_…` macro.
- A `DEFINE_…` macro expands the procedure's code right where it appears.
- Definitions usually go between `RET` and `END`. Every DEFINE_ macro starts with a `JMP` over the body, so execution never falls into the procedure.
- In the code you `CALL` a procedure by name, written in lower case ( `print_num` ).
- `PRINT_NUM` calls `PRINT_NUM_UNS` internally, so both must be defined.



# Input and Output Registers

| Procedure | Input | Output | Notes |
|---|---|---|---|
| `PRINT_STRING` | DS:SI = string address | — | string must end in 0 |
| `PTHIS` | string right after the CALL | — | advances the return address itself |
| `GET_STRING` | DS:DI = buffer, DX = size | 0-terminated string in the buffer | at most DX − 1 characters |
| `CLEAR_SCREEN` | — | — | cursor goes to (0, 0) |
| `SCAN_NUM` | keyboard | **CX** (signed) | old value of CX is lost |
| `PRINT_NUM` | AX (signed) | — | PRINT_NUM_UNS must be defined too |
| `PRINT_NUM_UNS` | AX (unsigned) | — | 0 … 65,535 |

Library procedures push every other register they use on entry and restore it on exit; only the output register changes. A program that keeps a loop counter in CX must save it before calling SCAN_NUM ( `PUSH CX` … `POP CX` ).

# String Procedures

### PRINT_STRING

- Prints a null-terminated (0) string at the cursor.
- Takes the string address from **DS:SI**.
- Defined by: `DEFINE_PRINT_STRING`.

```
LEA  SI, msg
CALL print_string
msg DB 'Hello', 0
```

### PTHIS

- Prints a null-terminated string.
- Takes the address **from the stack**: the string is placed right after the `CALL`.
- Defined by: `DEFINE_PTHIS`.

```
CALL pthis
DB 'Hello', 0
; execution resumes here
```

The PRINT macro copies its code at every use; PRINT_STRING and PTHIS print any number of strings with a single copy of the code.

# Ways to Print a String

| Method | Where is the string? | Terminator | How is the address passed? | Best for |
|---|---|---|---|---|
| `PRINT 'text'` | inside the code (macro expansion) | 0 (added by the macro) | handled by the macro (CS:SI) | a constant message printed once |
| `PRINT_STRING` | in the data area | 0 | DS:SI | strings printed in several places or built at run time |
| `PTHIS` | right after the CALL | 0 | return address on the stack | many short constant messages |
| `INT 21h`, AH = 09h | in the data area | `'$'` | DS:DX | DOS programs without the library |

### The terminator can't be printed

A '$'-terminated string cannot contain '$', and a 0-terminated one cannot contain a 0 byte: the string ends there.

### Match the buffer to the method

The buffer GET_STRING fills ends in 0, so print it with PRINT_STRING. With AH = 09h, memory is dumped to the screen until a '$' turns up.

# GET_STRING and CLEAR_SCREEN

### GET_STRING

- Reads a null-terminated string from the user.
- The string is stored in the buffer at **DS:DI**.
- **DX** = buffer size; a 0 is appended, so at most DX − 1 characters are read.
- **Enter** ends the input.
- Defined by: `DEFINE_GET_STRING`.

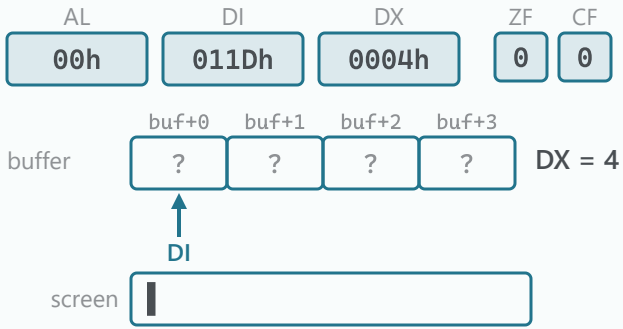
### CLEAR_SCREEN

- Clears the screen and moves the cursor to the top-left corner (0, 0).
- Defined by: `DEFINE_CLEAR_SCREEN`.

```
LEA  DI, buf          ; buffer
MOV  DX, 20           ; size
CALL get_string
CALL clear_screen
LEA  SI, buf
CALL print_string
RET
buf DB 20 DUP(0)
```

# Code Trace: GET_STRING Logic (DX = 4)

```
▶ LEA  DI, buf       ; buffer address
▶ MOV  DX, 4         ; buffer size
3  get_key: CMP DX, 1    ; keep 1 byte for the 0
4  JBE  done
5  MOV  AH, 01h
6  INT  21h          ; AL = key (echoed)
7  CMP  AL, 13       ; Enter?
8  JE   done
9  MOV  [DI], AL     ; store in buffer
10 INC  DI
11 DEC  DX
12 JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




A simplified GET_STRING read loop (the library version also handles backspace). DI = buffer, DX = 4 bytes; the user will type “Ece”.

Step 1 / 8

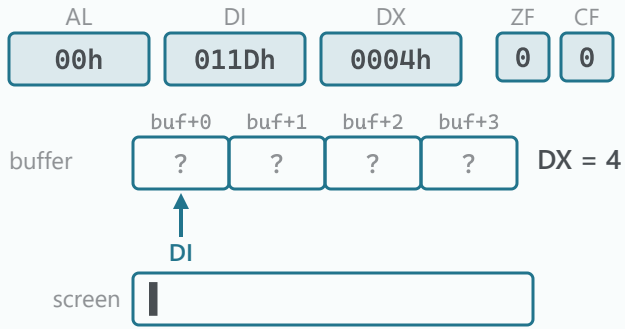
**VARIABLES**

AL = **00h** DI = **buf+0** DX = **4**

**OUTPUT**

# Code Trace: GET_STRING Logic (DX = 4)

```
1  LEA  DI, buf       ; buffer address
2  MOV  DX, 4         ; buffer size
▶ get_key: CMP DX, 1    ; keep 1 byte for the 0
▶ JBE  done
5  MOV  AH, 01h
6  INT  21h           ; AL = key (echoed)
7  CMP  AL, 13        ; Enter?
8  JE   done
9  MOV  [DI], AL      ; store in buffer
10 INC  DI
11 DEC  DX
12 JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




4 > 1: room for a character plus the final 0 → `JBE` is not taken.

Step 2 / 8

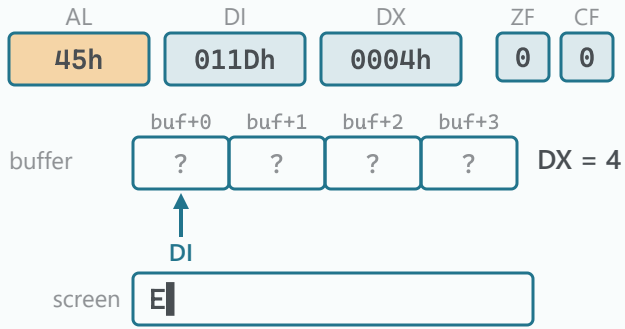
**VARIABLES**

`AL = 00h` `DI = buf+0` `DX = 4`

**OUTPUT**

# Code Trace: GET_STRING Logic (DX = 4)

```
1  LEA  DI, buf       ; buffer address
2  MOV  DX, 4         ; buffer size
3  get_key: CMP DX, 1    ; keep 1 byte for the 0
4  JBE  done
▶  MOV  AH, 01h
▶  INT  21h           ; AL = key (echoed)
▶  CMP  AL, 13        ; Enter?
▶  JE   done
9  MOV  [DI], AL      ; store in buffer
10 INC  DI
11 DEC  DX
12 JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




`INT 21h`, AH = 01h: 'E' is read and echoed. Not Enter (13) → `JE` is not taken.

Step 3 / 8

**VARIABLES**

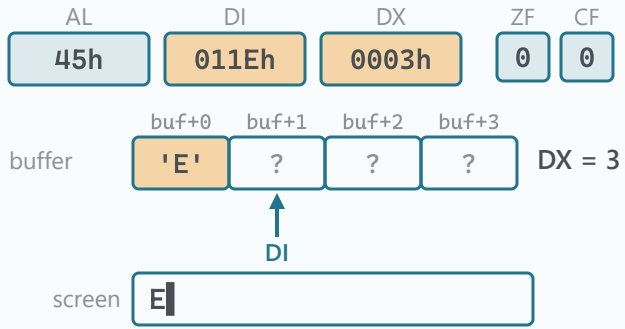
`AL = 45h` `DI = buf+0` `DX = 4`

**OUTPUT**

```
E
```

# Code Trace: GET_STRING Logic (DX = 4)

```
1  LEA  DI, buf       ; buffer address
2  MOV  DX, 4         ; buffer size
3  get_key: CMP DX, 1    ; keep 1 byte for the 0
4  JBE  done
5  MOV  AH, 01h
6  INT  21h           ; AL = key (echoed)
7  CMP  AL, 13        ; Enter?
8  JE   done
▶  MOV  [DI], AL      ; store in buffer
▶  INC  DI
▶  DEC  DX
12 JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




'E' goes into the buffer; DI advances, room left DX = 3.

Step 4 / 8

**VARIABLES**

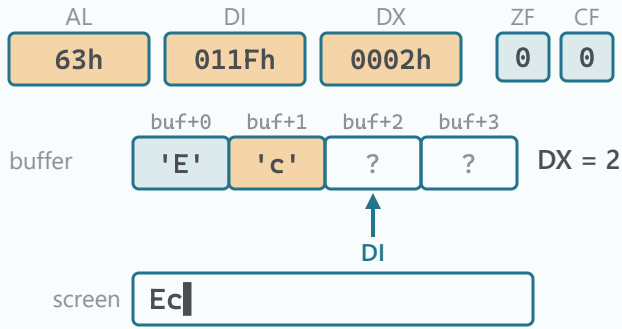
AL = **45h** | DI = **buf+1** | DX = **3**

**OUTPUT**

```
E
```

# Code Trace: GET_STRING Logic (DX = 4)

```
 1 LEA  DI, buf       ; buffer address
 2 MOV  DX, 4         ; buffer size
▶ get_key: CMP DX, 1   ; keep 1 byte for the 0
▶ JBE  done
▶ MOV  AH, 01h
▶ INT  21h           ; AL = key (echoed)
▶ CMP  AL, 13        ; Enter?
▶ JE   done
▶ MOV  [DI], AL      ; store in buffer
▶ INC  DI
▶ DEC  DX
▶ JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




Second pass: 'c' is read and stored at buf+1; DX = 2.

Step 5 / 8

**VARIABLES**

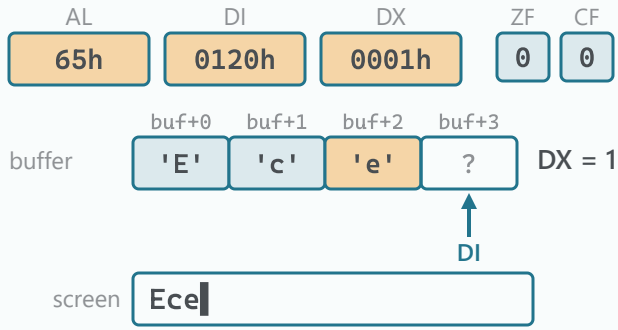
`AL = 63h` `DI = buf+2` `DX = 2`

**OUTPUT**

```
Ec
```

# Code Trace: GET_STRING Logic (DX = 4)

```
 1 LEA  DI, buf       ; buffer address
 2 MOV  DX, 4         ; buffer size
▶ get_key: CMP DX, 1   ; keep 1 byte for the 0
▶ JBE  done
▶ MOV  AH, 01h
▶ INT  21h           ; AL = key (echoed)
▶ CMP  AL, 13        ; Enter?
▶ JE   done
▶ MOV  [DI], AL      ; store in buffer
▶ INC  DI
▶ DEC  DX
▶ JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




Third pass: 'e' goes to buf+2; DX = 1.

Step 6 / 8

**VARIABLES**

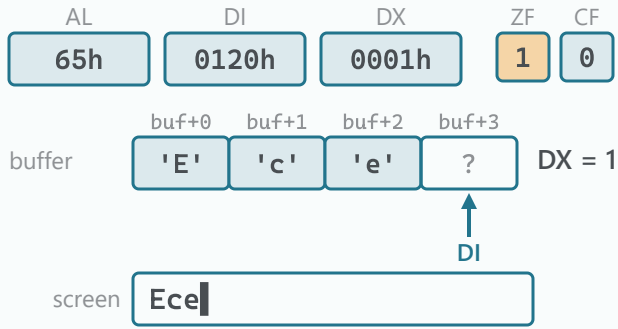
AL = **65h** DI = **buf+3** DX = **1**

**OUTPUT**

```
Ece
```

# Code Trace: GET_STRING Logic (DX = 4)

```
 1 LEA  DI, buf       ; buffer address
 2 MOV  DX, 4         ; buffer size
▶get_key: CMP DX, 1    ; keep 1 byte for the 0
▶JBE  done
 5 MOV  AH, 01h
 6 INT  21h           ; AL = key (echoed)
 7 CMP  AL, 13        ; Enter?
 8 JE   done
 9 MOV  [DI], AL      ; store in buffer
10 INC  DI
11 DEC  DX
▶JMP  get_key
13 done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




DX = 1: room only for the final 0 → `JBE done`. Further keys are not read: at most DX − 1 = 3 characters.

Step 7 / 8

**VARIABLES**

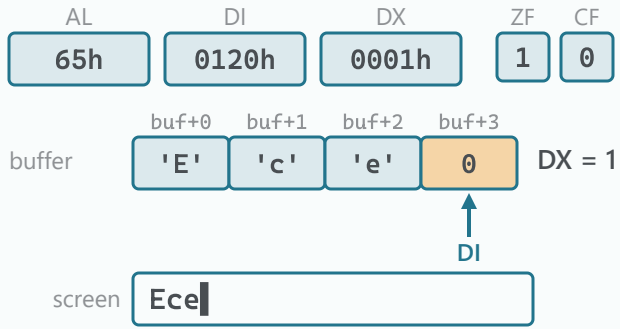
`AL = 65h` `DI = buf+3` `DX = 1`

**OUTPUT**

```
Ece
```

# Code Trace: GET_STRING Logic (DX = 4)

```
1  LEA  DI, buf       ; buffer address
2  MOV  DX, 4         ; buffer size
3  get_key: CMP DX, 1    ; keep 1 byte for the 0
4  JBE  done
5  MOV  AH, 01h
6  INT  21h           ; AL = key (echoed)
7  CMP  AL, 13        ; Enter?
8  JE   done
9  MOV  [DI], AL      ; store in buffer
10 INC  DI
11 DEC  DX
12 JMP  get_key
▶ done: MOV BYTE PTR [DI], 0
14 RET
15 buf   DB 4 DUP(?)
```




The string gets its 0 terminator: buffer 'E','c','e',0 → ready for PRINT_STRING.

Step 8 / 8

**VARIABLES**

AL = **65h** | DI = **buf+3** | DX = **1**

**OUTPUT**

```
Ece
```

# GET_STRING Details

### Controlled echo

The library version reads keys without echo ( `INT 16h` , AH = 0) and echoes only the characters it stores, so rejected keys leave no trace on screen.

### Full buffer

After DX − 1 characters, further letters are ignored; the procedure waits for Enter or backspace. Unlike the simplified trace, it does not stop by itself.

### Backspace (8)

If the buffer holds a character, DI steps back and 8, ' ', 8 is printed. On an empty buffer the key is ignored: the cursor never moves left of the input start.

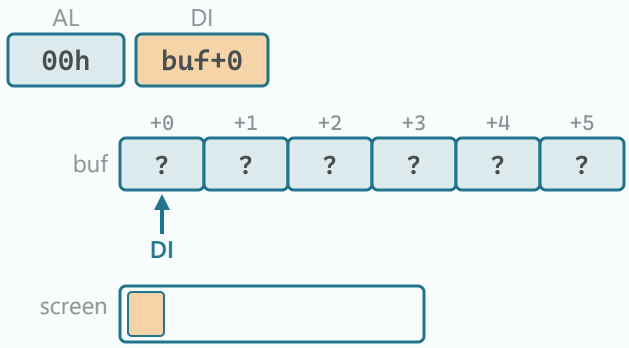
### Enter (13)

It is not stored; a 0 goes in its place and the procedure returns. With DX ≤ 1 nothing is read at all, since there is room only for the final 0.

After GET_STRING the cursor stays just right of the input, so the next output starts on the same line. Add `PUTC 13` and `PUTC 10` to move down.

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 3 get_key: MOV AH, 0
 4 INT  16h          ; AL = key (no echo)
 5 CMP  AL, 13       ; Enter?
 6 JE   done
 7 MOV  AH, 0Eh
 8 CMP  AL, 8        ; backspace?
 9 JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
19 store: MOV [DI], AL
20 INC  DI
21 INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




Buffer `buf` is 6 bytes; DI points to its first byte. The user will press A, b, backspace, c and Enter.

Step 1 / 9

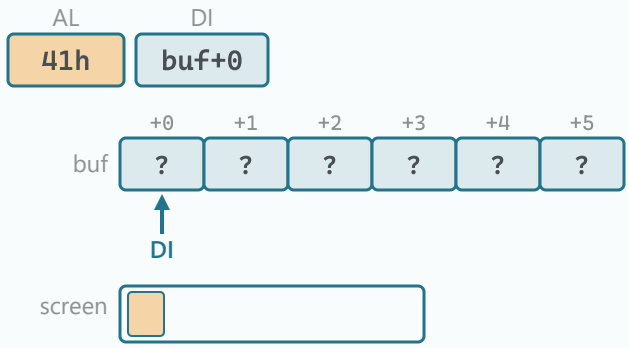
**VARIABLES**

`AL = 00h` `DI = buf+0` `cursor = 0`

**OUTPUT**

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 ▶ get_key: MOV AH, 0
 ▶ INT  16h          ; AL = key (no echo)
 ▶ CMP  AL, 13       ; Enter?
 ▶ JE   done
 ▶ MOV  AH, 0Eh
 ▶ CMP  AL, 8        ; backspace?
 ▶ JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
19 store: MOV [DI], AL
20 INC  DI
21 INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




`INT 16h` : AL = 'A' (41h); the key is read without echo, so the screen is still empty. Neither Enter (13) nor backspace (8): `store` .

Step 2 / 9

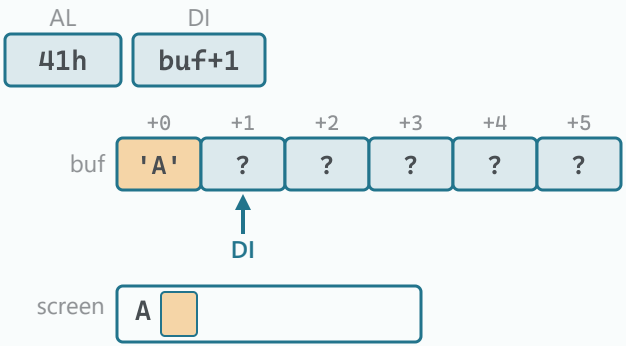
**VARIABLES**

`AL = 41h` `DI = buf+0` `cursor = 0`

**OUTPUT**

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 3 get_key: MOV AH, 0
 4 INT  16h          ; AL = key (no echo)
 5 CMP  AL, 13       ; Enter?
 6 JE   done
 7 MOV  AH, 0Eh
 8 CMP  AL, 8        ; backspace?
 9 JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
▶ store: MOV [DI], AL
▶ INC  DI
▶ INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




'A' is stored and DI advances; then the echo: A on screen.

Step 3 / 9

**VARIABLES**

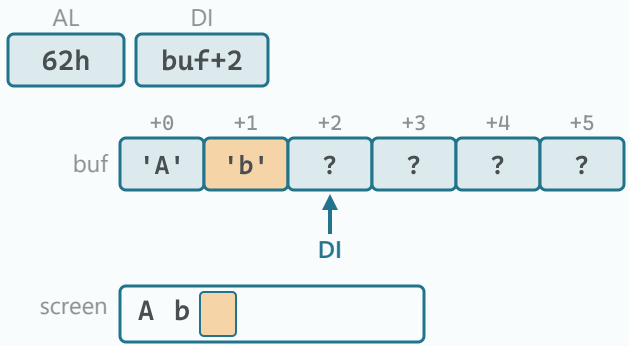
AL = 41h  DI = buf+1  cursor = 1

**OUTPUT**

```
A
```

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 ▶ get_key: MOV AH, 0
 ▶ INT  16h          ; AL = key (no echo)
 ▶ CMP  AL, 13       ; Enter?
 ▶ JE   done
 ▶ MOV  AH, 0Eh
 ▶ CMP  AL, 8        ; backspace?
 ▶ JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
 ▶ store: MOV [DI], AL
 ▶ INC  DI
 ▶ INT  10h          ; echo
 ▶ JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




Same path for 'b': buffer A, b; DI = buf+2; the screen shows Ab.

Step 4 / 9

**VARIABLES**

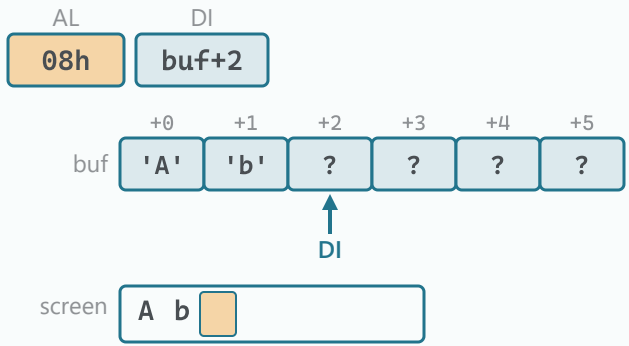
`AL = 62h` `DI = buf+2` `cursor = 2`

**OUTPUT**

```
Ab
```

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 3 get_key: MOV AH, 0
 4 INT  16h          ; AL = key (no echo)
 5 CMP  AL, 13       ; Enter?
 6 JE   done
 7 MOV  AH, 0Eh
 8 CMP  AL, 8        ; backspace?
 9 JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
19 store: MOV [DI], AL
20 INC  DI
21 INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




Backspace: AL = 08h. `CMP AL, 8` → ZF = 1, `JNE store` is not taken.

Step 5 / 9

**VARIABLES**

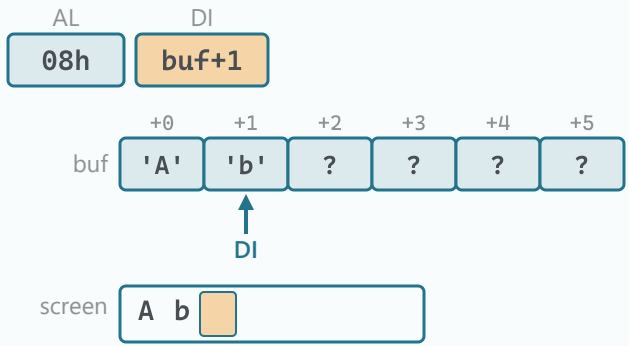
`AL = 08h` `DI = buf+2` `cursor = 2`

**OUTPUT**

```
Ab
```

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 3 get_key: MOV AH, 0
 4 INT  16h          ; AL = key (no echo)
 5 CMP  AL, 13       ; Enter?
 6 JE   done
 7 MOV  AH, 0Eh
 8 CMP  AL, 8        ; backspace?
 9 JNE  store
▶ CMP  DI, OFFSET buf
▶ JE   get_key          ; nothing to erase
▶ DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
19 store: MOV [DI], AL
20 INC  DI
21 INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




DI is not at the buffer start (buf+2 ≠ buf): there is a character to erase. `DEC DI` frees b's byte; it stays in memory but is no longer part of the string.

Step 6 / 9

**VARIABLES**

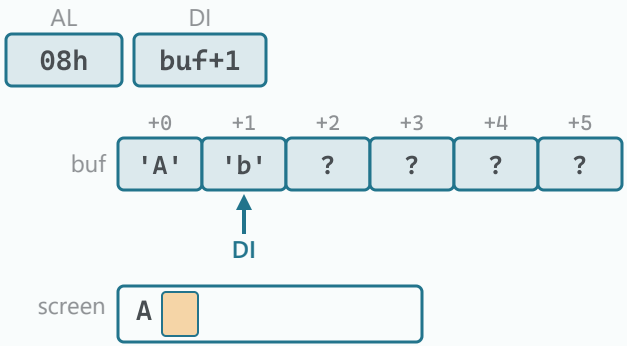
`AL = 08h` `DI = buf+1` `cursor = 2`

**OUTPUT**

```
Ab
```

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 3 get_key: MOV AH, 0
 4 INT  16h          ; AL = key (no echo)
 5 CMP  AL, 13       ; Enter?
 6 JE   done
 7 MOV  AH, 0Eh
 8 CMP  AL, 8        ; backspace?
 9 JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
▶ INT  10h          ; 8: cursor left
▶ MOV  AL, ' '
▶ INT  10h          ; blank the char
▶ MOV  AL, 8
▶ INT  10h          ; left again
18 JMP  get_key
19 store: MOV [DI], AL
20 INC  DI
21 INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




On screen, the triple: 8 moves the cursor left, a space overwrites b, the second 8 moves left again. Only A remains.

Step 7 / 9

**VARIABLES**

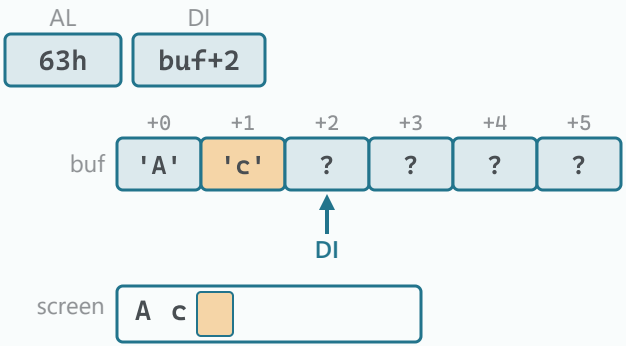
AL = 08h  DI = buf+1  cursor = 1

**OUTPUT**

```
A
```

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 ▶ get_key: MOV AH, 0
 ▶ INT  16h          ; AL = key (no echo)
 ▶ CMP  AL, 13       ; Enter?
 ▶ JE   done
 ▶ MOV  AH, 0Eh
 ▶ CMP  AL, 8        ; backspace?
 ▶ JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
 ▶ JMP  get_key
 ▶ store: MOV [DI], AL
 ▶ INC  DI
 ▶ INT  10h          ; echo
22 JMP  get_key
23 done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




'c' goes into b's old byte: buffer A, c; the screen shows Ac.

Step 8 / 9

**VARIABLES**

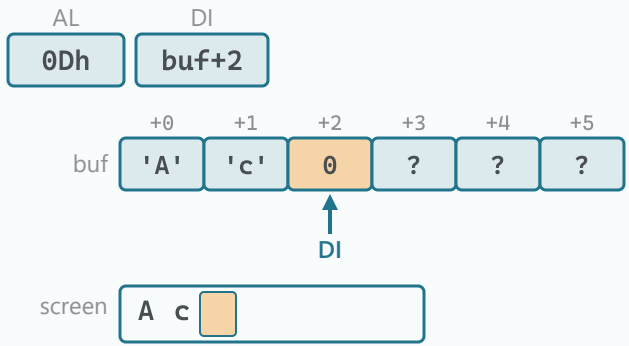
`AL = 63h` `DI = buf+2` `cursor = 2`

**OUTPUT**

```
Ac
```

# Code Trace: Reading with Backspace

```
 1 ORG  100h
 2 LEA  DI, buf
 ▶ get_key: MOV AH, 0
 ▶ INT  16h          ; AL = key (no echo)
 ▶ CMP  AL, 13       ; Enter?
 ▶ JE   done
 7 MOV  AH, 0Eh
 8 CMP  AL, 8        ; backspace?
 9 JNE  store
10 CMP  DI, OFFSET buf
11 JE   get_key          ; nothing to erase
12 DEC  DI
13 INT  10h          ; 8: cursor left
14 MOV  AL, ' '
15 INT  10h          ; blank the char
16 MOV  AL, 8
17 INT  10h          ; left again
18 JMP  get_key
19 store: MOV [DI], AL
20 INC  DI
21 INT  10h          ; echo
 ▶ JMP  get_key
 ▶ done: MOV BYTE PTR [DI], 0
24 RET
25 buf   DB 6 DUP(?)
```




Enter: `JE done` ; a 0 is written where DI points. Buffer 'A', 'c', 0: the erased letter is gone.

Step 9 / 9

**VARIABLES**

AL = 0Dh | DI = buf+2 | cursor = 2

**OUTPUT**

```
Ac
```

# Number Procedures

| Procedure | What it does | Input / output | Defined by |
|---|---|---|---|
| `SCAN_NUM` | Reads a multi-digit **signed** number from the keyboard (ends with Enter). | output: **CX** | `DEFINE_SCAN_NUM` |
| `PRINT_NUM` | Prints the **signed** number in AX in decimal. | input: **AX** | `DEFINE_PRINT_NUM` + `DEFINE_PRINT_NUM_UNS` |
| `PRINT_NUM_UNS` | Prints the **unsigned** number in AX. | input: **AX** | `DEFINE_PRINT_NUM_UNS` |

**Signed range**

16 bits: −32,768 … 32,767

**Unsigned range**

16 bits: 0 … 65,535

**Same bits**

`FFE7h` = −25 (signed) = 65,511 (unsigned)

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
▶  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX           ; AX = result × 10
12 ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0000h | 0000h | 0000h | 1 |

CX decimal: **0**

screen

The core of SCAN_NUM: digits are read left to right, result = result × 10 + digit (the library version also handles a minus sign and overflow). CX = 0.

Step 1 / 11

**VARIABLES**

AL = 00h | CX = 0000h | BX = 0000h

**OUTPUT**

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
▶ get_key: MOV AH, 01h
▶ INT  21h           ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX           ; AX = result × 10
12 ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0133h | 0000h | 0000h | 1 |

CX decimal: **0**

screen: 3

`INT 21h`, AH = 01h: key '3', AL = 33h; the character is echoed.

Step 2 / 11

**VARIABLES**

AL = 33h | CX = 0000h | BX = 0000h

**OUTPUT**

```
3
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
▶  CMP  AL, 13       ; Enter?
▶  JE   done
▶  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX           ; AX = result × 10
12 ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0103h | 0000h | 0000h | 0 |

CX decimal: **0**

screen: `3`

Not Enter. `SUB AL, '0'` : 33h − 30h = 3.

Step 3 / 11

**VARIABLES**

AL = 03h | CX = 0000h | BX = 0000h

**OUTPUT**

```
3
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
▶  CBW               ; AX = digit
▶  XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX           ; AX = result × 10
12 ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0000h | 0003h | 0000h | 0 |

CX decimal: **3**

screen: 3

`CBW` sets AX = 0003h; `XCHG` swaps: AX = old result (0), CX = digit (3).

Step 4 / 11

**VARIABLES**

AL = 00h | CX = 0003h | BX = 0000h

**OUTPUT**

```
3
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
▶  MOV  BX, 10
▶  MUL  BX           ; AX = result × 10
▶  ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0000h | 0003h | 000Ah | 0 |

CX decimal: **3**

screen: `3`

`MUL BX`: 0 × 10 = 0; `ADD CX, AX` → CX = 3.

Step 5 / 11

**VARIABLES**

AL = 00h | CX = 0003h | BX = 000Ah

**OUTPUT**

```
3
```

# Code Trace: SCAN_NUM Logic (input 307)

```
 1 ORG  100h
 2 XOR  CX, CX       ; result = 0
▶ get_key: MOV AH, 01h
▶ INT  21h          ; AL = key (echoed)
▶ CMP  AL, 13       ; Enter?
▶ JE   done
▶ SUB  AL, '0'      ; ASCII → digit
▶ CBW               ; AX = digit
▶ XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX          ; AX = result × 10
12 ADD  CX, AX      ; CX = result×10 + digit
▶ JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0003h | 0000h | 000Ah | 1 |

CX decimal: **0**

screen: `30`

'0' is read: digit 0; after XCHG, AX = 3, CX = 0.

Step 6 / 11

**VARIABLES**

AL = 03h | CX = 0000h | BX = 000Ah

**OUTPUT**

```
30
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
▶  MOV  BX, 10
▶  MUL  BX           ; AX = result × 10
▶  ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 001Eh | 001Eh | 000Ah | 0 |

CX decimal: **30**

screen: 30

3 × 10 = 30 (1Eh) → CX = 30 + 0 = 30.

Step 7 / 11

**VARIABLES**

AL = 1Eh | CX = 001Eh | BX = 000Ah

**OUTPUT**

```
30
```

# Code Trace: SCAN_NUM Logic (input 307)

```
 1 ORG  100h
 2 XOR  CX, CX       ; result = 0
▶get_key: MOV AH, 01h
▶INT  21h           ; AL = key (echoed)
▶CMP  AL, 13        ; Enter?
▶JE   done
▶SUB  AL, '0'       ; ASCII → digit
▶CBW                ; AX = digit
 9 XCHG AX, CX      ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX          ; AX = result × 10
12 ADD  CX, AX      ; CX = result×10 + digit
▶JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 0007h | 001Eh | 000Ah | 0 |

CX decimal: **30**

screen: **307**

'7' is read: digit 7 into AX (CBW); the result 30 is still in CX.

Step 8 / 11

**VARIABLES**

AL = 07h | CX = 001Eh | BX = 000Ah

**OUTPUT**

```
307
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
▶  XCHG AX, CX       ; AX=result, CX=digit
▶  MOV  BX, 10
▶  MUL  BX           ; AX = result × 10
▶  ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 012Ch | 0133h | 000Ah | 0 |

CX decimal: **307**

screen: 307

XCHG: AX = 30, CX = 7. 30 × 10 = 300 (012Ch) → CX = 300 + 7 = 307 = 0133h.

Step 9 / 11

**VARIABLES**

AL = 2Ch | CX = 0133h | BX = 000Ah

**OUTPUT**

```
307
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
▶ get_key: MOV AH, 01h
▶ INT  21h           ; AL = key (echoed)
▶ CMP  AL, 13        ; Enter?
▶ JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX           ; AX = result × 10
12 ADD  CX, AX       ; CX = result×10 + digit
▶ JMP  get_key
14 done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 010Dh | 0133h | 000Ah | 1 |

CX decimal: **307**

screen: 307

Enter: AL = 0Dh = 13 → ZF = 1, `JE done` is taken.

Step 10 / 11

**VARIABLES**

AL = 0Dh | CX = 0133h | BX = 000Ah

**OUTPUT**

```
307
```

# Code Trace: SCAN_NUM Logic (input 307)

```
1  ORG  100h
2  XOR  CX, CX       ; result = 0
3  get_key: MOV AH, 01h
4  INT  21h          ; AL = key (echoed)
5  CMP  AL, 13       ; Enter?
6  JE   done
7  SUB  AL, '0'      ; ASCII → digit
8  CBW               ; AX = digit
9  XCHG AX, CX       ; AX=result, CX=digit
10 MOV  BX, 10
11 MUL  BX           ; AX = result × 10
12 ADD  CX, AX       ; CX = result×10 + digit
13 JMP  get_key
▶  done: RET
```

| AX | CX | BX | ZF |
|---|---|---|---|
| 010Dh | 0133h | 000Ah | 1 |

CX decimal: **307**

screen: 307

As in SCAN_NUM, the result is in CX: 0133h = 307.

Step 11 / 11

**VARIABLES**

AL = 0Dh | CX = 0133h | BX = 000Ah

**OUTPUT**

```
307
```

# What SCAN_NUM Checks

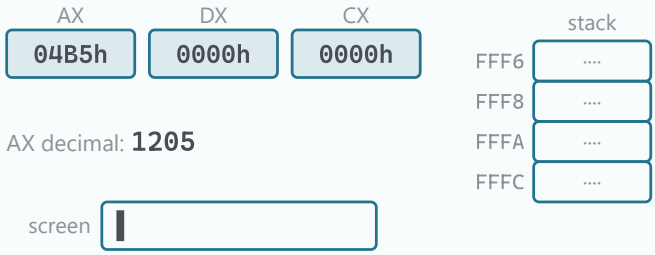
- The `-` key sets a sign flag; on Enter, if the flag is set, `NEG CX` is applied.
- A non-digit key is erased from the screen (8, ' ', 8) and ignored.
- Backspace drops the last digit: CX ← CX / 10.
- If the number would not fit in 16 bits (the × 10 or the addition overflows), the last digit is rejected.
- Enter is echoed too: the cursor returns to the line start (CR) but does not move down. That is why the PTHIS string in the example program starts with 13, 10.

| Input | CX | PRINT_NUM |
|---|---|---|
| 307 | `0133h` | 307 |
| -25 | `FFE7h` | -25 |
| 12a3 | `007Bh` | 123 |
| 40000 | `9C40h` | -25536 |
| 70000 | `1B58h` | 7000 |
| -40000 | `63C0h` | 25536 |

Digits accumulate as an unsigned 16-bit value and the sign is applied last, so inputs outside −32,768 … 32,767 silently give a wrongly signed result. (The table comes from running code equivalent to the library routine.)

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
 1 ORG  100h
 ▶ MOV  AX, 1205     ; number to print
 ▶ MOV  BX, 10
 ▶ XOR  CX, CX       ; digit counter
 ▶ divide: XOR DX, DX
 6 DIV  BX           ; AX=quot, DX=rem
 7 PUSH DX           ; push the digit
 8 INC  CX
 9 OR   AX, AX       ; quotient 0?
10 JNZ  divide
11 emit: POP AX       ; MSD first
12 ADD  AL, '0'
13 MOV  AH, 0Eh
14 INT  10h
15 LOOP emit
16 RET
```




Equivalent code with the same output (the library version divides by 10000, 1000, ...). AX = 1205 = 04B5h, divisor BX = 10.

Step 1 / 10

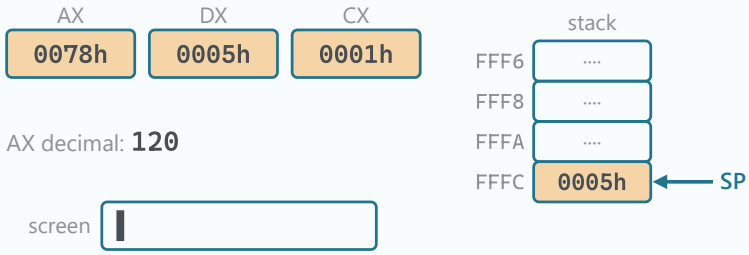
**VARIABLES**

AX = **04B5h** DX = **0000h** CX = **0000h** SP = **FFFEh**

**OUTPUT**

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
5  divide: XOR DX, DX
▶ DIV  BX            ; AX=quot, DX=rem
▶ PUSH DX            ; push the digit
▶ INC  CX
9  OR   AX, AX       ; quotient 0?
10 JNZ  divide
11 emit: POP AX      ; MSD first
12 ADD  AL, '0'
13 MOV  AH, 0Eh
14 INT  10h
15 LOOP emit
16 RET
```




`DIV BX` : DX:AX / 10 → AX = 120, remainder DX = 5. The last digit is pushed, CX = 1.

Step 2 / 10

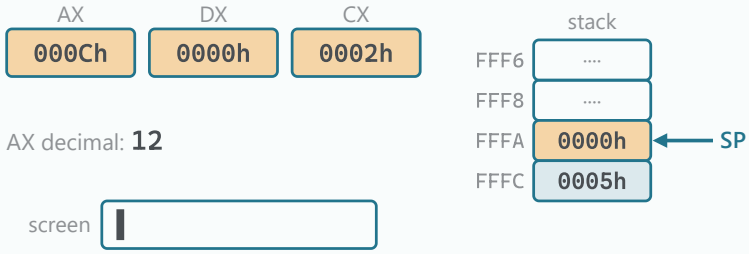
**VARIABLES**

AX = 0078h  DX = 0005h  CX = 0001h  SP = FFFCh

**OUTPUT**

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
▶ divide: XOR DX, DX
▶ DIV  BX            ; AX=quot, DX=rem
▶ PUSH DX            ; push the digit
▶ INC  CX
▶ OR   AX, AX        ; quotient 0?
▶ JNZ  divide
11 emit: POP AX       ; MSD first
12 ADD  AL, '0'
13 MOV  AH, 0Eh
14 INT  10h
15 LOOP emit
16 RET
```




AX ≠ 0 → loop again: 120 / 10 = 12, remainder 0 pushed; CX = 2.

Step 3 / 10

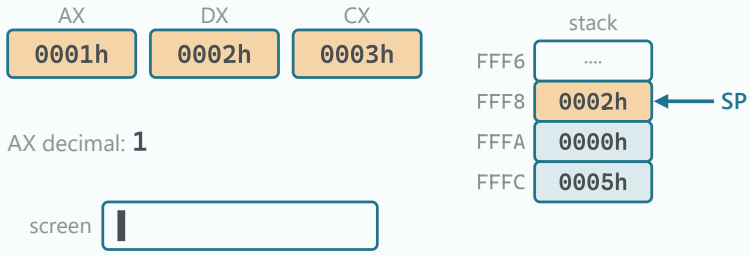
**VARIABLES**

AX = 000Ch  DX = 0000h  CX = 0002h  SP = FFFAh

**OUTPUT**

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
 1 ORG  100h
 2 MOV  AX, 1205     ; number to print
 3 MOV  BX, 10
 4 XOR  CX, CX       ; digit counter
▶ divide: XOR DX, DX
▶ DIV  BX           ; AX=quot, DX=rem
▶ PUSH DX           ; push the digit
▶ INC  CX
▶ OR   AX, AX       ; quotient 0?
▶ JNZ  divide
11 emit: POP AX      ; MSD first
12 ADD  AL, '0'
13 MOV  AH, 0Eh
14 INT  10h
15 LOOP emit
16 RET
```




12 / 10 = 1, remainder 2 pushed; CX = 3.

Step 4 / 10

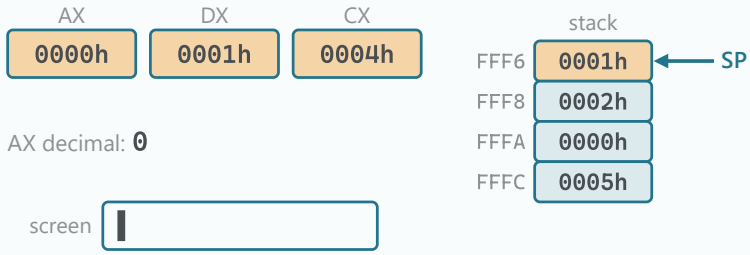
**VARIABLES**

AX = 0001h  DX = 0002h  CX = 0003h  SP = FFF8h

**OUTPUT**

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
▶ divide: XOR DX, DX
▶ DIV  BX            ; AX=quot, DX=rem
▶ PUSH DX            ; push the digit
▶ INC  CX
▶ OR   AX, AX        ; quotient 0?
▶ JNZ  divide
11 emit: POP AX       ; MSD first
12 ADD  AL, '0'
13 MOV  AH, 0Eh
14 INT  10h
15 LOOP emit
16 RET
```




1 / 10 = 0, remainder 1 pushed; CX = 4 digits.

Step 5 / 10

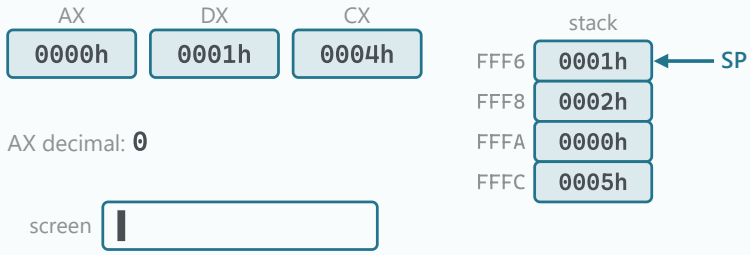
**VARIABLES**

AX = 0000h  DX = 0001h  CX = 0004h  SP = FFF6h

**OUTPUT**

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
5  divide: XOR DX, DX
6  DIV  BX           ; AX=quot, DX=rem
7  PUSH DX           ; push the digit
8  INC  CX
▶ OR   AX, AX        ; quotient 0?
▶ JNZ  divide
11 emit: POP AX      ; MSD first
12 ADD  AL, '0'
13 MOV  AH, 0Eh
14 INT  10h
15 LOOP emit
16 RET
```




`OR AX, AX` : AX = 0 → ZF = 1, `JNZ` is not taken. The most significant digit (1) is on top of the stack.

Step 6 / 10

**VARIABLES**

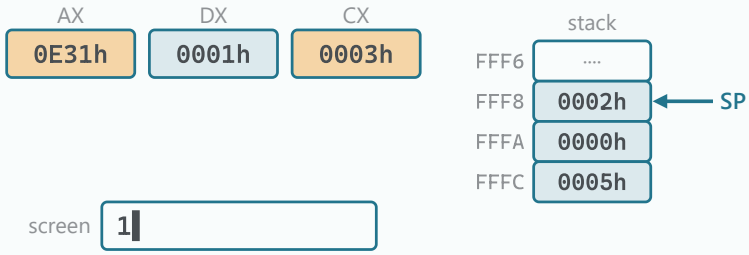
AX = **0000h** DX = **0001h** CX = **0004h**

SP = **FFF6h**

**OUTPUT**

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
5  divide: XOR DX, DX
6  DIV  BX           ; AX=quot, DX=rem
7  PUSH DX           ; push the digit
8  INC  CX
9  OR   AX, AX       ; quotient 0?
10 JNZ  divide
▶  emit: POP AX      ; MSD first
▶  ADD  AL, '0'
▶  MOV  AH, 0Eh
▶  INT  10h
▶  LOOP emit
16 RET
```




POP AX = 1; + '0' = 31h; `INT 10h` prints '1'. LOOP: CX = 3.

Step 7 / 10

**VARIABLES**

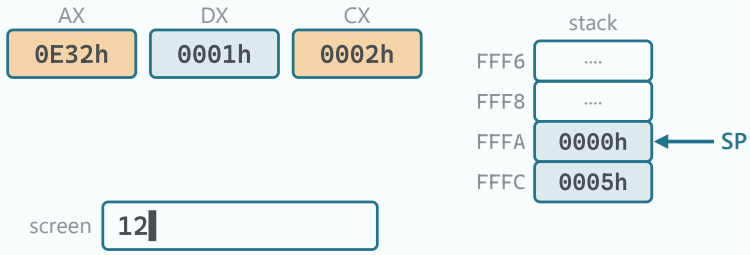
AX = 0E31h  DX = 0001h  CX = 0003h  SP = FFF8h

**OUTPUT**

```
1
```

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
5  divide: XOR DX, DX
6  DIV  BX           ; AX=quot, DX=rem
7  PUSH DX           ; push the digit
8  INC  CX
9  OR   AX, AX       ; quotient 0?
10 JNZ  divide
▶ emit: POP AX       ; MSD first
▶ ADD  AL, '0'
▶ MOV  AH, 0Eh
▶ INT  10h
▶ LOOP emit
16 RET
```




Next digit 2 → '2'. CX = 2.

Step 8 / 10

**VARIABLES**

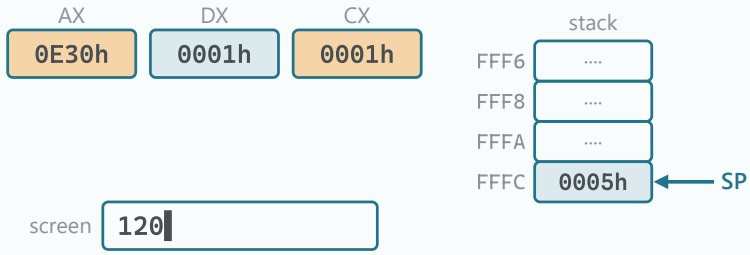
AX = 0E32h  DX = 0001h  CX = 0002h  SP = FFFAh

**OUTPUT**

```
12
```

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
5  divide: XOR DX, DX
6  DIV  BX           ; AX=quot, DX=rem
7  PUSH DX           ; push the digit
8  INC  CX
9  OR   AX, AX       ; quotient 0?
10 JNZ  divide
▶ emit: POP AX       ; MSD first
▶ ADD  AL, '0'
▶ MOV  AH, 0Eh
▶ INT  10h
▶ LOOP emit
16 RET
```




0 → '0'. CX = 1.

Step 9 / 10

**VARIABLES**

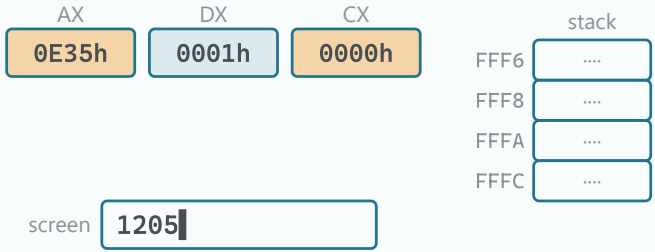
AX = 0E30h  DX = 0001h  CX = 0001h  SP = FFFCh

**OUTPUT**

```
120
```

# Code Trace: PRINT_NUM_UNS Logic (AX = 1205)

```
1  ORG  100h
2  MOV  AX, 1205     ; number to print
3  MOV  BX, 10
4  XOR  CX, CX       ; digit counter
5  divide: XOR DX, DX
6  DIV  BX           ; AX=quot, DX=rem
7  PUSH DX           ; push the digit
8  INC  CX
9  OR   AX, AX       ; quotient 0?
10 JNZ  divide
▶ emit: POP AX       ; MSD first
▶ ADD  AL, '0'
▶ MOV  AH, 0Eh
▶ INT  10h
▶ LOOP emit
16 RET
```




5 → '5'. CX = 0, the loop ends; SP is back at FFFEh. The screen shows 1205.

Step 10 / 10

**VARIABLES**

AX = 0E35h  DX = 0001h  CX = 0000h  SP = FFFEh

**OUTPUT**

```
1205
```

# The Library's Method: A Shrinking Divisor

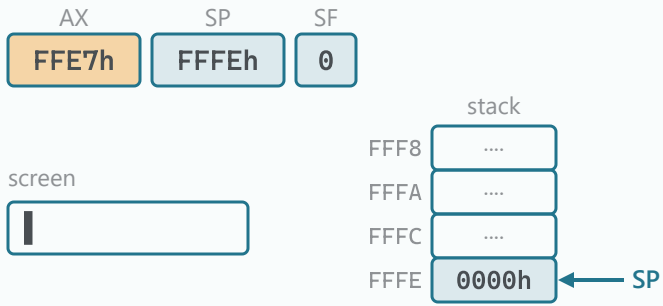
- The divisor starts at 10,000 and is divided by 10 each pass; printing stops when it reaches 0.
- The quotient (0–9) is one digit, printed after `ADD AL, 30h`; the remainder becomes the number for the next pass.
- A flag (CX = 1) skips leading zeros; once the first digit is printed the flag is cleared and inner zeros are printed.
- A separate branch prints '0' directly when AX = 0.
- Since 65,535 ÷ 10,000 = 6, the quotient always fits in AL.

| Divisor | Number | Step | Screen |
|---|---|---|---|
| 10000 | 3007 | number < divisor, no digit yet: skip | — |
| 1000 | 3007 | 3007 ÷ 1000 = 3, rem. 7 | 3 |
| 100 | 7 | 7 ÷ 100 = 0, rem. 7 | 30 |
| 10 | 7 | 7 ÷ 10 = 0, rem. 7 | 300 |
| 1 | 7 | 7 ÷ 1 = 7, rem. 0 | 3007 |

Digits are printed left to right as they are computed; no pushing and no reversing needed.

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG  100h
▶ MOV  AX, -25        ; AX = FFE7h
3  CALL print_num
4  RET
5  print_num PROC
6  PUSH AX
7  OR   AX, AX        ; SF = sign bit
8  JNS  positive
9  NEG  AX            ; AX = 0019h
10 PUSH AX            ; PUTC '-' expanded
11 MOV  AL, '-'
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 positive: CALL print_num_uns
16 POP  AX            ; AX = FFE7h again
17 RET
18 print_num ENDP
```




AX = −25, which is FFE7h in two's complement. The code shown is equivalent to the library's PRINT_NUM.

Step 1 / 8

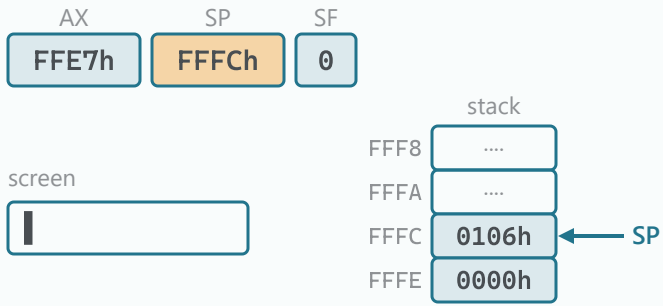
**VARIABLES**

AX = **FFE7h** SP = **FFFEh** SF = **0**

**OUTPUT**

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG  100h
2  MOV  AX, -25        ; AX = FFE7h
▶  CALL print_num
4  RET
5  print_num PROC
6  PUSH AX
7  OR   AX, AX         ; SF = sign bit
8  JNS  positive
9  NEG  AX             ; AX = 0019h
10 PUSH AX             ; PUTC '-' expanded
11 MOV  AL, '-'
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 positive: CALL print_num_uns
16 POP  AX             ; AX = FFE7h again
17 RET
18 print_num ENDP
```




`CALL print_num` : return address 0106h is pushed, SP = FFFCh.

Step 2 / 8

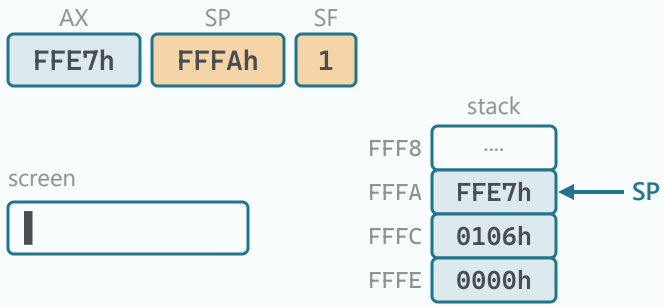
**VARIABLES**

AX = **FFE7h** SP = **FFFCh** SF = **0**

**OUTPUT**

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG  100h
2  MOV  AX, -25        ; AX = FFE7h
3  CALL print_num
4  RET
5  print_num PROC
▶  PUSH AX
▶  OR   AX, AX         ; SF = sign bit
8  JNS  positive
9  NEG  AX             ; AX = 0019h
10 PUSH AX             ; PUTC '-' expanded
11 MOV  AL, '-'
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 positive: CALL print_num_uns
16 POP  AX             ; AX = FFE7h again
17 RET
18 print_num ENDP
```




`PUSH AX` saves the caller's value. `OR AX, AX` leaves the value unchanged but sets the flags: MSB is 1 → SF = 1.

Step 3 / 8

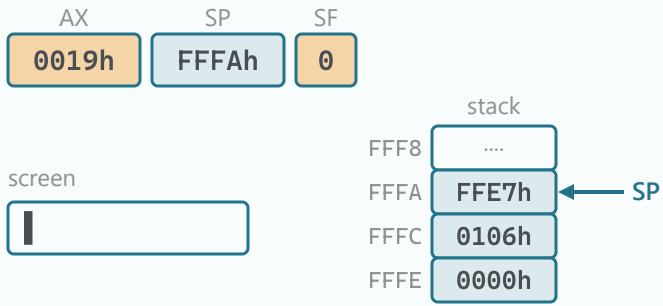
**VARIABLES**

`AX = FFE7h` `SP = FFFAh` `SF = 1`

**OUTPUT**

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG  100h
2  MOV  AX, -25        ; AX = FFE7h
3  CALL print_num
4  RET
5  print_num PROC
6  PUSH AX
7  OR   AX, AX         ; SF = sign bit
▶  JNS  positive
▶  NEG  AX             ; AX = 0019h
10 PUSH AX             ; PUTC '-' expanded
11 MOV  AL, '-'
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 positive: CALL print_num_uns
16 POP  AX             ; AX = FFE7h again
17 RET
18 print_num ENDP
```




`JNS` is not taken (the number is negative). `NEG AX`: 0 − FFE7h = 0019h = 25, the absolute value.

Step 4 / 8

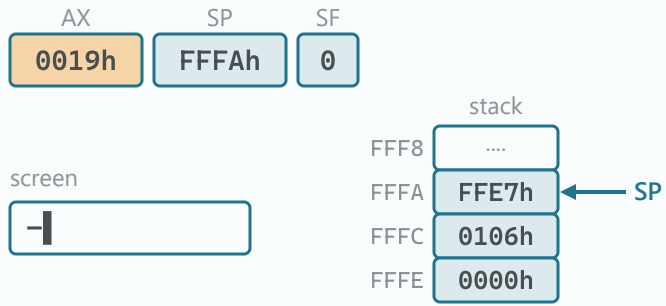
**VARIABLES**

`AX = 0019h` `SP = FFFAh` `SF = 0`

**OUTPUT**

# Code Trace: PRINT_NUM and the Minus Sign

```
 1 ORG  100h
 2 MOV  AX, -25        ; AX = FFE7h
 3 CALL print_num
 4 RET
 5 print_num PROC
 6 PUSH AX
 7 OR   AX, AX         ; SF = sign bit
 8 JNS  positive
 9 NEG  AX             ; AX = 0019h
▶ PUSH AX             ; PUTC '-' expanded
▶ MOV  AL, '-'
▶ MOV  AH, 0Eh
▶ INT  10h
▶ POP  AX
15 positive: CALL print_num_uns
16 POP  AX            ; AX = FFE7h again
17 RET
18 print_num ENDP
```




`PUTC '-'` expansion: AX is pushed, '-' is printed, AX = 0019h is restored.

Step 5 / 8

**VARIABLES**

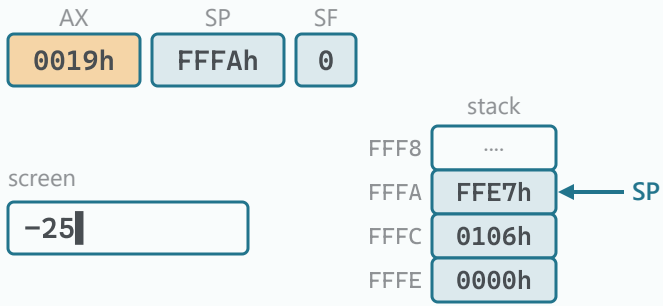
AX = **0019h** SP = **FFFAh** SF = **0**

**OUTPUT**

```
-
```

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG   100h
2  MOV   AX, -25        ; AX = FFE7h
3  CALL  print_num
4  RET
5  print_num PROC
6  PUSH  AX
7  OR    AX, AX         ; SF = sign bit
8  JNS   positive
9  NEG   AX             ; AX = 0019h
10 PUSH  AX             ; PUTC '-' expanded
11 MOV   AL, '-'
12 MOV   AH, 0Eh
13 INT   10h
14 POP   AX
▶  positive: CALL print_num_uns
16 POP   AX             ; AX = FFE7h again
17 RET
18 print_num ENDP
```




`CALL print_num_uns` : the digits of 25 are printed (same digit loop as in the PRINT_NUM_UNS trace; the procedure saves the registers it uses).

Step 6 / 8

**VARIABLES**

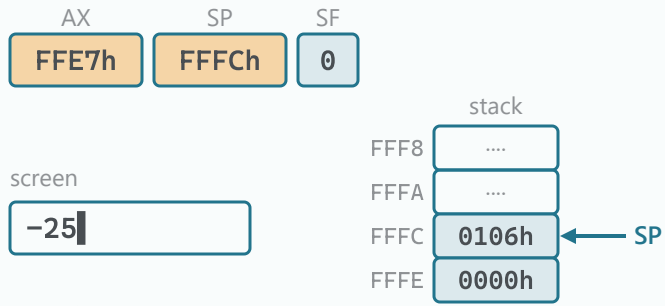
AX = **0019h** SP = **FFFAh** SF = **0**

**OUTPUT**

```
-25
```

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG  100h
2  MOV  AX, -25        ; AX = FFE7h
3  CALL print_num
4  RET
5  print_num PROC
6  PUSH AX
7  OR   AX, AX         ; SF = sign bit
8  JNS  positive
9  NEG  AX             ; AX = 0019h
10 PUSH AX             ; PUTC '-' expanded
11 MOV  AL, '-'
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 positive: CALL print_num_uns
▶ POP  AX             ; AX = FFE7h again
17 RET
18 print_num ENDP
```




`POP AX` : AX is FFE7h again. The number is still in AX after printing.

Step 7 / 8

**VARIABLES**

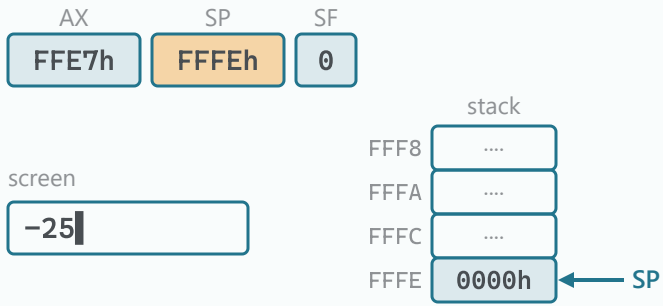
AX = FFE7h  SP = FFFCh  SF = 0

**OUTPUT**

```
-25
```

# Code Trace: PRINT_NUM and the Minus Sign

```
1  ORG  100h
2  MOV  AX, -25        ; AX = FFE7h
3  CALL print_num
4  RET
5  print_num PROC
6  PUSH AX
7  OR   AX, AX         ; SF = sign bit
8  JNS  positive
9  NEG  AX             ; AX = 0019h
10 PUSH AX             ; PUTC '-' expanded
11 MOV  AL, '-'
12 MOV  AH, 0Eh
13 INT  10h
14 POP  AX
15 positive: CALL print_num_uns
16 POP  AX             ; AX = FFE7h again
▶ RET
18 print_num ENDP
```




Both RETs: SP = FFFEh. The library's PRINT_NUM also prints '0' directly when AX = 0.

Step 8 / 8

**VARIABLES**

AX = **FFE7h** SP = **FFFEh** SF = **0**

**OUTPUT**

```
-25
```

# Extracting Digits: Two Methods

| Criterion | Stack method (divide by 10, push, pop in reverse) | Shrinking divisor (10,000, 1000, ...) |
|---|---|---|
| Divisions | one per digit | up to five, plus one each to shrink the divisor |
| Stack use | 2 bytes per digit | only the saved registers |
| Leading zeros | never appear | skipped with a flag |
| Fixed width ( `00042` ) | needs extra code | free: just drop the flag |
| Other bases | use divisor 2, 8 or 16; in base 16, digits above 9 become 'A'–'F' | the divisor sequence depends on the base (4096, 256, 16, 1 for base 16) |
| Output order | digits go to the stack, then to the screen in reverse | left to right, as computed |

Both methods give the same output: for 1205 each prints the digits 1, 2, 0, 5. Stack use, fixed width and the base you need decide which to pick.

PROCEDURES

# Demo: PRINT_NUM and PRINT_NUM_UNS

Enter a value for AX (decimal, negative, or hex with an `h` suffix) to see what each procedure prints.

AX = -25 **−25** **32767** **8000h** **FFFFh**

| AX | PRINT_NUM | PRINT_NUM_UNS |
|---|---|---|
| FFE7h<br>1111 1111 1110 0111 | −25 | 65511 |

Bit 15 = 1: negative when read as signed (65511 − 65536 = −25).

EXAMPLE PROGRAM · STEP THROUGH THE CODE

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
▶ LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
5  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

`LEA SI, msg1` : SI ← offset of the msg1 string.

Step 1 / 9

**VARIABLES**

`SI = msg1` `CX = 0000h` `AX = 0000h` `SP = FFFEh`

**CALL STACK**

`main`

**OUTPUT**

EXAMPLE PROGRAM · STEP THROUGH THE CODE

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
▶ 4  CALL    print_string    ;
5  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

`CALL print_string` : the return address is pushed (SP = FFFCh); the procedure prints the string at DS:SI up to the 0.

Step 2 / 9

**VARIABLES**

`SI = msg1` `CX = 0000h` `AX = 0000h` `SP = FFFCh`

**CALL STACK**

- **print_string**
- main

**OUTPUT**

```
Enter the number:
```

EXAMPLE PROGRAM · STEP THROUGH THE CODE

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
▶  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

`CALL scan_num` : the user types **-25** and presses Enter; the keys are echoed.

Step 3 / 9

**VARIABLES**

`SI = msg1` `CX = 0000h` `AX = 0000h` `SP = FFFCh`

**CALL STACK**

- `scan_num`
- `main`

**OUTPUT**

```
Enter the number: -25
```

EXAMPLE PROGRAM · STEP THROUGH THE CODE

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
▶  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

On return: **CX = FFE7h** = −25 (two's complement). SP is back at FFFEh.

Step 4 / 9

**VARIABLES**

`SI = msg1` `CX = FFE7h` `AX = 0000h` `SP = FFFEh`

**CALL STACK**

`main`

**OUTPUT**

```
Enter the number: -25
```

EXAMPLE PROGRAM · STEP THROUGH THE CODE

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
5  CALL    scan_num        ; get number in CX.
▶  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

`MOV AX, CX` : PRINT_NUM expects the number in AX.

Step 5 / 9

**VARIABLES**

SI = **msg1** | CX = **FFE7h** | AX = **FFE7h** | SP = **FFFEh**

**CALL STACK**

**main**

**OUTPUT**

```
Enter the number: -25
```

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
5  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
▶  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

`CALL pthis` : the “return address” it pushes is really **the address of the string** on the next line.

Step 6 / 9

**VARIABLES**

`SI = msg1` `CX = FFE7h` `AX = FFE7h` `SP = FFFCh`

**CALL STACK**

- **pthis**
- main

**OUTPUT**

```
Enter the number: -25
```

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
5  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
▶  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
10 RET                     ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

PTHIS prints the string: 13, 10 (newline) and “You have entered: ”. Then it moves the return address past the 0.

Step 7 / 9

**VARIABLES**

SI = **msg1** | CX = **FFE7h** | AX = **FFE7h** | SP = **FFFCh**

**CALL STACK**

- **pthis**
- main

**OUTPUT**

```
Enter the number: -25
You have entered:
```

# Code Trace: Read and Print a Number

```
 1 include 'emu8086.inc'
 2 ORG    100h
 3 LEA    SI, msg1        ; ask for the number
 4 CALL   print_string    ;
 5 CALL   scan_num        ; get number in CX.
 6 MOV    AX, CX          ; copy the number to AX.
 7 CALL   pthis           ; print the following string:
 8 DB  13, 10, 'You have entered: ', 0
 9 CALL   print_num       ; print number in AX.
10 RET                    ; return to operating system.
11 msg1   DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS   ; required for print_num.
16 DEFINE_PTHIS
17 END                    ; directive to stop the compiler.
```

Execution returns **past** the string. `CALL print_num` : AX = FFE7h is printed as signed.

Step 8 / 9

**VARIABLES**

`SI = msg1` `CX = FFE7h` `AX = FFE7h` `SP = FFFCh`

**CALL STACK**

- `print_num`
- `main`

**OUTPUT**

```
Enter the number: -25
You have entered: -25
```

# Code Trace: Read and Print a Number

```
1  include 'emu8086.inc'
2  ORG     100h
3  LEA     SI, msg1        ; ask for the number
4  CALL    print_string    ;
5  CALL    scan_num        ; get number in CX.
6  MOV     AX, CX          ; copy the number to AX.
7  CALL    pthis           ; print the following string:
8  DB  13, 10, 'You have entered: ', 0
9  CALL    print_num       ; print number in AX.
▶ RET                      ; return to operating system.
11 msg1    DB  'Enter the number: ', 0
12 DEFINE_SCAN_NUM
13 DEFINE_PRINT_STRING
14 DEFINE_PRINT_NUM
15 DEFINE_PRINT_NUM_UNS    ; required for print_num.
16 DEFINE_PTHIS
17 END                     ; directive to stop the compiler.
```

`RET` : 0000h is popped into IP, SP = 0000h; back to DOS. Lines 11–17 are data and procedure definitions, never executed in this flow.

Step 9 / 9

**VARIABLES**

SI = **msg1** | CX = **FFE7h** | AX = **FFE7h** | SP = **0000h**

**CALL STACK**

**main**

**OUTPUT**

```
Enter the number: -25
You have entered: -25
```

EXAMPLE PROGRAM

# How PTHIS Works

code segment (memory, addresses increase to the right)




stack: 


← SP = FFFEh

In memory, the string bytes follow `CALL pthis` directly; the string ends in 0.

Step 1 / 5

The trick relies on the return address being kept on the stack. The PRINT macro also embeds data in the code, but jumps over it with `JMP` .

EXAMPLE PROGRAM

# How PTHIS Works

code segment (memory, addresses increase to the right)




stack: **addr(0Dh)** ← SP = FFFCh

`CALL` pushes the address of the "next instruction", which is really the address of **0Dh**, the first byte of the string.



The trick relies on the return address being kept on the stack. The PRINT macro also embeds data in the code, but jumps over it with `JMP`.

# How PTHIS Works

code segment (memory, addresses increase to the right)




stack: `addr(0Dh)` ← SP = FFFCh

PTHIS reads that address from the stack and prints bytes from there up to the 0.



The trick relies on the return address being kept on the stack. The PRINT macro also embeds data in the code, but jumps over it with `JMP`.

# How PTHIS Works

code segment (memory, addresses increase to the right)




stack: `addr(CALL print_num)` ← SP = FFFCh

The address on the stack is replaced by the one just past the 0 byte: it now points to `CALL print_num`.



The trick relies on the return address being kept on the stack. The PRINT macro also embeds data in the code, but jumps over it with `JMP`.

EXAMPLE PROGRAM

# How PTHIS Works

code segment (memory, addresses increase to the right)




stack:  ← SP = FFFEh



`RET` : the CPU **skips** the string and continues at `CALL print_num` . The data is never executed as code.

Step 5 / 5

The trick relies on the return address being kept on the stack. The PRINT macro also embeds data in the code, but jumps over it with `JMP` .

EXAMPLE PROGRAM

# Example: Adding Two Numbers

```
include 'emu8086.inc'
ORG  100h
CALL scan_num      ; 1st number → CX
MOV  BX, CX        ; save it
PUTC 13
PUTC 10            ; next line
CALL scan_num      ; 2nd number → CX
PUTC 13
PUTC 10
MOV  AX, BX
ADD  AX, CX        ; signed sum
JO   overflow
CALL print_num
RET
overflow: PUTC '?'
RET
DEFINE_SCAN_NUM
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS
END
```

- The second SCAN_NUM also returns in CX, so the first number is saved in BX.
- Enter only returns to the line start; `PUTC 13`, `PUTC 10` move the next input down a line.
- `JO` catches signed overflow: 30,000 + 5000 = 35,000 exceeds the 16-bit signed range (32,767).

| Inputs | Screen (last line) |
|---|---|
| 120, -45 | 75 |
| -7, -8 | -15 |
| 30000, 5000 | ? |

EXAMPLE PROGRAM

# Example: A Framed Title

```
include 'emu8086.inc'
ORG  100h
CALL clear_screen
GOTOXY 10, 2
CALL hline
GOTOXY 10, 3
PRINT '|  HEADER  |'
GOTOXY 10, 4
CALL hline
GOTOXY 0, 6       ; cursor below
RET
hline PROC        ; +----------+
PUTC '+'
MOV  CX, 10
dash: PUTC '-'
LOOP dash
PUTC '+'
RET
hline ENDP
DEFINE_CLEAR_SCREEN
END
```



columns 10 … 21

- Our own procedure ( `hline` ) works alongside the library macros; it is defined after RET.
- The `PUTC '-'` inside the loop is expanded once and runs 10 times: the code is not copied 10 times.
- PUTC uses only AX and restores it; it never touches CX, so the loop counter is safe.

EXAMPLE PROGRAM

# Common Mistakes

### Forgetting DEFINE_

`CALL scan_num` without `DEFINE_SCAN_NUM`: the assembler reports an *undefined* label.

### Where DEFINE_ goes

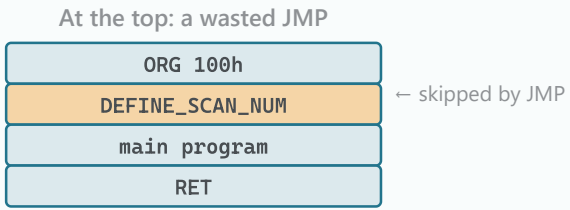
Every DEFINE_ macro starts with `JMP skip_proc_…`, so even at the top of the program the CPU skips the body. The usual place is still between `RET` and `END`.

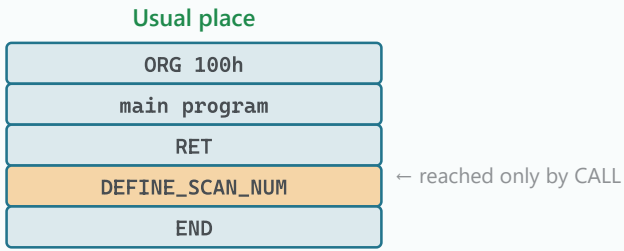
### Missing the 0 terminator

PRINT_STRING and PTHIS print until they see a 0; without one, random memory bytes spill onto the screen.

### Assuming registers survive

Procedures return results in specific registers (SCAN_NUM → CX). Don't assume any other register is preserved unless the documentation says so.



EXAMPLE PROGRAM

# Number and Screen Mistakes

### Looking for the result in AX

SCAN_NUM returns in CX. Forget `MOV AX, CX` before PRINT_NUM and the stale AX is printed.

### Printing unsigned as signed

AX = 40,000 (9C40h) shows up as −25536 with PRINT_NUM. Unsigned values that can exceed 32,767 need PRINT_NUM_UNS.

### Expecting a new line after Enter

After SCAN_NUM the cursor is at the start of the input line, so the next output overwrites the number. Put CR + LF in between.

### Mixing up row and column

`GOTOXY 5, 10` moves the cursor to column 5, row 10, not the other way round.

### Passing a register the expansion clobbers

`GOTOXY AL, AH`: the macro loads 02h into AH first, so the row becomes 02h. Move the position into other registers first.

### Overstating the buffer size

If DX exceeds the real buffer size, GET_STRING overwrites the variables that follow the buffer.

# Lab: The emu8086.inc Library

Screen output with macros (PRINT, GOTOXY, PUTC)

CODE · **69 bytes** Expansion

```
  1 include emu8086.inc
  2 ORG    100h
  3 PRINT 'Hello World!'
+ PUSH    AX
+ PUSH    SI
+ JMP     ??0003
+ ??0001 DB 'Hello World!', 0
+ ??0003:
+ LEA     SI, ??0001
+ ??0000:
+ MOV     AL, CS:[SI]
+ CMP     AL, 0
+ JZ      ??0002
+ INC     SI
```

REGISTERS

| | | | | | |
|---|---|---|---|---|---|
| AX | 0000 | AH 00 | AL 00 | | |
| BX | 0000 | BH 00 | BL 00 | | |
| CX | 0045 | CH 00 | CL 45 | | |
| DX | 0000 | DH 00 | DL 00 | | |
| SI | 0000 | DI 0000 | BP 0000 | | |
| SP | FFFE | CS 0700 | IP 0100 | | |
| DS | 0700 | ES 0700 | SS 0700 | | |

FLAGS

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

CS:IP = 0700:0100 · step 0

MEMORY **DS = 0700h**

```
0100 50 56 EB 0D 48 65 6C 6C
0108 6F 20 57 6F 72 6C 64 21
0110 00 BE 04 01 2E 8A 04 3C
```

SCREEN

Program loaded: 69 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (0045h). The Step button executes instructions one at a time; the highlighted line is the next instruction.

SUMMARY

# Summary

**INCLUDE**

`include 'emu8086.inc'`: searched in the source folder first, then in the Inc folder.

**Macros**

PUTC, GOTOXY (col, row), PRINT, PRINTN, CURSOROFF, CURSORON: built on INT 10h, expanded wherever they are used.

**Procedures**

Define with `DEFINE_...` before END, call with `CALL`. Input/output registers: SI, DI, DX, CX, AX.

**Numbers**

SCAN_NUM → CX (signed); PRINT_NUM prints AX as signed, PRINT_NUM_UNS as unsigned.

**PTHIS**

The string sits right after the CALL; the procedure moves the return address past it.

**Strings**

PRINT embeds the string in the code; PRINT_STRING and PTHIS print 0-terminated strings with one copy of the code; GET_STRING ends its buffer with 0.

SUMMARY

# Exercises

1. With macros, print “CENTER” mid-screen (column 35, row 12).
2. Read two numbers from the user and print their sum.
3. If `PRINT` is used 10 times, how many copies of the strings and loop code does the program hold? Compare with PRINT_STRING.
4. What do PRINT_NUM and PRINT_NUM_UNS print when AX = 8000h?
5. What happens if `DEFINE_PTHIS` is placed right below `ORG 100h`?
6. Read the user's name with GET_STRING and print “Hello, name!”.

▶ **Hints / answers**

# References


- emu8086 help: the “emu8086.inc — library of common functions” and “Macros” pages.
- The `Inc/emu8086.inc` file in the emu8086 installation (source code of the macros and procedures).
- B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — procedures and macros.
- A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Pearson, 2008.

# The End

## Chapter 9.2: The emu8086.inc Library

Previous: Chapter 9.1: Macros · Next: Chapter 10.1: 8086 Pins