ACEGALS - Assembly Comments Explained: Guide for Advanced Learning and Style

This document has been compiled on 2021-05-17.

Contents

Section 1: Protocol comments

These comments list various aspects of a function's protocol. While the description can be extensive, sometimes parts are omitted and left to implication. (Example 14.1, example 14.2, example 14.3, example 14.4, example 14.5.)

1.1 Protocol comment sections

Protocol comments are composed of several sections, all of which are optional. If there is a section name given, then the contents of that section are indented the same as the section name, with a semicolon in the same column, and are then indented by an additional tab.

1.1.1 Header (no section name given).

English-language description of the function's purpose. (Example 14.1, example 14.3.)

1.1.2 ‘`INP:`’ Input parameters.

These can be given as variables (labels in memory), flags or flag states, registers, and other circumstances (described in English). Usually a concise explanation of the given input is written down along with it. Equality expressions describe the content of the input here.

1.1.3 ‘`OUT:`’ Output parameters.

These can again be variables, flags, flag states, registers, or other circumstances. Sometimes the latter will be descriptions of the resulting behaviour of the function. Equality expressions are to be read as assignments to the left-hand operand.

1.1.4 ‘`CHG:`’ Changed registers.

This lists registers and sometimes variables that are possibly overwritten by the function. It is not a given that they are indeed written, just that they may be. It is however a given, if a CHG section is specified, that registers not listed in it and also not listed in an OUT section are not modified by the function. Flags in the flags register are usually implied to be changed, and not listed here specifically.

Sometimes a minus sign (`-`) is given to indicate no changes occur beyond those to registers listed in the OUT section. Even if OUT only lists a register under certain conditions, it is assumed that for all other conditions this register may be changed too. (Example 14.1, example 14.3, example 14.5.)

Occasionally, it may be specified in the CHG section that flags do not change, for example, by specifying ‘`CHG: - (not even flags)`’.

1.1.5 ‘STT:’ State expected or established by the function.

This is often segment registers, and occasionally flags, but can also be general program logic state. Sometimes there will be lines like ‘`sets es to ss`’ which point out that the specified action may occur. This should be seen as advisory, as these changes are not mandatory, they merely can occur. If a state change is expected as part of an interface, it should be specified in the OUT section.

STT UP (Direction Flag cleared) is often assumed without being noted. Care must be taken not to leave DN when returning or calling a function.

1.1.6 ‘STK:’ Stack usage.

This occurs seldom. It lists the stack usage, in words, of the function. It may list several quotes for different circumstances. Hardware interrupts and CPU exceptions are not observed by these counts, but function and software interrupt calls are. (Example 14.3.)

1.1.7 ‘REM:’ or ‘Note:’ Further remarks or notes.

(Sometimes with no section name given.) Explain further details of the function's behaviour. This can range from listing side-effects to assumptions to implementation details. (Example 14.1.)

1.2 Register names and variables

All the register names can be given to indicate single registers. An ‘`h`’ after a usual 32-bit register name indicates the high 16 bits of the 32-bit register, e.g. `eaxh`. Two or more register names can be given consecutively, separated by colons. This means the registers are appended to represent a larger variable. For example, `cs:ip` refers to a 32-bit (16-bit segment + 16-bit offset) far pointer, `dx:ax` refers to a 32-bit variable, and `dl:ax` refers to a 24-bit variable. (Example 14.1, example 14.2, example 14.4.)

To refer to memory variables usually the address is set in brackets, so as to indicate an effective address. This can be prefixed by a variable size keyword, such as `byte`, `word`, `3byte`, `dword`, etc. These can be shortened to `by` or `b`, `wo` or `w`, `dwo` or `d`, etc. (Example 14.5.)

In the `OUT` section, registers or variables may be given with an `INP:` prefix, such as `INP:bx`. This refers to the value held at the function's entrypoint.

Section 2: Stack frame variables and parameters

The `lframe` stack frame macros of `lmacros2.mac` allow specifying parameters (`lpar`) and variables (`lvar`). Parameters may be discarded by the function or left on the stack as returned to the caller. Variables may be reserved space for by macros or by pushing onto the stack. For both, the first macro parameter specifies the size, which may be numeric, or one of the keywords `byte`, `word`, or `dword`. The second macro parameter is the label, which is then prepended by a question mark. For example, '`lvar word, variable`' defines a label called `?variable` which is the offset from bp to access a word in the stack frame. It can, once reserving space for it, be accessed like in '`mov ax, word [bp + ?variable]`'. (Example 14.4, example 14.5.)

Section 3: Pointers

A pointer can be described by listing the registers or variables that make up a near, far, or segment pointer, followed by an arrow, and then a description of what is being pointed to. For example, '`si -> input string`'. The arrow is '`->`' for near or far pointers, and '`=>`' (or '`:0 ->`') for segment pointers. If a certain register or register pair is set by an instruction, a comment after that code line may specify an arrow without any pointer register given, which implies that whichever register is set holds that pointer. An expression may be specified to hold the pointer, such as '`bp + di -> error info word`'. (Example 14.6.)

Section 4: Jump targets

A jump, usually a conditional one, may be followed by a comment indicating the jump action with an arrow, ‘- ->’. Before or after the arrow, either the jump condition or the target circumstances may be described. (Example 14.7.)

Section 5: Highlighted information

A question or condition may be followed by a trailing sequence composed of a blank and then a question mark (?). This is to highlight that there is consideration of a question there. Similarly, a prefix or suffix of an exclamation point (!) that is separated from text with a blank may be used. This highlights certain circumstances of importance. (Example 14.1, example 14.18.)

Section 6: Flag states and flags

The flag states may be indicated with the abbreviations used by the debugger to indicate the flags. These are:

Carry Flag

CY = Carry set, NC = No Carry

Zero Flag

ZR = Zero, NZ = No Zero

Direction Flag

DN = Down (DF set), UP = Up (DF clear)

Interrupt Flag

EI = Enable Interrupts, DI = Disable Interrupts

Note that `test`, `and`, `or`, and `xor` leave NC, which is often depended upon. This includes the uses of `__TEST_IMM8`, `__TEST_IMM16`, and `__TEST_OFS16_IMM8` to skip the following instructions. It is courteous in such cases, and some others, to indicate a certain flag state in a comment after the instruction that establishes the state, or after a jump that should occur with a certain state.

When using `adc` or `sbb` or `rcr` or `rcl` in the obvious ways, no comment is given that the Carry Flag is used. The same is true of the flags used by conditional instructions, which are `jcc`, `setcc` (386+), and `cmovcc` (usual for 686). The instructions `stc`, `clc`, `cmc`, `std`, `cld`, `sti`, and `cli` are also not commented solely to indicate their flag change.

Note that `mov`, `xchg`, `push`, `pop`, `lea`, `jmp`, `call`, `retn`, and `retf` do not modify any flags. Further, `inc` and `dec` modify some flags, notably the Zero Flag, but do not modify the Carry Flag.

The `iret` instruction does modify flags, by doing the equivalent of `popf`. To establish the flags set by `iret`, the word on the stack has to be accessed via a stack frame. Instead, a '`retf 2`' instruction can be used to return from an interrupt handler, but note that this does not restore the Trap Flag nor the Interrupt Flag nor the Direction Flag, which may be undesirable.

(Example 14.2, example 14.3, example 14.7.)

Section 7: Indentation, comment positioning, and spacing

Our assembly source is formatted with 8 blanks per tab. There are two tabs in front of protocol comments, and one or two tabs in front of instructions. Labels generally start in the first column, and are given their own line, except for messages and other data. Comments can occur in single lines or multi-line paragraphs between code lines, usually indented by two tabs. Comments between code lines may start in the first column instead of being indented. Long comments, including the file purpose and usage conditions header comment, may be set in '`%if 0`' blocks without any indentation. Otherwise, a comment is always indicated by a semicolon followed by a blank, unless a linebreak immediately follows the semicolon. Comments can also occur after code lines. These are indented to a certain tab stop position, which can vary from function to function. In the case that such a comment is split across several lines or the code line is too long, after-code comments can follow on the next line after the code line, which doesn't have any code on its own. Rarely, a single comment may be continued across the margins of two or more code lines.

Comments continued across subsequent lines may indent the text in these lines with an additional blank to indicate continuation. (Example 14.9, example 14.12.) That is, there may be a semicolon followed by two blanks, or a semicolon followed by a tab followed by a blank. There is also a common usage of such additional indentation in protocol comment sections other than the header and Note sections; in these cases the indentation indicates parts that are applicable when some circumstance is met, which is given on the prior less-indented line. (Example 14.1, example 14.2.)

Preprocessor directives may be indented with single blanks to indicate the nesting level of directives within conditionals. `%macro` and `%endmacro` may be indented by one tab or not at all. Preprocessor directives not otherwise mentioned start in the first column, except for when nesting is indicated by using one or more blanks as indentation. (Example 14.10, example 14.16.)

Sometimes, labels or instructions may also be indented by an additional blank, which for instructions is placed after the default indentation tab. This usually indicates that the instruction is on another 'level' in a certain sense. For example, the instruction '`push bx`' might be indented by one tab plus one blank, amidst code otherwise indented by one tab, to indicate that it is pushing into a parameter to a function call. (Example 14.18.) Another common example is when a lot of stack access happens in a code fragment, and it could be unclear which `push` corresponds to which `pop`. The additional indentation blank indicates a relation between the marked instructions. (Example 14.11.)

The `struc`, `endstruc`, `cpu`, `org`, `section`, `incbin`, `times`, `db`, `dw`, `dd`, and `align` directives are indented by one tab. `istruc` and `iend` are indented by one tab. The `at` macro call starts in the first column, its data is indented to some tab stop. Labels in a struc are on the same line as their reserve space instructions. (Example 14.13, example 14.15.)

Long lines may be split into several lines with the line continuation character, which is the backslash (\). The parts on the subsequent lines are indented to match the operand which is continued. (Example 14.16.) Longer messages or other data are split up into several directives. (Example 14.17.)

7.1 Vertical spacing

Vertical space is broken up using blank lines. Between distinct pieces of program logic, one blank line may be inserted. After an unconditional jump or a return instruction, there is generally a blank line. Longer, more distinct code fragments, including functions, may be separated by two blank lines. This is generally the longest chain of blank lines that occurs. Before the header comment's '`%if 0`' there is a blank line. Comments in '`%if 0`' blocks have a blank line each after the '`%if 0`', before the '`%endif`', and after it. (Example 14.8.) The header (if any) of a protocol comment is separated from other sections (if any) by one line that has the two indentation tabs and a semicolon, but is otherwise blank. Such a blank-comment line is also used between the last named section and the notes sections, if any. (Example 14.1, example 14.3.) There is no blank line between the end of a protocol comment and the label of the described function. Labels may generally be preceded or followed by blank lines, to indicate program logic association. A file may end in one blank line, or none.

7.2 Horizontal spacing

A comma is followed by a single blank in an instruction. Commas in data definitions may or may not be followed by a blank. Binary operators and operands in expressions, including in effective addresses within brackets, are generally separated by single blanks. The colon (`:`) used to indicate a register pair or segmented address is not preceded or followed by a blank. Unary operators are preceded by a blank, but usually not followed by one. Parentheses and brackets are surrounded by single blanks on the outsides, but there are no blanks on their insides. The exception is that expressions with multiple pairs of parentheses may have additional blanks inserted to clarify the intended precedence. However, between a single-line macro's name and its parameter's opening parenthesis there is no blank.

Section 8: Capitalisation

Instructions, directives, and macro names and calls are always all uncapitalised. Labels are usually not capitalised. Labels in structures start uncapitalised with subsequent words in the label capitalised. Defines and section names are usually set in all-caps, as may be equates. Structure names are also in all-caps. Uncapitalised suffixes like `_size` may be appended to a structure name. Hexadecimal numeric values are written in capital digits, with a noncapital '`h`' suffix. Register names are uncapitalised. `@@` label references (`@F`, `@B`, etc) and flag states (CY, NC, ZR, NZ, etc) are capitalised. After-code comments are generally not proper sentences and thus the first word is not capitalised as when starting a sentence. Comments between code lines and in protocol comments may be proper sentences and are then usually capitalised.

Section 9: Local labels

There are some common local labels. These are `.ret`, `.retn`, `.retf`, `.return`, `.loop`, `.next`, `.end`, `.done`, `.skip`, `.error`, `.fail`, `.success`, and `.` (only dot). A common nonlocal label used in conjunction with local labels is `msg`. (Example 14.18, example 14.19.)

Section 10: Control characters

In messages, CR and LF are represented as the numbers 13 and 10. Similarly, tab is represented as the number 9, and backspace as 8. A dollar sign ($) may be given as the number 36. A zero terminator is of course given as the number 0. Blanks as immediate operands are usually given as the number 32. (Example 14.20, example 14.23.)

Section 11: Skipping instructions

The constants `__TEST_IMM8`, `__TEST_IMM16`, and `__TEST_OFS16_IMM8` are defined to the respective byte strings for these instructions. These equate names refer to 16-bit code, in one of Real or Virtual 86 Mode or a 16-bit code segment in Protected Mode. They can be used to skip subsequent instructions that fit into the following 1, 2, or 3 bytes. However, note that they modify the flags register, including always setting NC. The 16-bit offset plus 16-bit immediate test instruction is not included for these purposes because it might access a word at offset 0FFFFh in a segment. Also, the `__TEST_OFS16_IMM8` as provided should only be used in 86M, to avoid accessing data beyond a segment limit. After the db instruction using one of these constants, a parenthetical remark should list which instructions are skipped. (Example 14.21, example 14.22.)

Section 12: Debugging defines

`lmacros1.mac` creates the defines `_DEBUG0` through `_DEBUG4` for debugging purposes. Their purpose may differ between projects. However, `_DEBUG4` almost always emits verbose messages to the terminal, with the occasional keypress prompt. The macro collection also defines the prefix macros `d0` to `d4`, which can be used to mark single lines as belonging to the debugging build, instead of wrapping that line into preprocessor conditionals. (Example 14.23.)

Section 13: Conditional jumps

The `jz` and `jnz` instructions are used when the Zero Flag state (Zero / Not Zero) is to be tested. For equality comparisons, the `je` and `jne` aliases are to be used instead. Likewise, `ja`, `jae`, `jb`, and `jbe` are used for comparisons, and `jc` and `jnc` for Carry Flag state (Carry / Not Carry). (Example 14.7, example 14.18, example 14.19, example 14.21.)

Section 14: Examples

14.1 Protocol comment of the lDOS boot iniload.asm read_sector

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l444

```
                    ; Read a sector using Int13.02 or Int13.42
                    ;
                    ; INP:  dx:ax = sector number within partition
                    ;       bx:0-> buffer
                    ;       (_LBA) ds = ss
                    ; OUT:  If unable to read,
                    ;        ! jumps to error instead of returning
                    ;       If sector has been read,
                    ;        dx:ax = next sector number (has been incremented
                    ;        bx:0-> next buffer (bx = es+word[para_per_sector
                    ;        es = input bx
                    ; CHG:  -
                    ; STT:  ds = ss
                    ;
                    ; Note: If error 09h (data boundary error) is returned,
                    ;        the read is done into the ldSectorSeg buffer,
                    ;        then copied into the user buffer.
read_sector:
```

14.2 Protocol comment of the lDOS boot iniload.asm clust_next

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l991

```
                    ; INP:  cx:bx = cluster (0-based)
                    ;       si:di = loaded FAT sector, -1 if none
                    ; OUT:  CY if no next cluster
                    ;       NC if next cluster found,
                    ;        dx:ax = next cluster value (0-based)
                    ;       si:di = loaded FAT sector
                    ; CHG:  cx, bx
clust_next:
```

14.3 Protocol comment of the lDOS memory.asm IsFirstUMCB?

https://hg.pushbx.org/ecm/ldos/file/ede05223b42e/memory.asm#l176

```
                    ; Compare ax to first UMCB
                    ;
```

```
                ; INP:  ax = MCB
                ; OUT:  NZ if not first UMCB (or if there's no UMCB)
                ;       ZR if first UMCB
                ; CHG:  -
                ; STK:  1 word
IsFirstUMCB?:
```

14.4 Near lframe of lDOS init.asm disp_dxax_times_cx_width_bx_size

https://hg.pushbx.org/ecm/ldos/file/ede05223b42e/init.asm#l2089

```
                ; INP:  dx:ax = numerator
                ;       cx = multiplier
                ;       bx = field width
                ; STT:  UP
                ; OUT:  displayed
        init1_insure_low_byte_not_0CCh
disp_dxax_times_cx_width_bx_size:
        lframe near
        lvar 4 + 4 + 2, buffer
        lvar 6, dividend
        lenter
         lvar word, width
         push bx
        push si
        push di
```

14.5 Near lframe with parameters, of symsnip binsrch.asm segment_to_selector

https://hg.pushbx.org/ecm/symsnip/file/9c232415d568/binsrch.asm#l1789

```
                ; INP:  word [ss:sp + 2] = segment value to access
                ;       word [ss:sp] = selector to use if in PM
                ; OUT:  one word popped from stack
                ;       word [ss:sp] = selector value to use for access
                ; CHG:  -
segment_to_selector:
        lframe near
        lpar word,      in_segment_out_selector
        lpar_return
        lpar word,      selector_to_use
        lenter
```

14.6 Pointer arrows in lDOS boot iniload.asm init_memory

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l1327

```
                ; ax => sector buffer
```

```
                ; bx => FAT buffer
                ; cx => above end of memory available for load
                ; dx => above end of memory used by us
        mov word [bp + ldMemoryTop], dx
        mov word [bp + ldLoadTop], cx
        mov word [bp + ldSectorSeg], ax

        mov ds, word [bp + lsvFATSeg]
        xor si, si      ; ds:si -> FAT buffer
        mov es, bx
        xor di, di      ; es:di -> where to move
        mov cx, 8192 >> 1
        rep movsw
        mov word [bp + lsvFATSeg], bx
```

14.7 Flag states and jump arrows in lDOS boot iniload.asm parse_partition_number

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l2077

```
.int13_retry:
        pushf
        push ax
        int 13h         ; first try
        jnc @F          ; NC, success on first attempt -->

; reset drive
        xor ax, ax
        int 13h
        jc @F           ; CY, reset failed, error in ah -->
```

14.8 Header comment with usage conditions in lDOS boot iniload.asm

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l1

```
%if 0

Loader for finishing file system booting
 by C. Masloch, 2017

Usage of the works is permitted provided that this
instrument is retained with the works, so that any entity
that uses the works is notified of this instrument.

DISCLAIMER: THE WORKS ARE WITHOUT WARRANTY.

%endif
```

14.9 Comment continued across lines in IDOS boot iniload.asm finish_relocation

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l824

```
        sub ax, cx      ; how much to relocate later
        shl cx, 1
        shl cx, 1
        shl cx, 1       ; how much to relocate first,
                        ;  << 3 == convert paragraphs to words
        retf            ; jump to relocator
```

14.10 Macro `lreserve` with additional indentation in lmacros2.mac

https://hg.pushbx.org/ecm/lmacros/file/6c19c9ef5997/lmacros2.mac#l556

```
                ; Reserve stack space for prior lvar variables.
                ;
                ; This is used after lenter and reserves the stack space
                ;  for all yet defined variables, including those defined
                ;  after the lenter usage (which aren't reserved space fo
                ;  by lenter itself).
                ;
                ; "lenter early \ lvar a \ lenter \ lvar b" is equivalent
                ;  to "lenter \ lvar a \ lreserve \ lvar b".
                ;
                ; lreserve can be used multiple times. It always uses
                ;  "lea sp, [bp - x]" to reserve space, so the space will
                ;  always be equal to the already defined variables.
                ;
                ; Warning: Using lreserve after having pushed additional
                ;  data, beyond the defined variables, will discard that
                ;  data from the stack as sp is reset.
%imacro lreserve 0.nolist
%ifnctx LENTER
 %fatal Must use lenter first
%endif
%ifn %$enter
 %fatal lenter did not set up stack frame
%endif
%assign %$lsizevariables (-%$ofs+1)&~1
%ifnidn %$bits,__BITS__
 %fatal BITS of lframe (%$bits) differs from current: __BITS__
%endif
%if %$emit
%if %$bits == 16
 %if %$ofs
```

```
        lea sp, [bp - %$lsizevariables]
 %endif
%elif %$bits == 32
 %if %$ofs
        lea esp, [ebp - %$lsizevariables]
 %endif
%elif %$bits == 64
 %if %$ofs
        lea rsp, [rbp - %$lsizevariables]
 %endif
%else
 %fatal Unknown BITS specified: %$bits
%endif
%endif
%endmacro
```

14.11 Indented related instructions in lDOS boot iniload.asm clust_to_first_sector

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l965

```
clust_to_first_sector:
        push dx
        push ax
         push dx
        mul word [bp + ldClusterSize]
        xchg bx, ax
        xchg cx, dx
         pop ax
        mul word [bp + ldClusterSize]
```

14.12 Indented function parameters in symsnip binsrch.asm move_insert_farpointer

https://hg.pushbx.org/ecm/symsnip/file/9c232415d568/binsrch.asm#l1088

```
        xchg ax, cx
        xchg bx, dx
         push word [bp + ?start_pointer + 2]
         push word [bp + ?start_pointer]
        call normalise_pointer_with_displacement_bxcx
         pop di
%if _PM
         push word [symsel1]
        call segment_to_selector
%endif
         pop es                 ; es:di -> ?start_pointer + where to star
                                ;  (source entry)
```

14.13 Initial macros and a structure in lDOS boot iniload.asm

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l16

```
%assign __lMACROS1_MAC__DEBUG_DEFAULTS 1
%include "lmacros3.mac"
        numdef DEBUG5
%idefine d5 _d 5,

        struc BS
bsJump: resb 3
bsOEM:  resb 8
bsBPB:
        endstruc
```

14.14 Indentation of dw and macro call in lDOS boot iniload.asm ldos_entry

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l1121

```
%assign num 1020-($-$$)
%warning num bytes in front of ldos_entry
        _fill 1020,38,start
        dw "lD"         ; always this signature (word [1020] == 446Ch)
        dw _INILOAD_SIGNATURE
                        ; two printable non-blank ASCII characters
                        ; (ie both bytes in the range 21h..7Eh)
                        ;  Rx = RxDOS kernel
                        ;  FD = FreeDOS kernel
                        ;  TP = TestPL
                        ;  (lD)eb = lDebug
                        ;  (lD)Db = lDDebug
%if ($ - $$) != 1024
 %error Invalid signature
%endif
ldos_entry:
```

14.15 Structure istruc usage and indentation in lDOS init.asm init0_end

https://hg.pushbx.org/ecm/ldos/file/ede05223b42e/init.asm#l292

```
        align 16
init0_end:
        istruc MCB
at mcbSignature,        db "M"
at mcbOwner,            dw 8
at mcbSize,             dw (dosentry_end - dosentry_start) >> 4
at smcbName,            dw "S"
at smcbType,            db S_DOSENTRY
        iend
```

14.16 Line continuation with backslashes in lDOS boot iniload.asm MZ EXE header

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l177

```
%ifndef _IMAGE_EXE_MIN_CALC
 %define _IMAGE_EXE_MIN_CALC    \
                (((_IMAGE_EXE_MIN \
                - (payload.actual_end - payload) \
                - 256 \
                + _IMAGE_EXE_AUTO_STACK) + 15) & ~15)
%endif
```

14.17 Data continued across several lines in lDebug debug.asm cmdlist

https://hg.pushbx.org/ecm/ldebug/file/89fa92b47966/source/debug.asm#l173

```
        align 2
cmdlist:        dw aa,bb,cc,ddd,ee,ff,gg,hh,ii,error,error,ll,mm,nn,oo
                dw pp,qq,rr,sss,tt,uu,error,ww,xx,yy
```

14.18 Common local label names in lDebug run.asm bb_handlefailedrestore

https://hg.pushbx.org/ecm/ldebug/file/89fa92b47966/source/run.asm#l1894

```
bb_handlefailedrestore:
        xor di, di
        xor cx, cx
        push ss
        pop es
        mov si, b_bplist.bp
.loop:
        mov ax, word [bp + di]
        test ah, 7Fh            ; failed ?
        jz .next
        push cx
        push di
                                ; cx = 0-based index
         push word [si + 2]
         push word [si]         ; stack: linear address
        mov bl, byte [si + BPSIZE - 1]
                                ; bl = what we tried to restore
        mov bh, 40h             ; bh = 40h (bb)
        or ah, 80h              ; ah & 80h = set (is restore)
        call display_breakpoint_failure
        pop di
        pop cx
.next:
        add si, BPSIZE
```

```
        scasw                   ; di += 2
        inc cx
        cmp cx, _NUM_B_BP
        jb .loop
.end:
        retn
```

14.19 Common local label name in lDOS boot iniload.asm disp_error

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l375

```
disp_error:
.:
        lodsb
        test al, al
        jz .ret
        mov ah, 0Eh
        mov bx, 7
        ; push bp
                ; (call may change bp, but it is not used here any longer
        int 10h
        ; pop bp
        jmp short .
```

14.20 Control characters given in numeric form for lDOS init.asm init1_msg

https://hg.pushbx.org/ecm/ldos/file/ede05223b42e/init.asm#l702

```
init1_msg:
.alloc_error:           asciz "init: Error "
.alloc_error_q:         asciz "h while allocating memory: "
.linebreak:             asciz 13,10
```

14.21 Skipping instructions in lDOS boot iniload.asm init_memory

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l1262

```
        mov dx, ss
        mov ax, bp
        add ax, 512 + 15
        jnc @F
        mov ax, 1_0000h >> 1
        db __TEST_IMM16 ; (skip one shr)
@@:
        shr ax, 1
        shr ax, 1
        shr ax, 1
        shr ax, 1
        add dx, ax
```

14.22 Skipping instructions using NC setting in lDebug boot.asm yy_boot_read

https://hg.pushbx.org/ecm/ldebug/file/89fa92b47966/source/boot.asm#l4319

```
.success:
        db __TEST_IMM8          ; (skip stc, NC)
.error:
        stc
```

14.23 Using d5 prefix in lDOS boot iniload.asm parse_partition_number

https://hg.pushbx.org/ecm/ldosboot/file/5815f5ba238b/iniload.asm#l1779

```
d5      call d5dumpregs
d5      call d5message
d5      asciz 13,10,"In query_geometry 0",13,10
```

Source Control Revision ID

hg 08ae867c7a76, from commit on at 2021-05-17 16:51:38 +0200

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/acegals/rev/08ae867c7a76