LAMPIRAN C
Listing Program

Listing Program *Hardware M.P.D Translator* C-1

```
d;################################################
;##                                              ##
;##  Project Name : M.P.D Translator             ##
;##        Author : Jonathan Chandra a.k.a JoC   ##
;##   Create Date : 20:57:02 29/04/2011          ##
;##                                              ##
;################################################

============================== ALIASES ========================================

; ### CONSTANTA ###

   ; Send message length maximum
   MESSAGE_SEND_MAX equ 42 ; 42 Chars maximum, by taking account of some function
limit, such as conversions[hex2int] in pdu_send
   ; (7) = NUMBER_MAX / 2
   ; 8 = Constant

   ; 16-bit mode | 99 - (42 * 2) - (7) - 8, maximum message length for 16-bit is
42(PDU), 42-chars(ASCII) [Lowest length]
   ; 7-bit/8-bit mode | 99 - (84) - (7) - 8, maximum message length for 7-bit is
84(PDU), 96-chars(ASCII) [Practically limitted by RAM]
   ; 8-bit mode | 99 - (84) - (7) - 8, maximum message length for 8-bit is
84(PDU), 84-chars(ASCII) [Practically limitted by RAM]

   ; Save message length maximum
   MESSAGE_SAVE_MAX equ 44
   ; 44(PDU) will expands to 50-chars(ASCII)

   PDU_MAX equ 25 ; [??? Process] Still limitted
   NUMBER_MAX equ 14

   MESSAGE_RAM_OFFSET equ 43h
   NUMBER_RAM_OFFSET equ 35h

; ### PORTS AND PINS ALIASES ###
; for further information, please look up the *.txt file!

   ; [BIDIRECTIONAL]
   COMM_Lines equ P1 ; -- DB25: 2~9 / DATA PORT

   ; [INPUT]
   nEN equ P2.0 ; Enable/Clock pin [inverted] -- DB25: 1 / nC0

   ; [OUTPUT]
   DR equ P2.1 ; Data_Ready pin -- DB25: 15 / S3

   ; [OUTPUT]
   Busy equ P2.2 ; Busy flag pin / Warning flag pin* -- DB25: 13 / S4

   ; [OUTPUT]
   CRC equ P2.3 ; Client_Request_Communication pin -- DB25: 12 / S5 [LED]

   ; [INPUT]
   ; This pin, will simply "reset" any occured hang/error on the program.
   ; WARNING: Not confuse with hardware-reset!
   nEL equ P2.4 ; Exit_Loop function pin. [inverted] -- DB25: 14 / nC1 [NC]

   ; [OUTPUT]
   Ready equ P2.5 ; this pin goess LOW if M.P.D is ready to use (boot time
complete)

   ; [LCD Pins]
   LCD_Lines equ P0
   LCD_LinesBit7 equ P0.7
   LCD_RS equ P3.5
   LCD_RW equ P3.6
   LCD_EN equ P3.7

; ### RAM LOCATIONS AND REGISTERS ALIASES ###
```

```
; for further information, please look up the *.txt file!

   TP_DATA_LENGTH equ R5
   TMP equ R7
   FLAG equ R6 ; Various system flags used

   FILTER_DATA equ 76h ;- FILTER_DATA / Untuk Filter Positif [Mengekstrak data]
   nFILTER_DATA equ 77h ;- NOT FILTER_DATA / Untuk Filter Negatif [Membuang data]
   Message_Length equ 78h ;- Message_Length
   Number_Length equ 79h ;- Number_Length
   Message_Index equ 7Ah ;- Message_Index
   Number_Index equ 7Bh ; - Number_Index

   ; just arrived SMS index number (+CNTI response)
   HI_SMS_INDEX equ 7Ch
   LO_SMS_INDEX equ 7Dh

   RAM_TEMP_A equ 7Eh
   RAM_TEMP_B equ 7Fh

====================== CONTROL BEGIN ========================================
org 0000h
   jmp program_begin ; Don't begin with executing interrupts routine! Jump to
program_begin ASAP!

====================== INTERRUPT HANDLER ====================================
org 0023h
   ; Interrupt Vector Address for UART Comm. (TI & RI)
   jmp serial_interrupt

====================== MAIN PROGRAM BEGIN  ==================================
org 0030h
   program_begin:
      ; ### PORTS SETUP ###
      ; > Default state
      ; (LCD)
         mov LCD_Lines, #00h
         clr LCD_EN
         setb LCD_RS
         clr LCD_RW
      ; > As Inputs
         mov COMM_Lines, #0FFh ; COMM Port Lines wait for instruction code.
         setb nEN ; wait for clock
         setb nEL ; Exit loop input mode

      ; > As Outputs
         setb READY ; (still not ready)
         clr DR ; no data
         clr BUSY ; ready.
         clr CRC ; no need for communication to host

      ; ### HARDWARE SETTINGS ###
      ; > Timers
         mov TMOD, #21h ; 8-bit (AUTO) mode for Timer1, 16-bit mode for Timer0
         ; Timer1 is used for serial baudrate generator
         ; Timer0 is for delay generator function.

      ; > Serial UART
         ; Timer1 is used for serial baudrate generator.
         ; 9600 baudrate with 11.0592 MHz
         ; [SMOD been set]
         ;     Baudrate = Crystal / (12 * 16) / T
         ; [SMOD cleared]
         ;     Baudrate = Crystal / (12 * 32) / T
         ; In this project we gonna use 9600 baudrate configuration, so the
equotation is given:
         ;     (SMOD cleared)
         ;     9600 = 11059200 / (12 * 32) / T
         ;     9600 = 28800 / T
         ;     T = 28800 / 9600
```

```
         ;     T = 3
         ;
         ; Timer1 counts 3 machine cycle, so we set reload value: 256 - 3 = 253
         mov TH1, #253
         mov TL1, #253
         ; SMOD Cleared, 1x speed
         mov A, PCON
         anl A, #01111111b
         mov PCON, A

         ; Setup UART in microcontroller
         ; UART in mode 1, combination SM0=0 SM1=1
         clr SM0
         setb SM1
         clr SM2
         setb REN ; Serial UART, Receive data enabled. This setting allow
microcontroller receive data from RXD pin.

         ; Start the UART, by starting Timer1
         setb TR1

      ; > * Interrupt
         setb EA ; enable global interrupt
         ; (Serial), look code below

      ; ### LCD BOOTING-UP ###
      ; > Wait for 50ms
         call delay_50ms
      ; > Call LCD_Init function
         call LCD_Init
      ; > Clear LCD display
         call LCD_Clear
      ; > Turn off cursor
         call LCD_Cursor_OFF

      ; ### VARIABLE INITIALIZATION ###
      ; > Init value
         mov MESSAGE_LENGTH, #0
         mov MESSAGE_INDEX, #0

         mov NUMBER_LENGTH, #0
         mov NUMBER_INDEX, #0

         mov HI_SMS_INDEX, #' '
         mov LO_SMS_INDEX, #' '

         mov FLAG, #00h ; no scheme is selected!, no flag has been set!
         ;[ (EF).(DF).(ODD).X.X.X.(SCHEME).(SCHEME)]
         ; 7 1-bit EF = Encode flag. If set then the program can proceed to send
encoded message
         ; 6 1-bit DF = Decode Flag. If set then the program can proceed to read
decoded message
         ; 5 1-bit ODD = Odd number detected
         ; 1,0 2-bit = SCHEME used. 0x01 = 7-bit, 0x02 = 8-bit, 0x03 = 16-bit

      ; ### INTRODUCTION ###
      ; > Display introduction text
         call display_header
         call LCD_goto_2nd
         call display_welcome
      ; > Delay 2 sec
         call delay_1s
         call delay_1s
      ; > Display copyright
         call display_design_by_joc
      ; > Delay 2 sec
         call delay_1s
         call delay_1s

      ; ### PROGRAM_BOOT ###
```

```
; > Checking for mobile phone availability using "AT<CR>" command
   call display_checking_mp
   call delay_1s

   mov A, #'A'
   call send_serial
   mov A, #'T'
   call send_serial
   mov A, #13 ; <CR>
   call send_serial

   ; > Wait until the UART RX get string "...OK"
   no_AT_OK_O:
   clr RI
   jnb RI, $ ; wait for 'O'
   mov A, SBUF
   cjne A, #'O', no_AT_OK_O
      clr RI
      jnb RI, $ ; wait for another data
      mov A, SBUF
      cjne A, #'K', no_AT_OK_K

         clr RI
         jnb RI, $ ; <CR>
         clr RI
         jnb RI, $ ; <LF>

         ; Mobile phone detected!
         ; Nice response

         call display_done
         call delay_1s

         jmp MP_detect_OK

      no_AT_OK_K:
   ;no_AT_OK_O:

   ;ERROR, WRONG RESPONSE
   jmp program_crash

MP_detect_OK:
call delay_1s

; > Setup MP echo disable
   clr REN ; turn off a while receive enable

   mov A, #'A'
   call send_serial
   mov A, #'T'
   call send_serial
   mov A, #'E'
   call send_serial
   mov A, #'0'
   call send_serial
   mov A, #13 ; <CR>
   call send_serial

   setb REN ; turn again receive enable

; > Display configuring MP
call display_conf_mp
call delay_1s

; > Configure storage settings

   ; > PROCESS HERE
   ;     Set Storage Message Mobile-Phone, `AT+CPMS="ME","ME","ME"<CR>`
```

```
mov A, #'A'
call send_serial
mov A, #'T'
call send_serial
mov A, #'+'
call send_serial
mov A, #'C'
call send_serial
mov A, #'P'
call send_serial
mov A, #'M'
call send_serial
mov A, #'S'
call send_serial
mov A, #'='
call send_serial
mov A, #'"'
call send_serial
mov A, #'M'
call send_serial
mov A, #'E'
call send_serial
mov A, #'"'
call send_serial
mov A, #','
call send_serial
mov A, #'"'
call send_serial
mov A, #'M'
call send_serial
mov A, #'E'
call send_serial
mov A, #'"'
call send_serial
mov A, #','
call send_serial
mov A, #'"'
call send_serial
mov A, #'M'
call send_serial
mov A, #'E'
call send_serial
mov A, #'"'
call send_serial
mov A, #13 ; <CR>
call send_serial

; > Wait for "OK" response!
wait_conf_storage_OK:
clr RI
jnb RI, $
mov A, SBUF

cjne A, #'O', no_conf_storage_O

   clr RI
   jnb RI, $
   mov A, SBUF
   cjne A, #'K', no_conf_storage_K
      ; Storage configuration ok!
      jmp conf_storage_OK
   no_conf_storage_K:

no_conf_storage_O:

; NO CARRIER response -- not implemented

jmp wait_conf_storage_OK

conf_storage_OK:
```

```
; > delay 1 second before next command
   call delay_1s

; > Configure SMS arrival warning

   ; > PLACE PROCESS HERE
   ; `AT+CNMI=2,1,0,0,0<CR>`

   mov A, #'A'
   call send_serial
   mov A, #'T'
   call send_serial
   mov A, #'+'
   call send_serial
   mov A, #'C'
   call send_serial
   mov A, #'N'
   call send_serial
   mov A, #'M'
   call send_serial
   mov A, #'I'
   call send_serial
   mov A, #'='
   call send_serial
   mov A, #'2'
   call send_serial
   mov A, #','
   call send_serial
   mov A, #'1'
   call send_serial
   mov A, #','
   call send_serial
   mov A, #'0'
   call send_serial
   mov A, #','
   call send_serial
   mov A, #'0'
   call send_serial
   mov A, #','
   call send_serial
   mov A, #'0'
   call send_serial

   mov A, #13 ; <CR>
   call send_serial

   ; > Now wait for "OK" response

   wait_for_OK_incsys:
   clr RI
   jnb RI, $
   mov A, SBUF
   cjne A, #'O', no_conf_incsys_O

      clr RI ; clear receive flag
      jnb RI, $
      mov A, SBUF
      cjne A, #'K', no_conf_incsys_K
         ; > Configuration for Incoming message is OK!
         jmp incsys_OK
      no_conf_incsys_K:

   no_conf_incsys_O:

   jmp wait_for_OK_incsys

   incsys_OK:

; > Configuration done!
```

```
         call display_done
         call delay_1s

      ; ### ENABLE INCOMING MESSAGE SYSTEM RECEIVER ###
      ; > Enable Serial RX Interrupt!
         setb ES ; enable serial interrupt TI / RI

      ; ### READY-STATE DISPLAY ###
      state_ready:

      ; > Set status READY
         clr READY

      ; > Waiting for 'exit loop' function to deactivate...
            call wait_soft_reset ; make sure the 'exit loop' function is off

      ; > State: [READY], Wait for any instruction...

         jb CRC, skip_display_ready ; if CRC pin are LOW/"No new SMS", then
display cursor

            call LCD_Clear_2nd
            call display_slash
            call LCD_Cursor_On

         skip_display_ready:

      wait_instructions:

      ;Instruction Codes and Names
      ;[01] = CLEAR ASCII_CODE STORAGE and CLEAR NUMBER STORAGE | Return: None
      ;[02] = INPUT_ASCII_CODE, <ASCII> | Return: None
      ;[03] = INPUT_RECIPIENT_NUMBER, <NUMBER> | Return: None
      ;[04] = ENCODE_MESSAGE, <MODE: 0x01(PDU 7-bit scheme) ; 0x02(PDU 8-bit
scheme) ; 0x03(PDU 16-but scheme)> | Return: None
      ;[05] = DECODE_MESSAGE (SAVE PDU CODES TOO)
      ;[06] = SEND_MESSAGE (This process cannot be done, until you ENCODE_MESSAGE
first!) | Return: None
      ;[07] = READ_MESSAGE (READ THE LAST MESSAGE ARRIVED THAT HAS BEEN DECODED) |
Return: ASCII_Length; (looping)ASCII n Items
      ;[08] = DELETE_LAST_MESSAGE (THE LAST MESSAGE ARRIVED) | Return: None
      ;[09] = GET HARDWARE MESSAGE | Return: HW_MSG, lookup table below
      ;[0A] = SET MESSAGE INDEX <HI> <LO> [DEBUG PURPOSE FUNCTION]
      ;[0B] = GET MOBILE_PHONE ATTENTION "AT" | Return: None

      ; set the COMM. PORT or COMM. Lines to be INPUT MODE.
      mov COMM_Lines, #0FFh
      ; 1st of all instructions is to wait CLOCK/ENABLE signal

      call Rise_nEN ; wait for clock

      ; ### USER INSTRUCTIONS SWITCHES ###

      call display_busy
      call LCD_Cursor_Off

      mov A, COMM_Lines ; for comparing purpose

      ; [Instruction: 01, CLEAR ASCII CODE STORAGE and CLEAR NUMBER STORAGE]
         cjne A, #01h, no_instruction_1

            ; No Parameter, skip

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY
```

```
               ; RAM contents are not really been cleared, it just set input
offset and index. And set data length

               mov MESSAGE_LENGTH, #0
               mov MESSAGE_INDEX, #0

               mov NUMBER_LENGTH, #0
               mov NUMBER_INDEX, #0

               ; Clear Encode-Flag(7) + Decode-Flag(6)
               mov A, FLAG
               clr Acc.7
               clr Acc.6
               mov FLAG, A

            clr BUSY

            ; No return value, skip

            ; Release control
            call Rise_nEN ; wait for a clock to acknowledge this instruction has
been finished
            jmp state_ready ; current execution is finished
         no_instruction_1:

      ; [Instruction: 02, INPUT_ASCII_CODE] -- for ENCODE_MESSAGE_INPUTS
            cjne A, #02h, no_instruction_2

            ; Parameter: ASCII_Code
            call Rise_nEN ; wait for a clock
            mov RAM_TEMP_A, COMM_Lines

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY

               ; check for encode flag? last data has been encoded? if yes, skip
this instruction
               mov A, FLAG
               jb Acc.7, inst_2_done ; if Acc.7 bit set, then skip

                  ; check for storage capacity... it's full / not ?
                  mov A, MESSAGE_LENGTH
                  cjne A, #MESSAGE_SEND_MAX, message_not_full
                     jmp message_full
                  message_not_full:

                  ; get RAM message index value, then add it with ADDRESS_OFFSET
value.
                  mov A, MESSAGE_INDEX
                  add A, #MESSAGE_RAM_OFFSET ; INDEX + 0x44
                  mov R1, A ; this is for indirect addressing operation

                  ; store parameter/ASCII data on RAM, address indicated at R1
                  mov A, #00h ; null data (to prevent false data)
                  mov @R1, A
                  mov A, RAM_TEMP_A ; actual data
                  mov @R1, A

                  ; next index and length is +1, stored new data
                  inc MESSAGE_INDEX
                  inc MESSAGE_LENGTH
                  mov TP_DATA_LENGTH, MESSAGE_LENGTH ; save to TP_DATA_LENGTH

                  jmp inst_2_done

                  message_full:
                  ; (OPTIONAL) show in LCD, the message capacity is overloaded!
```

```
               inst_2_done:
            clr BUSY

            ; No return value, skip

            ; Release control
            call Rise_nEN
            jmp state_ready ; current execution is finished
         no_instruction_2:

      ; [Instruction: 03, INPUT_RECIPIENT_NUMBER] -- for ENCODE_MESSAGE_INPUTS
         cjne A, #03h, no_instruction_3

            ; Parameter: Recipient_Number (ASCII)
            call Rise_nEN ; wait for a clock
            mov RAM_TEMP_A, COMM_Lines

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY

               ; check for encode flag? last data has been encoded? if yes, skip
this instruction
               mov A, FLAG
               jb Acc.7, inst_3_done ; if Acc.7 bit set, then skip

                  mov A, Number_Length
                  cjne A, #NUMBER_MAX, number_not_full
                     ; capacity for number is full!
                     jmp number_full
                  number_not_full:

                  mov A, NUMBER_INDEX
                  add A, #NUMBER_RAM_OFFSET
                  mov R1, A ; this is for indirect addressing operation

                  mov A, RAM_TEMP_A ; save 'number' data at @R1
                  mov @R1, A

                  ; increment index, and increment length
                  inc NUMBER_INDEX
                  inc NUMBER_LENGTH

                  ; clear / set odd flag (if (length mod 2) == 1)
                  mov A, NUMBER_LENGTH
                  mov B, #2
                  div AB
                  mov TMP, B
                  mov A, FLAG
                  cjne TMP, #01h, no_odd_number
                     setb Acc.5
                     jmp odd_finish
                  no_odd_number:
                     clr Acc.5
                  odd_finish:
                  mov FLAG, A

                  jmp inst_3_done

                  number_full:
                  ; (OPTIONAL) show in LCD, the number capacity is overloaded!

               inst_3_done:
            clr BUSY

            ; No return value, skip
```

```
            ; Release control
            call Rise_nEN
            jmp state_ready ; current execution is finished
         no_instruction_3:

      ; [Instruction: 04, ENCODE_MESSAGE]
         cjne A, #04h, no_instruction_4

            ; Parameter: <PDU_BIT_SCHEME>
            ;  0x01 : 7-bit
            ;  0x02 : 8-bit
            ;  0x03 : 16-bit
            call Rise_nEN
            mov TMP, COMM_LINES
            mov B, COMM_LINES
            push B ; push to STACK

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY

               mov A, TMP ; get stored parameter

               ; SWITCH parameter.

               cjne A, #01h, no_PDU_7b_scheme
                  call PDU_encode_7bit
                  jmp inst_4_done
               no_PDU_7b_scheme:

               cjne A, #02h, no_PDU_8b_scheme
                  call PDU_encode_8bit
                  jmp inst_4_done
               no_PDU_8b_scheme:

               cjne A, #03h, no_PDU_16b_scheme
                  call PDU_encode_16bit
                  jmp inst_4_done
               no_PDU_16b_scheme:

               ; if parameter is incorrect, then skip any process inside this
instruction.
               jmp cancel_inst_4

               inst_4_done:

               ; Encode number into semi-octets
               call encode_numbers

               ; set bit Encode-Flag
               ; clear Decode-flag
               mov A, FLAG
                  setb Acc.7
                  clr Acc.6
                  ; clear scheme before OR-ing
                  anl A, #11111100b
                  ; save PDU scheme used
                  pop B ; restore from STACK
                  orl A, B
               mov FLAG, A

               cancel_inst_4:

            clr BUSY

            ; No return value, skip
```

```
            ; Release control
            call Rise_nEN
            jmp state_ready ; current execution is finished
         no_instruction_4:

      ; [Instruction: 05, DECODE_MESSAGE]
         cjne A, #05h, no_instruction_5

            ; No Parameter, skip this.

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY

               ; RAM contents are not really been cleared, it just set input
offset and index. And set data length
               ; Any data residue on the RAM will be lost and replaced!

               mov MESSAGE_LENGTH, #0
               mov MESSAGE_INDEX, #0

               mov NUMBER_LENGTH, #0
               mov NUMBER_INDEX, #0

               ; clear also last decoding/encoding mode used. (because in savePDU
op, there is an OR mode),
               ; to make sure that no last mode that collide with other new mode.
(clear at once).
               ; default state for Odd-flag(5) is clear
               ; set bit Decode-Flag(6)
               ; clear Encode-flag(7)
               mov A, FLAG
                  clr Acc.0
                  clr Acc.1

                  clr Acc.5
                  setb Acc.6
                  clr Acc.7
               mov FLAG, A

               ; Fetch message from mobile phone! using AT-Command.
               ; AT+CMGR=<index>

               mov A, #'A'
               call send_serial
               mov A, #'T'
               call send_serial
               mov A, #'+'
               call send_serial
               mov A, #'C'
               call send_serial
               mov A, #'M'
               call send_serial
               mov A, #'G'
               call send_serial
               mov A, #'R'
               call send_serial
               mov A, #'='
               call send_serial

               mov A, HI_SMS_INDEX
               cjne A, #' ', indx2_digit_decode ; check for 2nd digit value, if
not ' ' then use 2 digit
                  ; 1-digit decode operation
                  mov A, LO_SMS_INDEX
                  call send_serial
                                          jmp indx_sel_decodeok
```

```
               indx2_digit_decode:

                  mov A, HI_SMS_INDEX
                  call send_serial
                                          mov A, LO_SMS_INDEX
                                          call send_serial

               indx_sel_decodeok:

               ;this section moved to 'savePDU' subprogram
               ;mov A, #13 ; <CR>
               ;call send_serial ; confirm reading...

               ; Store Sender number in NUMBER_STORAGE
               ; Read PDU Scheme, save it in FLAG register
               ; Save PDU-Codes into RAM / MESSAGE_STORAGE
               ; Filter Codes by PDU-Scheme (only by 8-bit mode and 16-bit mode)
               call savePDU ; this function cannot be moved! / used anywhere! --
programmed to stay here ONLY!

               ; Decode the PDU-Codes by appropiate PDU scheme used, if Scheme
used is 7-bit then proceed
               mov A, FLAG
               anl A, #03h
               cjne A, #01h, no_7bit_decode ; check it is a 7-bit mode?
                  call PDU_decode_7bit ; yes, call the function
               no_7bit_decode: ; no

               ; Decode sender semi-octets number into actual number (readable)
               call decode_number

            clr BUSY

            ; No return value, skip

            ; Release control
            call Rise_nEN
            jmp state_ready ; current execution is finished
         no_instruction_5:

      ; [Instruction: 06, SEND_MESSAGE]
         cjne A, #06h, no_instruction_6

            ; No Parameter

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY

               call instruction_6

            clr BUSY

            ; No return value, skip

            ; Release control
            call Rise_nEN
            jmp state_ready ; current execution is finished
         no_instruction_6:

      ; [Instruction: 07, READ_MESSAGE]
         cjne A, #07h, no_instruction_7

            ; Parameter

            ; Wait for "execute now!" signal
            call Rise_nEN
```

```
      ; Process current instruction
      setb BUSY

         ; > Check decode process 1st before reading any message!!
         mov A, FLAG
         jnb Acc.6 , not_decoded_yet

            ; Read message, byte / byte
            call instruction_7

         not_decoded_yet:

      clr BUSY

      ; No return value from function, skip

      ; Release control
      call Rise_nEN
      jmp state_ready ; current execution is finished
   no_instruction_7:

; [Instruction: 08, DELETE_LAST_MESSAGE]
   cjne A, #08h, no_instruction_8

      ; Parameter

      ; Wait for "execute now!" signal
      call Rise_nEN

      ; Process current instruction
      setb BUSY

         call instruction_8

      clr BUSY

      ; No return value, skip

      ; Release control
      call Rise_nEN
      jmp state_ready ; current execution is finished
   no_instruction_8:

; [Instruction: 09, GET HARDWARE MESSAGE]
   cjne A, #09h, no_instruction_9

      ; No Parameter, skip this

      ; Wait for "execute now!" signal
      call Rise_nEN

      ; Process current instruction
      setb BUSY

         nop
         nop
         nop

      clr BUSY

      ; Return value: <HARDWARE_MESSAGE>
         ; Set COMM. PORT Lines a HARDWARE MESSAGE.
         ; for this project, we only use 1.

         mov COMM_Lines, #00h
         setb DR ; set DATA_READY HIGH

         jnb CRC, no_hw_message

            mov COMM_Lines, #01h
```

```
                  ;======== HARDWARE MESSAGE TABLE ================================
                  ; 0x01 : New Message Arrived!

                  clr CRC ; host already know what message what we're sending

               no_hw_message:

               call Rise_nEN ; acknowledge data from host, by sending a clock
               clr DR ; clear DATA_READY LOW

               mov COMM_Lines, #0FFh ; set to INPUT MODE AGAIN

            ; Release control
            call Rise_nEN
            jmp state_ready ; current execution is finished
         no_instruction_9:

      ; [DEBUG PURPOSE ONLY]
      ; [Instruction: 0A, INPUT/REPLACE NEW MESSAGE INDEX]
         cjne A, #0Ah, no_instruction_10

            ; Parameter: HI_Index (ASCII)
            call Rise_nEN
            mov RAM_TEMP_A, COMM_Lines

            ; Parameter: LO_Index (ASCII)
            call Rise_nEN
            mov RAM_TEMP_B, COMM_Lines

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
            setb BUSY

               clr CRC ; replace, so clear CRC

               mov A, RAM_TEMP_A
               cjne A, #'0', no_space_hi
                  mov RAM_TEMP_A, #' '
               no_space_hi:

               mov A, RAM_TEMP_B
               cjne A, #'0', no_space_lo
                  mov RAM_TEMP_B, #' '
               no_space_lo:

               mov HI_SMS_INDEX, RAM_TEMP_A
               mov LO_SMS_INDEX, RAM_TEMP_B

            clr BUSY

            ; No return value, skip

            ; Release control
            call Rise_nEN ; wait for a clock to acknowledge this instruction has
been finished
            jmp state_ready ; current execution is finished
         no_instruction_10:

      ; [Instruction: 0B, GET MOBILE_PHONE ATTENTION]
         cjne A, #0Bh, no_instruction_11

            ; Parameter: NONE

            ; Wait for "execute now!" signal
            call Rise_nEN

            ; Process current instruction
```

```
            setb BUSY

               ; Send "AT<CR>" command

               mov A, #'A'
               call send_serial
               mov A, #'T'
               call send_serial
               mov A, #13 ; <CR>
               call send_serial

               call delay_1s ; let DCE/Mobile phone clear it's TX buffer

            clr BUSY

            ; No return value, skip

            ; Release control
            call Rise_nEN ; wait for a clock to acknowledge this instruction has
been finished
            jmp state_ready ; current execution is finished
         no_instruction_11:

      jmp state_ready ; loop to ready state, until the program found a matching
execution code!

      call delay_forever ; This stop the CPU to going further. (Avoid program
misbehave).

   program_crash:
      call delay_1s ; Give some delay, so this gonna tell the user when they reset
the device

      call display_error
      setb READY ; NOT RESPONDING

      call delay_forever ; This stop the CPU to going further. (Avoid program
misbehave).

   program_end:

======================== INTERRUPT PROGRAM ====================================
; ******* Serial interrupt if there are a Transmit / Receive operation called
*******
   serial_interrupt:
      ; save any ACC/PSW data
      push Acc
      push PSW

      ; Check for new message!
      ; parse the data if data is correctly received "XX+CMTI: XXXX,ABXX"

      jb TI, end_serial_interrupt ; if TI / transmit interrupt are set then goto
end_serial_interrupt ( which is not processed by this interrupt function )

         ; This space is for RI interrupt (receive only interrupt)

         clr RI
         mov A, SBUF
         cjne A, #'+', no_si_1
            jnb RI, $ ; wait for next data
            clr RI
            mov A, SBUF
            cjne A, #'C', no_si_2
               jnb RI, $ ; wait for next data
               clr RI
               mov A, SBUF
               cjne A, #'M', no_si_3
                  jnb RI, $ ; wait for next data
                  clr RI
```

```
                  mov A, SBUF
                  cjne A, #'T', no_si_4
                     jnb RI, $ ; wait for next data
                     clr RI
                     mov A, SBUF
                     cjne A, #'I', no_si_5
                        jnb RI, $ ; wait for next data
                        clr RI
                        mov A, SBUF
                        cjne A, #':', no_si_6
                           jnb RI, $ ; wait for next data
                           clr RI
                           mov A, SBUF
                           cjne A, #' ', no_si_7
                              jnb RI, $ ; wait for next data, X
                              clr RI
                              jnb RI, $ ; wait for next data, X
                              clr RI
                              jnb RI, $ ; wait for next data, X
                              clr RI
                              jnb RI, $ ; wait for next data, X
                              clr RI

                              jnb RI, $ ; wait for next data, ','
                              clr RI
                              mov A, SBUF
                              cjne A, #',', no_si_8
                                 ; the next 2 data is A and B
                                 ; A is the HI Index Number
                                 ; B is the LO Index Number

                                 ; SAVE NEW SMS INDEX NUMBER
                                 call save_sms_index

                                 ; Set the CRC/Client_Request_Communication pin
HIGH

       setb CRC

                                 ; (OPTIONAL)
                                 call display_new_sms
                                 call LCD_Cursor_OFF

                              no_si_8:

                           no_si_7:
                        no_si_6:
                     no_si_5:
                  no_si_4:
               no_si_3:
            no_si_2:
         no_si_1:


      end_serial_interrupt:

      ; clear serial flags that included in this interrupt
      ;clr TI ; do not intefere with this TI flag! cause some malfunction in
system!
      clr RI

      ; restore any ACC/PSW data
      pop PSW
      pop Acc
   reti

============================ SUBPROGRAM ========================================
   savePDU:
      ; going here from "Decode Process"
      mov A, #13 ; <CR>
```

```
      call send_serial ; confirm reading...

      clr EA ; disable global interrupt

         ; [READING RESPONSE EXAMPLE]
         ;
         ; <CR><LF>
         ; +CMGR: 1,,32<CR><LF>
         ; 06 912618010000 24 0D 91 261823228877F9 00 08 11:60:01/71:91:63G82 0C
0052,0069,0066,006C,0065,003F<CR><LF>
         ; <CR><LF>
         ; OK<CR><LF>
         ;

         ; [ 'Line Feed' checkpoint-1 ]

            LF_chk1:
               clr RI
               jnb RI, $
               mov A, SBUF
            cjne A, #10, LF_chk1

         ; [ 'Line Feed' checkpoint-2 ]

            LF_chk2:
               clr RI
               jnb RI, $
               mov A, SBUF
            cjne A, #10, LF_chk2

         ; [ Length of the SMSC information ]

            ; (HIGH-order byte)
            clr RI
            jnb RI, $
            mov A, SBUF

            ; convert to value
            call convert_to_hex

            ; shift from LOW-order output from converter to HIGH-order
               RL A
               RL A
               RL A
               RL A

            mov B, A

            ; (LOW-order byte)
            clr RI
            jnb RI, $
            mov A, SBUF

            ; convert to value
            call convert_to_hex

            ; set as a pair, combining HIGH and LOW byte
            orl A, B

            ; > before do skips, check the value first! if 0x00 then cancel to
skips data
            cjne A, #00h, do_skips_smsc
               jmp stop_smsc
            do_skips_smsc:

            ; we gonna skip <value> octets (2-ASCII data) , not a single ASCII, so
multiply it by 2
            mov B, #2
            mul AB
```

```
   mov TMP, A

   skip_smsc:
      clr RI
      jnb RI, $
   djnz TMP, skip_smsc

   stop_smsc:

; [ First octet of this SMS-DELIVER message. ]
   ; Ignored

   clr RI
   jnb RI, $

   clr RI
   jnb RI, $

; [ Address-Length. Length of the sender number ]

   ; get HIGH-byte order
   clr RI
   jnb Ri, $
   mov A, SBUF

   ; convert to hex first
   call convert_to_hex

   ; convert from LOW-order converter output to HIGH-order value
      RL A
      RL A
      RL A
      RL A

   ; save to B
   mov B, A

   ; get LOW-byte order
   clr RI
   jnb RI, $
   mov A, SBUF

   ; convert to hex first
   call convert_to_hex

   ; combine HIGH-order and LOW-order value
   orl A, B

   ; save to TMP
   mov TMP, A

   ; > Check for ODD
   mov B, #2
   div AB
   mov A, B ; place modulus from B to A
   ; compare, if modulus_result = 1 then Odd.
   cjne A, #01h, no_odd_read
      ; Odd
      ; There is a trailing data 'F'
      ; so add 1 length to read
      inc TMP

      ; turn on Odd flag
      mov A, FLAG
      setb Acc.5
      mov FLAG, A

   no_odd_read:

; [ Type-of-Address. ]
```

```
   ; Ignored

   clr RI
   jnb RI, $

   clr RI
   jnb RI, $

; [ The phone number in semi octets ]

   read_number:

      ; indirect-addressing
      mov A, NUMBER_INDEX
      add A, #NUMBER_RAM_OFFSET
      mov R1, A

      ; wait and get data
      clr RI
      jnb RI, $
      mov A, SBUF

      ; store data at pointed index
      mov @R1, A

      inc NUMBER_INDEX
      inc NUMBER_LENGTH

   djnz TMP, read_number

; [ TP-PID. Protocol identifier. ]
   ; Ignore

   clr RI
   jnb RI, $

   clr RI
   jnb RI, $

; [ TP-DCS Data coding scheme ]

   ; HIGH-order byte -- ignore this.
   clr RI
   jnb RI, $

   ; LOW-order byte
   clr RI
   jnb RI, $
   mov A, SBUF

   ; convert to hex
   call convert_to_hex
   ; filter coding mode only
   anl A, #00001100b
   ; "standarize" mode to follow our plans
      RR A ; shift right 2x
      RR A

      inc A ; by incrementing the mode once

   ; temporary save to B
   mov B, A

   ; save to FLAG register
   mov A, FLAG
   orl A, B
   mov FLAG, A

; [ TP-SCTS. Time stamp (semi-octets) ]
   ; Ignore - 7 octets - 14 chars
```

```
            mov TMP, #14
            skip_tp_scts:
               clr RI
               jnb RI, $
            djnz TMP, skip_tp_scts


         ; [ TP-UDL. User data length, length of actual message. ]

            ; HIGH-order byte
            clr RI
            jnb RI, $
            mov A, SBUF
            ; convert to hex
            call convert_to_hex
            ; convert from LOW-order output from converter to HIGH-order
               RL A
               RL A
               RL A
               RL A
            ; temporary save to B
            mov B, A

            ; LOW-order byte
            clr RI
            jnb RI, $
            mov A, SBUF
            ; convert to hex
            call convert_to_hex

            ; combine HIGH-order and LOW-order value
            orl A, B

            ; [WARNING!] > Check the value before storing into register
            ; if its same as MESSAGE_SAVE_MAX or exceed it?
            push Acc

            cjne A, #MESSAGE_SAVE_MAX, tp_dl_max
               ; equal to MESSAGE_SAVE_MAX, so no problem
               pop Acc
               jmp tp_dl_end
            tp_dl_max:
               ; check, is it '<' or '>' ?
               ; '<' CY is set
               ; '>' CY is cleared

               jnc greater_tp_dl
               less_tp_dl:
                  pop Acc
                  jmp tp_dl_end
               greater_tp_dl:
                  ; force to use "total data = 45 chars"
                  ; to prevent program from overwriting RAM, especially other
variables.

                  pop Acc ; Don't care, just release from stack, because we will
replace this value
                  mov A, #MESSAGE_SAVE_MAX ; replace

            tp_dl_end:


            ; store at TP_DATA_LENGTH register
            mov TP_DATA_LENGTH, A


         ; [ TP-UD. / Message ]

            ; check decoding mode before proceeding any reading process!
```

```
            mov A, FLAG
            anl A, #03h ; filter mode only

            ; set counter
            mov TMP, #1

            ; switch mode
            cjne A, #03h, pdu_read78
               jmp pdu_read16

            pdu_read78:

               ; ======= THIS SUB-PROGRAM ONLY APPLY TO 7-bit / 8-bit ===========

               ; get HIGH-order data (1st ASCII).pair
               clr RI
               jnb RI, $
               mov A, SBUF
               ; check if it's raw value is 13 / <CR> then quit receiving anymore
data
               cjne A, #13, no_crA
                  jmp pdu_read_end
               no_crA:
               ; check if now length exceed maximum read length, then quit
               cjne TMP, #MESSAGE_SAVE_MAX, no_equal
                  jmp still_ok
               no_equal:
                  ; check for CY
                  jc still_ok
                     ; GREATER

                     ; wait for <CR> char before quitting
                     wait_cr:
                        clr RI
                        jnb RI, $
                        mov A, SBUF
                        cjne A, #13, no_crA2
                           jmp pdu_read_end
                        no_crA2:
                     jmp wait_cr
               still_ok:
               ; convert to hex (A doesn't affected at all)
               call convert_to_hex
               ; convert from LOW-order output of the converter to HIGH-order
value
                  RL A
                  RL A
                  RL A
                  RL A
               ; store HIGH-order temporary at B
               mov B, A

               ; get LOW-order data (2nd ASCII).pair
               clr RI
               jnb RI, $
               mov A, SBUF
               ; convert to hex
               call convert_to_hex

               ; combine HIGH-order and LOW-order value
               orl A, B
               ; hold result temporary at RAM
               push Acc

               ; save to message RAM by indirect-addressing
               mov A, MESSAGE_INDEX
               add A, #MESSAGE_RAM_OFFSET
               mov R1, A

               ; save at pointed location
```

```
                pop Acc ; get data back from RAM
                mov @R1, A

                ; increase index and length
                inc MESSAGE_INDEX
                inc MESSAGE_LENGTH

                ; increase counter
                inc TMP

             jmp pdu_read78 ; loop until we detect <CR>

             pdu_read16:

                ; ========== THIS SUB-PROGRAM ONLY APPLY TO 16-bit ===============

                ; skip frame 1 data '0' / compare to <CR> value to end this process
                clr RI
                jnb RI, $
                mov A, SBUF
                ; check if it's raw value is 13 / <CR> then quit receiving anymore
data
                cjne A, #13, no_crB
                   jmp pdu_read_end
                no_crB:
                ; check if now length exceed maximum read length, then quit
                cjne TMP, #MESSAGE_SAVE_MAX, no_equal2
                   jmp still_ok2
                no_equal2:
                   ; check for CY
                   jc still_ok2
                      ; GREATER

                      ; wait for <CR> before quitting
                      wait_cr2:
                         clr RI
                         jnb RI, $
                         mov A, SBUF
                         cjne A, #13, no_crB2
                            jmp pdu_read_end
                         no_crB2:
                      jmp wait_cr2
                still_ok2:
                ; skip frame 2 data '0'
                clr RI
                jnb RI, $

                ; get HIGH-order data (1st ASCII).pair
                clr RI
                jnb RI, $
                mov A, SBUF
                ; convert to hex
                call convert_to_hex
                ; convert from LOW-order output of the converter to HIGH-order
value
                   RL A
                   RL A
                   RL A
                   RL A
                ; store HIGH-order temporary at B
                mov B, A

                ; get LOW-order data (2nd ASCII).pair
                clr RI
                jnb RI, $
                mov A, SBUF
                ; convert to hex
                call convert_to_hex

                ; combine HIGH-order and LOW-order value
```

```
               orl A, B
               ; hold result temporary at RAM
               push Acc

               ; save to message RAM by indirect-addressing
               mov A, MESSAGE_INDEX
               add A, #MESSAGE_RAM_OFFSET
               mov R1, A

               ; save at pointed location
               pop Acc ; get data back from RAM
               mov @R1, A

               ; increase index and length
               inc MESSAGE_INDEX
               inc MESSAGE_LENGTH

               ; increase counter
               inc TMP

            jmp pdu_read16

            pdu_read_end:

         ; [ 'Line Feed' checkpoint-3 ]

            LF_chk3:
               clr RI
               jnb RI, $
               mov A, SBUF
            cjne A, #10, LF_chk3

         ; [ 'Line Feed' checkpoint-4 ]

            LF_chk4:
               clr RI
               jnb RI, $
               mov A, SBUF
            cjne A, #10, LF_chk4

         ; [ 'Line Feed' checkpoint-5 ]

            LF_chk5:
               clr RI
               jnb RI, $
               mov A, SBUF
            cjne A, #10, LF_chk5

      setb EA ; enable global interrupt

      call delay_1s ; let DCE/Mobile phone clear it's TX buffer

   ret

   instruction_6: ; Send Message Instruction

      ; AT-COMMAND:
      ; AT+CMGS=<pdu-length><CR>
      ; <pdu><ESC/CTRL-Z>

      ; Check encode process 1stbefore sending any message!! this actually check
encode flag
      mov A, FLAG
      jnb Acc.7, ext6_1

      ; jump extension to skip_instruction_6 begin --
      jmp ext6_0
         ext6_1: jmp skip_instruction_6
      ext6_0:
      ; -- jump extension end
```

```
      ; > Send message

         ; BASIC COMMAND

            mov A, #'A'
            call send_serial
            mov A, #'T'
            call send_serial
            mov A, #'+'
            call send_serial
            mov A, #'C'
            call send_serial
            mov A, #'M'
            call send_serial
            mov A, #'G'
            call send_serial
            mov A, #'S'
            call send_serial
            mov A, #'='
            call send_serial

            ; The first octet ("00") doesn't count, it is only an indicator of the
length of the SMSC information supplied (0).
            ; <pdu-length> in (Decimal)ASCII Format
               mov B, #8 ; data that has share same size (1 septet) is totally 8
data.
               push B

               mov A, NUMBER_LENGTH ; number length, with/without a trailing F
               mov B, #2
               div AB
               mov RAM_TEMP_A, A ; set the result to RAM_TEMP_A

               mov A, FLAG
               anl A, #00000011b
               cjne A, #03h, length_bit78

                  mov A, MESSAGE_LENGTH ; message length ( 16-bit mode only )
                  ; length * 2
                  ; this must be done like this, because we add frame "00" to each
data
                  mov B, #2
                  mul AB

                  mov RAM_TEMP_B, A ; set the result to RAM_TEMP_B

                  jmp length_end

               length_bit78:
                  ; message length ( 7-bit* / 8-bit mode only )

                  ; > Warning: Check for 0x00 data, every 8th iteraction
                  mov A, #0
                  mov B, #1
                  mov RAM_TEMP_B, MESSAGE_LENGTH
                  check_d00:
                     push Acc
                     push B

                     ; indirect-addressing
                     add A, #MESSAGE_RAM_OFFSET
                     mov R1, A

                     ; get data from RAM
                     mov A, @R1

                     ; compare the data!, if 0x00 then discard/dec by 1
                     cjne A, #00h, dont_discard
                        pop B ; - get B
```

```
                        push B ; - save B
                        mov A, B
                        cjne A, #8, dont_discard
                           ; DISCARD
                           dec RAM_TEMP_B
                           pop B
                           mov B, #0
                           push B
                     dont_discard:

                        pop B
                        inc B

                        pop Acc
                        inc A
                     cjne A, MESSAGE_LENGTH, check_d00

                     ;mov RAM_TEMP_B, A ; set the result to RAM_TEMP_B

                  length_end:

                  mov A, RAM_TEMP_A
                  mov B, RAM_TEMP_B
                  add A, B ; add these 2 length (number and message)

                  pop B

                  add A, B ; add +8

                  ; the length has summed up. Then we need convert it to decimal

                  call convert_hex_to_int
                  push Acc ; save output temporary to STACK

                  ; send through serial UART (HIGH-order byte) by converting it first
to ascii
                  anl A, #0F0h
                  RR A
                  RR A
                  RR A
                  RR A
                  call convert_to_ascii
                  ; send
                  call send_serial

                  ; send through serial UART (LOW-order byte) by converting it first
to ascii
                  pop Acc ; get data from STACK
                  anl A, #0Fh
                  call convert_to_ascii
                  ; send
                  call send_serial

               mov A, #13 ; <CR>
               call send_serial

            ; > Now wait till we get '>' char

            clr EA

            wait_mp_send:
               clr RI
               jnb RI, $
               mov A, SBUF
            cjne A, #'>', wait_mp_send

            setb EA

            ; PDU Codes
```

```
            ; [Length of SMSC information. Here the length is 0, which means that
the SMSC stored in the phone should be used.]
               ; "00"
               mov A, #'0'
               call send_serial
               call send_serial

            ; [First octet of the SMS-SUBMIT message.] 1
               ; "11"
               mov A, #'1'
               call send_serial
               call send_serial

            ; [TP-Message-Reference. The "00" value here lets the phone set the
message reference number itself.] 2
               ; "00"
               mov A, #'0'
               call send_serial
               call send_serial

            ; [Recepient Number/Address-Length.] 3
               ; get NUMBER_LENGTH converted to ASCII

               mov TMP, NUMBER_LENGTH

               ; check for odd
               mov A, FLAG
               jnb Acc.5, no_odd_send
                  ; Odd number
                  dec TMP
               no_odd_send:

               ; get HIGH-order byte first
               mov A, TMP
               anl A, #0F0h ; filter HIGH-order only
               RR A ; convert HIGH to LOW
               RR A
               RR A
               RR A
               ; [convert to ASCII]
               call convert_to_ascii
               ; send to serial
               call send_serial

               ; get LOW-order byte first
               mov A, TMP
               anl A, #0Fh ; filter LOW-order only
               ; [convert to ASCII]
               call convert_to_ascii
               ; send to serial
               call send_serial

            ; [Type-of-Address. (91 indicates international format of the phone
number).] 4
               ; "91"
               mov A, #'9'
               call send_serial
               mov A, #'1'
               call send_serial

            ; [The phone number in semi octets.] +(NUMBER_LENGTH/2)
               ; send recepient number into serial port

               mov TMP, #0 ; var i
               Number_sent:

                  ; indirect-addressing
                  mov A, TMP
                  add A, #NUMBER_RAM_OFFSET
                  mov R1, A
```

```
      ; > get data from RAM
      mov A, @R1
      ; > send to UART
      call send_serial

      inc TMP
      mov A, TMP

   cjne A, NUMBER_LENGTH, Number_sent


; [TP-PID. Protocol identifier] 5
   ; "00"
   mov A, #'0'
   call send_serial
   call send_serial

; [TP-DCS Data coding scheme.] 6
   ; send '0' for HIGH-order byte
   ; the program doesn't care about the HIGH-order setting.
   mov A, #'0'
   call send_serial

   ; get coding scheme from FLAG register
   mov A, FLAG

   anl A, #00000011b ; filter data on SCHEME only

   ; decrement its value once.
   dec A

   ; "Destandarized" to follow PDU guideline
   RL A
   RL A

   ; [Convert to ASCII]
   call convert_to_ascii

   ; send to serial
   call send_serial

; [TP-Validity-Period.] 7
   ; "A9" SMS Validity sets to 3 days
   mov A, #'A'
   call send_serial
   mov A, #'9'
   call send_serial

; [TP-User-Data-Length. Length of message.] 8
   ; send TP_DATA_LENGTH register

   ; > before sending anything, check the mode first
   ;       if the mode is 0x03, which means 16-bit mode used,
   ;       then multiply length by 2

   mov A, FLAG
   anl A, #03h
   cjne A, #03h, dont_multiply_TP_LEN
      ; yes this is 16-bit mode used, so multiply it by 2
      mov A, TP_DATA_LENGTH
      mov B, #2
      mul AB
      mov TP_DATA_LENGTH, A
   dont_multiply_TP_LEN:

   ; get TP_DATA_LENGTH (HIGH-order byte)
   mov A, TP_DATA_LENGTH
   anl A, #0F0h ; filter HIGH-order only!
   RR A ; convert from HIGH to LOW
```

```
                RR A
                RR A
                RR A
                ; [convert to ASCII]
                call convert_to_ascii
                ; send to serial
                call send_serial

                ; get TP_DATA_LENGTH (LOW-order byte)
                mov A, TP_DATA_LENGTH
                anl A, #0Fh ; filter LOW-order only!
                ; [convert to ASCII]
                call convert_to_ascii
                ; send to serial
                call send_serial

             ; [Real message / TP-User-Data.] +(MESSAGE_LENGTH/2)

                ; send PDU Message into serial port

                ; > get mode that encoder used
                mov A, FLAG
                anl A, #00000011b

                cjne A, #03h, sending_type_a
                   jmp sending_type_b

                ; [MESSAGE SENDING TYPE A -- send without extension] -- 7-bit(0x01)
and 8-bit(0x02)

                sending_type_a:

                mov A, #0 ; var i
                mov B, #1 ; 7-bit (8x)-counter
                Message_sentA:

                   push Acc ; save A into STACK
                   push B ; save B into STACK

                      ; indirect-addressing
                      add A, #MESSAGE_RAM_OFFSET
                      mov R1, A

                      ; > get data
                      mov A, @R1
                      ; backup actual data to RAM_TEMP_B
                      mov RAM_TEMP_B, A

                      ; > if data == 0x00, then check for DISCARD?
                      cjne A, #00h, no_discard ; <- this check must be placed here,
becase this process handle 7-bit and 8-bit data.
                         ; (only 7-bit data goes through here, because 8-bit data
never NULL/00)
                         mov A, B
                         cjne A, #8, no_discard ; if we on the 8th iteraction, it's
data must be DISCARDED
                            pop B
                            mov B, #0 ; set B data residue on RAM to 0
                            push B
                            jmp discard_data ; discard data
                      no_discard:

                      ; > [process RAM A 1st / HIGH ORDER BYTE]
                         mov A, RAM_TEMP_B

                         anl A, #0F0h ; clear the LOW side
                         RR A ; shift right 4x
                         RR A
                         RR A
                         RR A
```

```
                        ; [convert from actual hex data to ASCII format]
                        call convert_to_ASCII
                        ; > send HIGH-ORDER byte
                        call send_serial

                     ; > [process RAM B / LOW ORDER BYTE]

                        mov A, RAM_TEMP_B
                        anl A, #0Fh

                        ; [convert from actual hex data to ASCII format]
                        call convert_to_ASCII
                        ; > send LOW-ORDER byte
                        call send_serial

                     discard_data:

                  pop B ; retrieve B from STACK
                  inc B

                  pop Acc ; retrieve A from STACK
                  inc A

               cjne A, MESSAGE_LENGTH, Message_sentA

               jmp skip_instruction_6

               ; [MESSAGE SENDING TYPE B -- send with '0''0' extension] -- 16-bit
only (0x03)
               sending_type_b:

               mov A, #0 ; var i
               Message_sentB:

                  push Acc ; save A into STACK

                     ; > set out frame/extension "00" before any data
                     mov A, #'0'
                     call send_serial
                     call send_serial

                     ; indirect-addressing
                     pop Acc
                     push Acc
                     add A, #MESSAGE_RAM_OFFSET
                     mov R1, A

                     ; > get data
                     mov A, @R1
                     ; backup actual data to RAM_TEMP_B
                     mov RAM_TEMP_B, A

                     ; > [process RAM A 1st / HIGH ORDER BYTE]
                        mov A, RAM_TEMP_B

                        anl A, #0F0h ; clear the LOW side
                        RR A ; shift right 4x
                        RR A
                        RR A
                        RR A

                        ; [convert from actual hex data to ASCII format]
                        call convert_to_ASCII
                        ; > send HIGH-ORDER byte
                        call send_serial

                     ; > [process RAM B / LOW ORDER BYTE]

                        mov A, RAM_TEMP_B
```

```
                        anl A, #0Fh

                        ; [convert from actual hex data to ASCII format]
                        call convert_to_ASCII
                        ; > send LOW-ORDER byte
                        call send_serial

                     discard_data2:

                  pop Acc ; retrieve A from STACK
                  inc A

               cjne A, MESSAGE_LENGTH, Message_sentB

            skip_instruction_6:

            ; [AT+CMGS Confirmation]
               ; send <CTRL-Z>

               mov A, #26
               call send_serial

            ; [WAIT UNTIL COMMAND COMPLETE]
               ;check the serial RX until we got "OK" response

               clr EA

               wait_send_resp:
                  clr RI
                  jnb RI, $
                  mov A, SBUF
               cjne A, #'O', wait_send_resp
                  clr RI
                  jnb RI, $
                  mov A, SBUF
                  cjne A, #'K', error_send_resp
                     jmp send_resp_ok
                  error_send_resp:
                     jmp $ ;hang
               send_resp_ok:

               clr RI
               jnb RI, $ ; <CR>
               clr RI
               jnb RI, $ ; <LF>

               setb EA

            call delay_1s ; let DCE/Mobile phone clear it's TX buffer

      ret

      instruction_7: ; Read Message Instruction

         ; Check for decode flag first, before read any data in RAM.
         mov A, FLAG
         jnb Acc.6, skip_instruction_7

            ; Read data on RAM.
            ; Sender Number first.
            ; Then message.

            ; Inside instruction function no 7 is to get all processed data from
   'decode' process

            ; > Set DR high
               setb DR

            ; > Send Number Length (actual value)
```

```
            ; Set COMM Port, number length
            mov COMM_Lines, NUMBER_LENGTH

            ; Wait for clock, (acknowledge)
            call Rise_nEN

         ; > Send Sender Numbers, digit by digit (by-product is auto ASCII).
            mov A, #0
            send_sn:
               push Acc

               ; indirect addressing
               add A, #NUMBER_RAM_OFFSET
               mov R1, A

               ; > get data from RAM
               mov A, @R1
               ; > set data to COMM. Port
               mov COMM_Lines, A

               ; Wait for clock, (acknowledge)
               call Rise_nEN

               pop Acc
               inc A
            cjne A, NUMBER_LENGTH, send_sn

         ; > Send Message Length
            ; Set COMM Port, number length
            mov COMM_Lines, MESSAGE_LENGTH

            ; Wait for clock, (acknowledge)
            call Rise_nEN

         ; > Send Message Data
            mov A, #0
            send_msg:
               push Acc

               ; indirect addressing
               add A, #MESSAGE_RAM_OFFSET
               mov R1, A

               ; > get data from RAM (after decode process, the data holds on RAM
actually ASCIIs)
               mov A, @R1

               ; > set data to COMM. Port
               mov COMM_Lines, A

               ; Wait for clock, (acknowledge)
               call Rise_nEN

               pop Acc
               inc A
            cjne A, MESSAGE_LENGTH, send_msg

         ; > Clear DR low and set COMM Port to (INPUT_MODE)
            clr DR
            mov COMM_Lines, #0FFh

      skip_instruction_7:
   ret

   instruction_8: ; DELETE_LAST_MESSAGE

      ; AT-COMMAND to delete message on mobile phone.
      ; the index number is followed by the last message index (stored in RAM)
      ; AT+CMGD=<INDEX>
```

```
      ; check index value first before deleting message!
               mov A, LO_SMS_INDEX  ; if LO_SMS_INDEX value still ' ', it means,
that no new message are received!
               cjne A, #' ', proceed_instruction_8
                      jmp cancel_inst_8

               proceed_instruction_8:

         clr TI
         mov A, #'A'
         call send_serial
         mov A, #'T'
         call send_serial
         mov A, #'+'
         call send_serial
         mov A, #'C'
         call send_serial
         mov A, #'M'
         call send_serial
         mov A, #'G'
         call send_serial
         mov A, #'D'
         call send_serial
         mov A, #'='
         call send_serial

         mov A, HI_SMS_INDEX
         cjne A, #' ', indx2_digit_delete ; check for 2nd digit value, if not ' '
then use 2 digit
            ; 1-digit delete operation
            mov A, LO_SMS_INDEX
            call send_serial
                              jmp indx_sel_ok

         indx2_digit_delete:

            mov A, HI_SMS_INDEX
            call send_serial
                              mov A, LO_SMS_INDEX
                              call send_serial

         indx_sel_ok:

         jb CRC, cancel_inst_8 ; when a new message arrived, when we still
executing this process, just end! don't confirm the delete process!! -- because
this process gonna delete the new one!
                              ; if no new message arrived, confirm this process.
                                     mov A, #13 ; <CR>
            call send_serial

            ; [WAIT UNTIL COMMAND COMPLETE]
               ;check the serial RX until we got "OK" response

               clr EA

               wait_del_resp:
                  clr RI
                  jnb RI, $
                  mov A, SBUF
               cjne A, #'O', wait_del_resp
                  clr RI
                  jnb RI, $
                  mov A, SBUF
                  cjne A, #'K', error_del_resp
                     jmp del_resp_ok
                  error_del_resp:
                     jmp $ ;hang
               del_resp_ok:

               clr RI
```

```
               jnb RI, $ ; <CR>
               clr RI
               jnb RI, $ ; <LF>

               setb EA

               call delay_1s ; let DCE/Mobile phone clear it's TX buffer

            ; clear the index number, just deleted!
            mov HI_SMS_INDEX, #' '
            mov LO_SMS_INDEX, #' '

         cancel_inst_8:

   ret

======================== ENCODER AND DECODER PROGRAM ==========================
; ******* Encode recepient number in RAM *******
   encode_numbers:

      ; check number length. Odd number / not?
         ; Number_Length % 2. If the result is 1, then it's an Odd number, else ,
it's not.

      mov A, FLAG
      jnb Acc.5, no_trail

         ; Odd number, add a trailling 'F'
         mov A, NUMBER_INDEX
         add A, #NUMBER_RAM_OFFSET

         mov R1, A ; indirect-addressing
         mov @R1, #'F' ; save data 'F'

         inc NUMBER_INDEX
         inc NUMBER_LENGTH
      no_trail:

      ; switch RAM addresses data.
         ; Save B to B'
         ; A -> B
         ; B' -> A

      mov A, #0

      next_digit_encode:
         push Acc ; save to STACK

            ; > Get data first.

            ; (ADDRESS: A)
               add A, #NUMBER_RAM_OFFSET
               mov R1, A ; indirect-addressing
               mov RAM_TEMP_A, @R1 ; get data, save into RAM

            pop Acc ; get RAM address from STACK
            push Acc ; --
            add A, #NUMBER_RAM_OFFSET ; --
            inc A ; next RAM address

            ; (ADDRESS: B)

               mov R1, A ; indirect-addressing
               mov RAM_TEMP_B, @R1 ; get data, save into RAM

            ; > Replace process

            mov TMP, RAM_TEMP_B ; RAM_TEMP_B -> TMP (Register) -- backup
            mov RAM_TEMP_B, RAM_TEMP_A ; RAM_TEMP_A -> RAM_TEMP_B
```

```
            mov RAM_TEMP_A, TMP; backed up(RAM_TEMP_B) / TMP (Register) ->
RAM_TEMP_A

            ; < Replace process end

               mov @R1, RAM_TEMP_B ; update B

            pop Acc ; get RAM address from STACK
            push Acc ; --
            add A, #NUMBER_RAM_OFFSET ; --

            ; (ADDRESS: A)

               mov R1, A ; indirect-addressing
               mov @R1, RAM_TEMP_A ; update A

         pop Acc ; get from STACK
         inc A ; next address, B
         inc A ; next address, Next Block, start again from A

      cjne A, NUMBER_LENGTH, next_digit_encode ; if processed data >=
NUMBER_LENGTH, then stop. else, loop back

   ret

; ******* Encode ASCII data into PDU code in RAM *******
   PDU_encode_7bit:

      mov TMP, Message_Length
      ;(EXAMPLE) mov R7, #11 ; R7 misalkan digunakan untuk data length

      ; SETELAH DIISI TERUS DIPROSES...

      mov R0, #1 ; ini merupakan counter untuk proses encoding
      ; Range yang dimiliki counter ini adalah 1 ~ 7
      ; Range: 0~6 Max: GESER 6x untuk R1
      ; Range: 1~7 Max: AMBIL/COPY 7x untuk R1+1 ke R1
      mov R1, #MESSAGE_RAM_OFFSET ; RAM Index

      ;--- FOR BEGIN --
      encode7_for:

      mov A, @R1 ; dapatkan data dari R1 ke Acc.

      ; 1. geser ke kanan sesuai dengan counternya... jika jumlah data bit-nya
sudah 7, tidak usah digeser!
      mov B, R0 ; melalui B soalnya ga bisa `mov R3, R0`. R3 mewakili nilai
counter
      mov R3, B
      ; Pergeseran akan hanya terjadi 6 kali, karena memiliki kemungkinan nilai R3
antara 1 ~ 7 dan dikurangi pertama kali oleh DJNZ, menghasilkan range 0 ~ 6
      pgeser_msb: djnz R3, geser_msb ; jika 0 maka jmpto nogeser_msb, jika bukan 0
maka geser_msb  <----. (berulang kali sesuai counter)
         jmp nogeser_msb ;
      geser_msb:      RR A ;
            jmp pgeser_msb  ;
      nogeser_msb:
      mov @R1, A ; simpan setelah digeser

      ; > buang data bagian MSB sampai dengan n sesuai counternya [1~7]
      mov A, R0 ; mengambil data filter, A = counter
      call filterMSB7bit ; mengambil data filter, save di 77h (Remove)
      mov A, @R1 ; ambil data aslinya. (R1)
      anl A, 77h ; buang sesuai data filter negatif di 77h!
      mov @R1, A ; simpan setelah dibuang datanya

      ; jika (R1) merupakan data terakhir, maka skip langkah No. 2 dibawah ini...
      cjne R7, #1, next_step_2
         jmp last_data
      next_step_2:
```

```
      ; 2. abis (R1) digeser dan dibersihkan, (R1+1) diekstrak,dicopy/OR ke prev
data(R1), terus (R1+1) difilter yang mo dibuang sesuai counter.
      inc R1 ; ganti dengan data selanjutnya, (R1+1)

      ; > ekstrak terlebih dahulu sebelum dibuang datanya
      mov A, R0 ; mengambil data filter, A = counter
      call filterLSB7bit ; mengambil data filter, save di 76h (Extract)
      mov A, @R1 ; ambil data aslinya (R1+1)
      anl A, 76h ; extract yang penting saja, sesuai dengan nilai filter positif
di 76h!
      ; tidak usah disimpan, karena jika disimpan maka data aslinya akan hilang!

      ; > berhubung posisi bit setelah diekstrak masih di LSB, maka kita geser
sehingga ada di MSB. Geser ke kanan sesuai counter
      mov B, R0 ; melalui B soalnya ga bisa `mov R3, R0`. R3 mewakili nilai
counter
      mov R3, B
      geser_lsb_ke_msb:       RR A ;       <----. (berulang kali sesuai counter)
      djnz R3, geser_lsb_ke_msb  ;            --'
      mov B, A ; untuk sementara kita simpan di B (hasil ekstrak pergeseran LSB ke
MSB)

      ; > OR kan dengan data sebelumnya (R1)
      dec R1 ; mundur ke data awal (R1)
      mov A, @R1
      orl A, B ; OR
      mov @R1, A ; simpan datanya ke (R1)

      inc R1 ; maju ke (R1+1)

      ; > buang datanya(LSB) sesuai counter
      mov A, R0 ; masukan ke fungsi Acc = filterLSB7bit(Acc)
      call filterLSB7bit ; gunakan 77h karena tujuannya adalah untuk membuang
data!
      mov A, @R1 ; ambil data aslinya (R1 + 1)
      anl A, 77h ; menghasilkan data yang terbuang!
      mov @R1, A ; simpan hasil yang terbuang.

      ; Tadinya akan ada proses ini, akan tetapi menjadi masalah karena akan
tergeser pula pada iteraksi selanjutnya... (pada tahap no. 1)
      ; > karena setelah dibuang posisi data bitnya belum benar, maka harus
digeser kembali... (ini juga sesuai dengan nilai counter)
      ;mov B, R0 ; melalui B soalnya ga bisa `mov R3, R0`. R3 mewakili nilai
counter
      ;mov R3, B
      ;geser_R1p1:  RR A ;       <----. (berulang kali sesuai counter)
      ;djnz R3, geser_R1p1 ;        --'
      ;mov @R1, A ; simpan hasil yang posisinya sudah benar.

      dec R1 ; kembalikan ke posisi awal

      ; 3 Proses filter,copy,remove selesai, lantas sekarang lanjutkan address
selanjutnya...

      inc R0 ; Tambahkan jumlah bit yang akan digeser dan dicopy...
      cjne R0, #8, no_reset_geser_counter
         ; Jika counter sudah mencapai 8, maka reset ke 1 lagi.
         mov R0, #1
         ; Pada tahap ini data selanjutnya adalah 0x00, maka dari itu, kita skip
data tsb, ke data selanjutnya
         inc R1 ; Next address RAM
         ; Akibat data yang ter-skip tadi, maka panjang dari data yang ada (asli)
dikurangi pula
         dec R7 ; Length - 1
      no_reset_geser_counter:

      inc R1 ; Next address RAM

      last_data:
```

```
      ;--- FOR END ---
      djnz R7, encode7_for

   ret

   PDU_encode_8bit:
      ; Nothing to encode, skip this.
   ret

   PDU_encode_16bit:
      ; Nothing to encode, skip this.
      ; Just add "00" to every ASCII Codes.
   ret

; ******* Decode sender number *******
   Decode_number:
      ; Swap sender number
      ; RAM_BLOCK ['A']['B']
      ;
      ; result: ['B']['A']

      ; TEMP = [1]
      ; [1] = [2]
      ; [2] = TEMP

      ; > swap data
      mov A, #0

      next_dn:
         push Acc ; save counter on STACK

            ; indirect-addressing
            add A, #NUMBER_RAM_OFFSET
            mov R1, A

            ; [BLOCK-1]
            ; > get data [1]
            mov A, @R1
            ; > save temporary on RAM, TEMP
            mov RAM_TEMP_A , A

            ; > next number RAM address
            inc R1 ; WARNING!: Next number RAM address
            ; > get data [2], save to B
            mov B, @R1
            ; > set data [1] = data [2]
            dec R1 ; WARNING!: Prev number address
            mov @R1, B

            ; [BLOCK-2]
            ; > next number RAM address
            inc R1
            ; > set data [2] = TEMP
            mov @R1, RAM_TEMP_A

         pop Acc ; get counter value back
         inc A ; goto [BLOCK-3], this can be achieved by incrementing the counter
twice
         inc A
      cjne A, NUMBER_LENGTH, next_dn

      ; > check for last data, is it 'F' ?. If yes then decrement sender length
         ; indirect-addressing
         mov A, NUMBER_INDEX
         dec A ; last data index
         add A, #NUMBER_RAM_OFFSET
         mov R1, A

         ; > get data
         mov A, @R1
```

```
         ; > compare
         cjne A, #'F', no_trail_F
            ; there is a trailling 'F'
            ; decrement data
            dec NUMBER_INDEX
            dec NUMBER_LENGTH
         no_trail_F:
   ret

; ******* Decode PDU Code in RAM back into ASCII Data *******
   PDU_decode_7bit:

         ; set panjang PDU / PDU data length
         mov TMP, MESSAGE_LENGTH
         ;(EXAMPLE) mov TMP, #9 ; total 9 Data PDU-Codes

         ; set alamat mulainya
         MESSAGE_DECODE_START_ADDR equ MESSAGE_RAM_OFFSET
         mov R1, #MESSAGE_DECODE_START_ADDR

         ; set counternya
         mov R0, #1 ; Counter ini memiliki range dari 1 hingga 8

         ; reset variables
         mov R2, #0
         mov R3, #0
         mov R4, #0

   decode_7bit:
      ; Perintah yang diberi tanda * berarti opsional

         ; 1. panggil data di alamat R1

            mov B, @R1 ; untuk sementara disimpan di B, karena A akan dipakai
untuk filter

         ; 2. Extract MSB (R1) sesuai jumlah counter (simpan di R2), lalu buang
data (R1) setelah diekstrak

            ; > Dapatkan nilai filter MSB dengan input nilai counter
            mov A, R0
            call filterMSB

            ; > Ekstrak nilai MSB
            mov A, B ; dapatkan kembali data asli ke A
            anl A, 76h ; extract nilai MSB
            mov R2, A ; simpan ke temporary storage1

            ; > Buang nilai MSB data asli
            mov A, B ; dapatkan kembali data asli ke A
            anl A, 77h ; remove nilai MSB
            mov @R1, A ; simpan ke lokasi aslinya

         ; 3. Sesuaikan data yang telah diextract (R2) dari posisi MSB ke LSB
dengan cara menggesernya ke kiri sesuai jumlah counter.

            ; > Load counter
            ;mov R4, R0
            mov B, R0 ; melalui B karena `mov R4, R0` itu tidak boleh
            mov R4, B

            ; > Load data ektrak-an tadi
            mov A, R2

            ; > Geser ke kiri sesuai jumlah counter
            geser_temp1:
               RL A
            djnz R4, geser_temp1
```

```
            ; > Simpan lagi ke tempatnya
            mov R2, A

         ; 4. Shift ke kiri data (R1) sesuai jumlah (counter-1)

            ; > Load counter
            ;mov R4, R0
            mov B, R0 ; melalui B karena `mov R4, R0` itu tidak boleh
            mov R4, B

            ; > Load datanya
            mov A, @R1

            ; > Pengurangan dimuka oleh perintah `djnz`, sehingga pergeserah
sesuai (counter - 1)
            step4d3:
            djnz R4, lanjut_shift_step4
               jmp end_step4 ; tidak ada penggeseran ke kiri, gara-gara sudah nol
            lanjut_shift_step4:
               RL A
               jmp step4d3

            end_step4:

            ; > Simpan kembali datanya
            mov @R1, A

         ; 5. OR kan dengan data dari iteraksi sebelumnya (R3)

            ; > Dapatkan data asli (R1)
            mov A, @R1

            ; > Dapatkan nilai temp2 (R3)
            mov B, R3

            ; > OR kan dengan data temp2 (R3)
            orl A, B

            ; > Simpan kembali datanya (R1)
            mov @R1, A

         ; 6. Swap posisi dari R2 ke R3. (Kemudian hapus nilai di R2 karena tidak
signifikan)*

            ; > Simpan nilai R2 ke R3
            mov A, R2
            mov R3, A

            ; > Hapus nilai di R2
            mov R2, #0

         ; 7. Last step on the loop

            ; > Next address
            inc R1 ; alamat selanjutnya di RAM

            ; > Increment counter
            inc R0 ; tambahkan nilai counter

            ; > check for counter overflow, jika melebihi 8 maka terdapat data
baru!
            cjne R0, #9, no_reset_counter

               ; simpan sementara Alamat / Current Address di STACK, melalui B
               mov B, R1
               push B

               ; # Terdapat data baru! geser data yang sebelumnya menempati posisi
ini ([R1+i]+1) ke posisi ([R1+i]+2) sampai seluruh akhir data.
```

```
                ; Proses ini dilakukan secara mundur (dari paling belakang lokasi
RAM)
                mov A, R7 ; dapatkan sisa length.
                add A, B ; tambahkan lokasi sekarang dengan sisa length
                inc A ; tambahkan sekali, karena adanya data baru ini
                mov R1, A ; simpan ke R1 sementara, hanya section ini saja.

                mov A, R7 ; dapatkan sisa length, simpan ke R4
                mov R4, A
                inc R4 ; karena terdapat data baru ini, maka length harus
ditambahkan terlebih dahulu

                move_RAM:
                   dec R1 ; mundur sekali untuk mendapatkan data
                   mov A, @R1 ; simpan ke A
                   inc R1 ; maju kembali sekali, untuk menyimpan data
                   mov @R1, A ; simpan ke lokasi dari A

                   dec R1 ; Address bergerak mundur
                djnz R4, move_RAM


                pop B
                mov R1, B

                inc R7 ; tambahkan length untuk iteraksi, karena terdapat data
baru!
                inc MESSAGE_INDEX
                inc MESSAGE_LENGTH

                ; # Simpan data baru tersebut (berada di R3) ke lokasi (R1+1)
                mov A, R3
                mov @R1, A
                mov R3, #0 ; hapus data baru yang tersimpan di R3!

                ; # reset counter ke 1
                mov R0, #1

            no_reset_counter:

         djnz R7, decode_7bit

   ret

   PDU_decode_8bit:
      ; Nothing to decode, the data itself is actually ASCII Codes.
   ret

   PDU_decode_16bit:
      ; Nothing to decode, the data itself is actually ASCII Codes.
      ; Just to remove frame "00" on every data.
   ret

====================== PROGRAM FUNCTIONS ====================================
; ******* LCD TEXTS DISPLAY FUNCTION *******

   display_header:

      ; "M.P.D Translator"

      call LCD_Clear

      mov A, #'M'
      call LCD_Write_Data
      mov A, #'.'
      call LCD_Write_Data
      mov A, #'P'
      call LCD_Write_Data
      mov A, #'.'
      call LCD_Write_Data
```

```
   mov A, #'D'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'T'
   call LCD_Write_Data
   mov A, #'r'
   call LCD_Write_Data
   mov A, #'a'
   call LCD_Write_Data
   mov A, #'n'
   call LCD_Write_Data
   mov A, #'s'
   call LCD_Write_Data
   mov A, #'l'
   call LCD_Write_Data
   mov A, #'a'
   call LCD_Write_Data
   mov A, #'t'
   call LCD_Write_Data
   mov A, #'o'
   call LCD_Write_Data
   mov A, #'r'
   call LCD_Write_Data

ret

;---------------------- these functions only writes to 2nd row ----------------

display_welcome:

   ; ": Welcome"

   ;call LCD_Clear_2nd

   mov A, #':'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'W'
   call LCD_Write_Data
   mov A, #'e'
   call LCD_Write_Data
   mov A, #'l'
   call LCD_Write_Data
   mov A, #'c'
   call LCD_Write_Data
   mov A, #'o'
   call LCD_Write_Data
   mov A, #'m'
   call LCD_Write_Data
   mov A, #'e'
   call LCD_Write_Data

ret

display_design_by_joc:

   ; ": Design by JoC"

   call LCD_Clear_2nd

   mov A, #':'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'D'
   call LCD_Write_Data
   mov A, #'e'
   call LCD_Write_Data
```

```
   mov A, #'s'
   call LCD_Write_Data
   mov A, #'i'
   call LCD_Write_Data
   mov A, #'g'
   call LCD_Write_Data
   mov A, #'n'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'b'
   call LCD_Write_Data
   mov A, #'y'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'J'
   call LCD_Write_Data
   mov A, #'o'
   call LCD_Write_Data
   mov A, #'C'
   call LCD_Write_Data

ret

display_done:

   ; "> Done."

   call LCD_Clear_2nd

   mov A, #'>'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'D'
   call LCD_Write_Data
   mov A, #'o'
   call LCD_Write_Data
   mov A, #'n'
   call LCD_Write_Data
   mov A, #'e'
   call LCD_Write_Data
   mov A, #'.'
   call LCD_Write_Data

ret

display_checking_mp:

   ; "> Checking M.P"

   call LCD_Clear_2nd

   mov A, #'>'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'C'
   call LCD_Write_Data
   mov A, #'h'
   call LCD_Write_Data
   mov A, #'e'
   call LCD_Write_Data
   mov A, #'c'
   call LCD_Write_Data
   mov A, #'k'
   call LCD_Write_Data
   mov A, #'i'
   call LCD_Write_Data
```

```
   mov A, #'n'
   call LCD_Write_Data
   mov A, #'g'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'M'
   call LCD_Write_Data
   mov A, #'.'
   call LCD_Write_Data
   mov A, #'P'
   call LCD_Write_Data

ret

display_conf_mp:

   ; "> Conf. M.P"

   call LCD_Clear_2nd

   mov A, #'>'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'C'
   call LCD_Write_Data
   mov A, #'o'
   call LCD_Write_Data
   mov A, #'n'
   call LCD_Write_Data
   mov A, #'f'
   call LCD_Write_Data
   mov A, #'.'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'M'
   call LCD_Write_Data
   mov A, #'.'
   call LCD_Write_Data
   mov A, #'P'
   call LCD_Write_Data

ret

display_slash:

   ; "/[CURSOR]

   call LCD_Clear_2nd

   mov A, #'/'
   call LCD_Write_Data

ret

display_busy:

   ; "> Busy!"

   call LCD_Clear_2nd

   mov A, #'>'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'B'
   call LCD_Write_Data
   mov A, #'u'
```

```
   call LCD_Write_Data
   mov A, #'s'
   call LCD_Write_Data
   mov A, #'y'
   call LCD_Write_Data
   mov A, #'!'
   call LCD_Write_Data

ret

display_new_sms:

   ; "> New SMS! [XX]"

   call LCD_Clear_2nd

   mov A, #'>'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'N'
   call LCD_Write_Data
   mov A, #'e'
   call LCD_Write_Data
   mov A, #'w'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'S'
   call LCD_Write_Data
   mov A, #'M'
   call LCD_Write_Data
   mov A, #'S'
   call LCD_Write_Data
   mov A, #'!'
   call LCD_Write_Data

   mov A, #' '
   call LCD_Write_Data
   mov A, #'['
   call LCD_Write_Data

   mov A, HI_SMS_INDEX
   call LCD_Write_Data
   mov A, LO_SMS_INDEX
   call LCD_Write_Data

   mov A, #']'
   call LCD_Write_Data

ret

display_error:

   ; "> Error!!"

   call LCD_Clear_2nd

   mov A, #'>'
   call LCD_Write_Data
   mov A, #' '
   call LCD_Write_Data
   mov A, #'E'
   call LCD_Write_Data
   mov A, #'r'
   call LCD_Write_Data
   mov A, #'r'
   call LCD_Write_Data
   mov A, #'o'
   call LCD_Write_Data
```

```
      mov A, #'r'
      call LCD_Write_Data
      mov A, #'!'
      call LCD_Write_Data
      mov A, #'!'
      call LCD_Write_Data

   ret

; ******* SERIAL UART FUNCTIONS *******
   send_serial:
      ; Send data from Acc to SBUF
      clr TI ; clear transmitted flag
      mov SBUF, A
      jnb TI, $ ; wait until all bits has been transferred.
      clr TI
   ret


; ******* MISC FUNCTIONS *******
   save_sms_index:
      jnb RI, $ ; wait for next data, A
      clr RI
      mov LO_SMS_INDEX, SBUF ; save at RAM (this is for 1-digit SMS INDEX)
      mov HI_SMS_INDEX, SBUF ; save at RAM (this is for 2-digit SMS INDEX)

      jnb RI, $ ; wait for next data, B
      clr RI
      ;mov LO_SMS_INDEX, SBUF ; save at RAM
      mov A, SBUF
      cjne A, #13, sms_2_digit_index ; check it is <CR> ?, if yes then
         ; Just 1-digit, so clear the HI_SMS_INDEX
         mov HI_SMS_INDEX, #' ' ; set the RAM content to ASCII ' '
         jmp sms_index_saved

      sms_2_digit_index:

         mov LO_SMS_INDEX, SBUF ; save at RAM

      sms_index_saved:
   ret

; ******* PROGRAM CONVERSIONS *******

   convert_hex_to_int:
      ; [Conversion from HexToInt]
      ; Input : Acc
      ; Output : Acc

      ; [WARNING!!] :
      ; This conversion result can up to 99 decimal points. The input maximum is
0x63.
      ; Any overflow input occured, that may set output unexpectedly!
      ; So precaution must be included before using this function!

      ; Source/Input : A

      mov B, #10
      div AB

      ; A = HIGH-ORDER decimal / puluhan [ division result ]
      ; B = LOW-ORDER decimal / satuan [ modulus result ]

      RL A ; convert to HIGH-order byte
      RL A
      RL A
      RL A

      orl A, B
```

```
      ; Destination : A
   ret


   convert_to_ASCII:
      ; INPUTS: Acc
      ; OUTPUT: Acc

      push Acc ; backup original data

      clr CY ; clear carry flag
      add A, #246

      jnb CY, numeric_type
         ; non-numeric data
         pop Acc ; get data
         add A, #55 ; 65 - 0x0A
         jmp ctA_finish
      numeric_type:
         ; numeric data
         pop Acc ; get data
         add A, #48
      ctA_finish:

   ret

   convert_to_Hex:
      ; INPUT: Acc
      ; OUTPUT: Acc

      push Acc
      clr C
      subb A, #65

      jnc hex_char ; compare

      hex_num:
         ; This section for numbers (0 ~ 9)
         pop Acc
         clr C
         subb A, #48

      jmp cth_done
      hex_char:
         ; This section for chars (A ~ F)
         pop Acc
         clr C
         subb A, #55

      cth_done:
   ret

; ******* SGINAL TRANSITION FUNCTIONS *******
   Fall_nEN:
      ; Wait for a falling signal transition occur on EN pin
      Fall_nEN_a:
         jnb nEL, Fall_nEN_finish ; Interrupt pin / Cancel Operation
      jnb nEN , Fall_nEN_a ; wait until pin is HIGH
      Fall_nEN_b:
         jnb nEL, Fall_nEN_finish ; Interrupt pin / Cancel Operation
      jb nEN , Fall_nEN_B ; wait until pin is LOW
      Fall_nEN_finish:
   ret

   Rise_nEN:
      ; Wait for a rising signal transition occur on EN pin
      Rise_nEN_a:
         jnb nEL, Rise_nEN_finish ; Interrupt pin / Cancel Operation
      jb nEN, Rise_nEN_a ; wait until pin is LOW
      Rise_nEN_b:
```

```
         jnb nEL, Rise_nEN_finish ; Interrupt pin / Cancel Operation
      jnb nEN , Rise_nEN_b ; wait until pin is HIGH
      Rise_nEN_finish:
   ret

   wait_soft_reset:
      ; Wait for a rising signal transition occur on EL pin
      ; skip if its level already HIGH (wait for input) / inactive function
      jb nEL, skip_Rise_nEL

         ; level at LOW (currently active)

         ; clear 2nd row of LCD
         call LCD_clear_2nd

         ; reset Communication Port
         mov COMM_Lines, #0FFh

         clr CRC ; clear new message indication

         ; reset values
         mov NUMBER_INDEX, #0
         mov NUMBER_LENGTH, #0
         mov MESSAGE_INDEX, #0
         mov MESSAGE_LENGTH, #0

         ; reset flag
         mov FLAG, #00h

         jnb nEL , $ ; wait nEL for going HIGH / inactive

      skip_Rise_nEL:



   ret

; ******* FILTER FUNCTIONS *******
   filterMSB:
      ; Ini adalah fungsi penghasil nilai filter.
      ; Nilai keluaran dari filter ini didapatkan dari perbandingan input dengan
programnya.

      ;
      ;  USAGE : filterMSB(n_size)
      ;
      ;  INPUT : Accumulator (from 7 to (n-7)) --
      ; OUTPUT : Accumulator with Filter value output

      cjne A, #0, no_fMSB8
         mov A, #00000000b
         jmp end_filterMSB
      no_fMSB8:
      cjne A, #1, no_fMSB7
         mov A, #10000000b
         jmp end_filterMSB
      no_fMSB7:
      cjne A, #2, no_fMSB6
         mov A, #11000000b
         jmp end_filterMSB
      no_fMSB6:
      cjne A, #3, no_fMSB5
         mov A, #11100000b
         jmp end_filterMSB
      no_fMSB5:
      cjne A, #4, no_fMSB4
         mov A, #11110000b
         jmp end_filterMSB
      no_fMSB4:
      cjne A, #5, no_fMSB3
```

```
      mov A, #11111000b
      jmp end_filterMSB
   no_fMSB3:
   cjne A, #6, no_fMSB2
      mov A, #11111100b
      jmp end_filterMSB
   no_fMSB2:
   cjne A, #7, no_fMSB1
      mov A, #11111110b
      jmp end_filterMSB
   no_fMSB1:
   cjne A, #8, no_fMSB0
      mov A, #11111111b
      jmp end_filterMSB
   no_fMSB0:

   end_filterMSB:

   ;save filter data on 2 location on RAM. One positive and One negative
   mov 76h, A ; Filter [EXTRACT MSB]={AND F, then OR with OtherVar}
   cpl A
   mov 77h, A ; NOT Filter [REMOVE MSB]={AND ~F}

ret

filterMSB7bit:
   ; Kegunaan: Fungsi ini digunakan HANYA pada data @R1
   ; Inputnya A.
   ; Outputnya A.
   ; Prosesnya: A = Process(A)
   ; Ini adalah filtering process... yang dimulai dari bit MSB

   cjne A, #1, no_MSBsetb7
      mov A, #10000000b
      jmp end_selectFMSB
   no_MSBsetb7:

   cjne A, #2, no_MSBsetb6
      mov A, #11000000b
      jmp end_selectFMSB
   no_MSBsetb6:

   cjne A, #3, no_MSBsetb5
      mov A, #11100000b
      jmp end_selectFMSB
   no_MSBsetb5:

   cjne A, #4, no_MSBsetb4
      mov A, #11110000b
      jmp end_selectFMSB
   no_MSBsetb4:

   cjne A, #5, no_MSBsetb3
      mov A, #11111000b
      jmp end_selectFMSB
   no_MSBsetb3:

   cjne A, #6, no_MSBsetb2
      mov A, #11111100b
      jmp end_selectFMSB
   no_MSBsetb2:

   cjne A, #7, no_MSBsetb1
      mov A, #11111110b
      jmp end_selectFMSB
   no_MSBsetb1:

   ; Secara kemungkinan sih, counter tidak sampai nilai 8
   ;cjne A, #7, no_MSBsetb0
      ;     mov A, #11111111b
```

```
      ;no_MSBsetb0:

      end_selectFMSB:

      mov 76h, A ; Filter [EXTRACT]={AND, then OR with OtherVar}
      cpl A
      mov 77h, A ; NOT Filter [REMOVE]={AND}
   ret

   filterLSB7bit:
      ; Kegunaan: Fungsi ini HANYA digunakan pada data @R1+1
      ; Inputnya A.
      ; Outputnya A.
      ; Prosesnya: A = Process(A)
      ; Ini adalah filtering process... yang dimulai dari bit LSB

      ; Kenapa berhenti di bit ke 6, bukan bit ke 7?
      ;Karena proses pembuangan itu dimulai dari bit 0, dan berakhir di bit ke 6
      ;dan itu juga dikarenakan jumlah datanya adalah 7-bit
      ;        (Untuk lebih jelasnya silahkan lihat tutorialnya)

      cjne A, #0, no_LSBsetb0
         mov A, #00000000b
         jmp end_selectFLSB ; ini agar tidak terjadi kesalahan, apabila nilai
telah diset, masuk kedalam if yang lain.
      no_LSBsetb0:

      cjne A, #1, no_LSBsetb1
         mov A, #00000001b
         jmp end_selectFLSB
      no_LSBsetb1:

      cjne A, #2, no_LSBsetb2
         mov A, #00000011b
         jmp end_selectFLSB
      no_LSBsetb2:

      cjne A, #3, no_LSBsetb3
         mov A, #00000111b
         jmp end_selectFLSB
      no_LSBsetb3:

      cjne A, #4, no_LSBsetb4
         mov A, #00001111b
         jmp end_selectFLSB
      no_LSBsetb4:

      cjne A, #5, no_LSBsetb5
         mov A, #00011111b
         jmp end_selectFLSB
      no_LSBsetb5:

      cjne A, #6, no_LSBsetb6
         mov A, #00111111b
         jmp end_selectFLSB
      no_LSBsetb6:

      cjne A, #7, no_LSBsetb7
         mov A, #01111111b
         jmp end_selectFLSB
      no_LSBsetb7:

      ; #1111.1111b tidak dimasukan karena kapasitas bitnya adalah 7, bukan 8!

      end_selectFLSB:

      mov 76h, A ; Filter [EXTRACT]={AND, then OR with OtherVar}
      cpl A
      mov 77h, A ; NOT Filter [REMOVE]={AND}
   ret
```

```
; ******* DELAY FUNCTIONS *******
   delay_50ms:
      ;This delay function use Timer0 for it's source.
      ;Setup Timer0 values
      ;Equotation :
      ; (1 tick equal to 1.085 us)
      ; 50 ms / 1.085 us = delay ticks
      ; delay ticks = 46080
      ;
      ; Timer0 overflow after 46080 ticks, so the initialization value is:
      ; init value = 65536 - 46080
      ; init value = 19456, in hex value is 0x4C00
      ;

      mov TL0, #00h
      mov TH0, #4Ch
      clr TF0 ; clear the overflow flag
      setb TR0 ; start Timer0
      jnb TF0 , $ ; wait and check until Timer0 overflow
      clr TR0 ; stop Timer0
      clr TF0 ; clear the overflow flag
   ret

   delay_1s:
      ; For a 1 seconds delay, we gonna use the 50ms delay.
      ; 1 seconds is equivalent to 20x 50ms delay.
      ; Loop 20 times for delay_50ms

      mov A, TMP ; save TMP on stack
      push Acc
      push PSW

      mov TMP, #20
      delay_1s_sub:
         call delay_50ms
      djnz TMP, delay_1s_sub

      pop PSW
      pop Acc
      mov TMP, A ; restore TMP
   ret


   delay_5s:
      ; For a 5 seconds delay, we gonna use the 50ms delay.
      ; 5 seconds is equivalent to 100x 50ms delay.
      ; Loop 100 times for delay_50ms
      mov TMP, #100
      delay_5s_sub:
         call delay_50ms
      djnz TMP, delay_5s_sub
   ret

   delay_forever:
      jmp $
   ret

; ******* LCD FUNCTIONS *******
; ### basic functions ###
   LCD_Write_Cmd:
      ; 1. EN default is LOW.
      ; 2. Clear RS to LOW (Instruction Mode.)
      ; 3. Clear RW to LOW (Mode Write.)
      ; 4. Set EN to HIGH (enable LCD. Comm.)
      ; 5. Put data into LCD data lines
      ; 6. Clear EN to LOW
      ; write mode finished.

      clr LCD_EN
```

```
   clr LCD_RS
   clr LCD_RW
   setb LCD_EN
   setb LCD_EN ; delay 1us
   mov LCD_Lines, A ; OUT
   clr LCD_EN

   call LCD_Wait

ret

LCD_Write_Data:
   ; 1. EN default is LOW.
   ; 2. Set RS to HIGH (Data Mode.)
   ; 3. Clear RW to LOW (Mode Write.)
   ; 4. Set EN to HIGH (enable LCD. Comm.)
   ; 5. Put data into LCD data lines
   ; 6. Clear EN to LOW
   ; write mode finished.

   clr LCD_EN
   setb LCD_RS
   clr LCD_RW
   setb LCD_EN
   setb LCD_EN ; delay 1us
   mov LCD_Lines, A ; OUT
   clr LCD_EN

   call LCD_Wait
ret

LCD_Read_Data:
   ; This function read current data on current RAM pointer on LCD.
   ; data stored at Acc.

   ; 1. EN default is LOW.
   ; 2. Set RS to HIGH (Data Mode.)
   ; 3. Set RW to HIGH (Mode Read.)
   ; 4. Set EN to HIGH (enable LCD. Comm.)
   ; 5. Put data into LCD data lines
   ; 6. Clear EN to LOW
   ; read mode finished.
   clr LCD_EN
   setb LCD_RS
   setb LCD_RW
   setb LCD_EN
   setb LCD_EN ; delay 1us
   mov A, LCD_Lines ; data direction IN
   clr LCD_EN
ret

LCD_Read_Cmd:
   ; data stored at Acc.

   ; 1. EN default is LOW.
   ; 2. Clear RS to LOW (Instruction Mode.)
   ; 3. Set RW to HIGH (Mode Read.)
   ; 4. Set EN to HIGH (enable LCD. Comm.)
   ; 5. Put data into LCD data lines
   ; 6. Clear EN to LOW
   ; read mode finished.
   clr LCD_EN
   clr LCD_RS
   setb LCD_RW
   setb LCD_EN
   setb LCD_EN ; delay 1us
   mov A, LCD_Lines ; data direction IN
   clr LCD_EN
ret
```

```
; ### end basic functions ###

; ### complex functions ###
   LCD_Init:

      ; 1. Function Set
      ; 2. Display ON/OFF (SWAPPED)
      ; 3. Entry Mode Set (SWAPPED)
      ; 4. Return Home.

      ; 1. Function Set
      ; configurations : Binary( 001_DL._N_F_XX )
      ; DL = 1, 8 bit interface.
      ; N = 1, 2 Line Display. We use both of them.
      ; F = 0, 5x8 Font. We use 2 line, so font must be small.

      mov A, #00111000b
      call LCD_Write_Cmd

      ; 2. Display ON/OFF
      ; configurations : Binary( 0000.1_D_C_B )
      ; D = 1, Display ON
      ; C = 0, Cursor OFF
      ; B = 0, Cursor Blinkin OFF

      mov A, #00001100b
      call LCD_Write_Cmd

      ; 3. Entry Mode Set, set cursor moving direction
      ; configurations : Binary( 0000.01_ID_S )
      ; ID = 0, display shift left
      ; S = 0, no screen shifting.

      mov A, #00000110b
      call LCD_Write_Cmd

      ; 4. Return Home, set cursor to top-left position.
      ; configurations : Binary( 0000.001X )

      mov A, #02h
      call LCD_Write_Cmd

   ret

   LCD_Clear:
      mov A, #01h
      call LCD_Write_Cmd
   ret

   LCD_Clear_1st:
      ; Clear the 1st line of LCD Display
      ; > cursor goto 1st row, 1st coloumn
      call LCD_goto_1st

      ; Write ' ' for each coloumn
      mov B, #16
      LCD_Clear_1st_a:
         mov A, #' '
         call LCD_Write_Data
      djnz B, LCD_Clear_1st_a
      ; > Back to 1st row 1st coloumn
      call LCD_goto_1st
   ret

   LCD_Clear_2nd:
      ; Clear the 2nd line of LCD Display
      ; > cursor goto 2nd row, 1st coloumn
      call LCD_goto_2nd
      ; Write ' ' for each coloumn
      mov B, #16
```

```
      LCD_Clear_2nd_a:
         mov A, #' '
         call LCD_Write_Data
      djnz B, LCD_Clear_2nd_a
      ; > Back to 2nd row 1st coloumn
      call LCD_goto_2nd
   ret

   LCD_goto_1st:
      mov A, #(80h + 00h) ; goto 1st row of the LCD Lines
      call LCD_Write_Cmd
   ret

   LCD_goto_2nd:
      mov A, #(80h + 40h) ; goto 2nd row of the LCD Lines
      call LCD_Write_Cmd
   ret

   LCD_Cursor_ON:
      ; Turn on Cursor, and activate cursor blink
      ; $Display_Switch function
      ; 0000.1DCB
      ; D stand for display ON/OFF
      ; C stand for cursor ON/OFF
      ; B stand for cursor blink mode ON/OFF

      mov A, #00001111b
      call LCD_Write_Cmd

   ret

   LCD_Cursor_OFF:
      ; Turn off Cursor, and deactivate cursor blink
      ; $Display_Switch function
      ; 0000.1DCB
      ; D stand for display ON/OFF
      ; C stand for cursor ON/OFF
      ; B stand for cursor blink mode ON/OFF

      mov A, #00001100b
      call LCD_Write_Cmd

   ret

   LCD_Wait:
      ; NOTE: WE CAN'T USE LCD_Read_Cmd, coz we need processing inside the
function.

      ; Get system status, busy or not?
      ; if busy, hold on here, if not, instruction goes on.
      ; Configurations : Binary( BF[Bit7]_AddressCounterContent[Bit6-0] )

      clr LCD_EN
      clr LCD_RS ; Instruction Mode.
      setb LCD_RW ; Read mode.

      mov LCD_Lines, #0FFh ; pull-up plz

      setb LCD_EN ; start E-Cycle.
      setb LCD_EN ; delay 1us

      jb LCD_LinesBit7, $ ; hold here, if BF still 1, which mean busy.
      ; LCD Ready.

      clr LCD_EN ; close cycle.

   ret
; ### end complex functions ###

============================== PROGRAM END ====================================
```

```
end
```