A Structured Approach to Assembly Language Programming

SCOTT MACKENZIE

Abstract—**A method is described for teaching structured programming techniques to students of assembly language programming. Structured programming, historically, has only been within the realm of high-level languages (Pascal, C, etc.), while a more loose approach—one lacking a formal syntax—has traditionally been applied to low-level programming in assembly language. Borrowing words and symbols from Pascal and C, a simple syntax has been devised, called Pseudo Code, that uses three basic structures: linear, conditional, and loop. Upon learning that all programs can be written using only these three structures, students become convinced of the reduced complexity brought by Pseudo Code. A method is adopted that proceeds from the problem definition to the assembly language program using Pseudo Code as an interim step. Using this method, students at Seneca College in Toronto have successfully developed software in assembly language that would have been too complex for them to attempt without coding their solutions in a structured form.**

Manuscript received January 26, 1987; revised July 14, 1987.
The author is with Seneca College of Applied Arts and Technology, Toronto, Ont., Canada.
IEEE Log Number 8718357.

INTRODUCTION

Students of electronics, sooner or later, must learn to program microprocessors in assembly language. The well-developed structured programming techniques used in high-level languages (the techniques that computer science students must master) are known, but are little applied in the low-level world of assembly language programming, with the result that students tend to adopt a chaotic, brute-force approach to problem solving. The code is often difficult to debug, impossible to read, and resembles unstructured Basic where programmers routinely "paint themselves into a corner," and use a GOTO statement to escape.

A pedagogy is presented that provides students with a systematic approach to assembly language programming, one which adheres to a small but complete set of structures. The method is not exhaustive; its intent is to introduce the concept of structuring while learning assembly language programming. Using this method, students at Seneca College have successfully tackled complex programming problems in assembly language.

The following paragraphs describe a "method"; little is new except the packaging. The key to success is to simplify the complex, to give shape and form to a problem, and not to expect too much too soon. The method is presented to the students in three stages, beginning with the rudiments of structuring, then progressing to subroutines and parameter passing, and then finishing with a polished syntax. Many programming exercises are given to the students at each level before progressing to the next. Five sample exercises are given as representative of the problems that students can be expected to solve using this method.

The structures are presented to the students as the constituent parts of a small hypothetical language which we call *Pseudo Code*. This language borrows words and symbols from Pascal and C, so as to strike a balance between legibility and brevity. Pseudo Code exists purely on paper and is used only as an interim stage in problem solving. While forcing a strict adherence to structure through the use of keywords and indentation, the language places statements and conditions in square brackets and encourages students, at least initially, to use whatever wording they feel appropriate to describe operations and conditions.

THE USE OF FLOWCHARTS

Flowcharts are used initially but become optional after students develop problem-solving cognition and master Pseudo Code. Solutions are reached by progressing from the problem definition to the assembly language program via Pseudo Code, using a flowchart if necessary. This is illustrated in Fig. 1.

The disadvantage of flowcharts is that they are bulky and unruly: small routines take entire pages, they cannot be typed into a computer using word processing software, and they are difficult to edit. Their advantage lies in the shape they give to a solution by using decision blocks and flow arrows to enhance the visual representation. Parallel operations are shown as such by juxtaposing statement blocks on the page. This visual property of flowcharts, which does not exist in programming languages due to their line-by-line notation, is invaluable to many students. Flowcharts are eventually

0018-9359/88/0500-0123$01.00 © 1988 IEEE

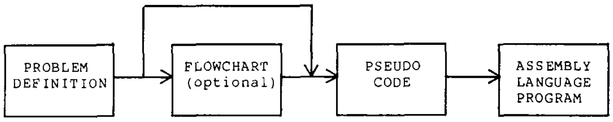


Fig. 1. The stages in problem solving use Pseudo Code as an interim step between the problem definition and the final program. Flowcharts are used initially but are discarded or used optionally after students become familiar with Pseudo Code.

discarded when students demonstrate proficiency in proceeding directly from the problem definition to Pseudo Code.

Step 1) Rigid Structures; Liberal Use of Language: Students are gradually introduced to Pseudo Code by allowing them to notate their solutions in comfortable terms, while adhering to the available structures. Before proceeding to the sample exercises, the structures are defined and illustrated in Pseudo Code and flowchart form. (The use of flowcharts will not be elaborated on further.)

Three structures are sufficient to solve any programming problem: the *linear structure*, the *loop structure*, and the *choose structure* [1]. The linear structure, shown in Fig. 2, is the familiar "statement"—the workhorse of computer programs. Via statements, programs accomplish their tasks.

The loop structure repeatedly performs an operation until a terminating condition becomes true. The two common arrangements, which are also called "statements," are WHILE/DO and REPEAT/UNTIL. These are illustrated in Fig. 3.

A WHILE/DO structure is required when the operation should not be performed in the event that the terminating condition already exists. On the other hand, the REPEAT/UNTIL structure is used when the operation must be performed at least once.

Although any WHILE/DO statement can be rearranged into a REPEAT/UNTIL statement and vice versa, both are used in Pseudo Code since they are commonplace and translate easily into assembly language. Another variation of the loop structure is the FOR statement which we avoid to keep the language small.

A common programming bug is an infinite loop. Students can guard against this by verifying that at least one operation in the loop affects the terminating condition. For example, if a counter is used, then the counter value must be affected (typically, it will be incremented or decremented) by an operation within the loop.

The choose structure, shown in Fig. 4, is the IF/THEN/ELSE statement. The CASE statement, which can be constructed from IF/THEN/ELSE statements, is not used.

The few rules adhered to at this stage are as follows.

1) Enclose conditions and statements in brackets [ ] and use any convenient language to describe the operation.

2) Statements within a CHOOSE or LOOP structure are indented to the next tab stop.

3) Multiple statements in the choose or loop structures are bracketed by BEGIN/END. (Note: Not necessary for REPEAT/UNTIL.)

4) Any structure can be inserted into the statement block of any other structure.

5) Keywords are WHILE, DO, REPEAT, UNTIL, IF, THEN, ELSE, BEGIN, END, AND, OR, and NOT.

6) Keywords are written using uppercase characters; all other words are written using lowercase characters.

7) Machine-dependent language should be avoided (i.e., use terms like "pointer" rather than "index register").

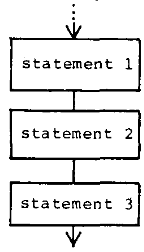
8) Use the Commercial At sign (@) to indicate indirect addressing.

9) Enclose comments within "/*" AND "*/". (For example: /* this is a comment */.)

After students have learned the instruction set of the microprocessor and are capable of writing programs using conditional branch instructions, they are ready to structure their solutions using Pseudo Code. Initially, the most appropriate problems are those that result in self-contained programs. Subroutines and parameter passing are introduced at Step 2).

In this note, the solutions to exercises are given in Pseudo Code and in the assembly language of Intel's 8051 Microcontroller,

FLOWCHART:

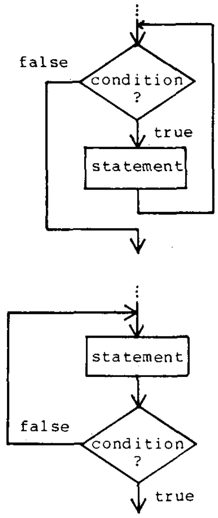


PSEUDO CODE:

```
[statement 1]
[statement 2]
[statement 3]
etc.
```

Fig. 2. The solution to all programming problems can be expressed using only three structures. The linear structure—the statement—is the workhorse of computer programs.

FLOWCHART:



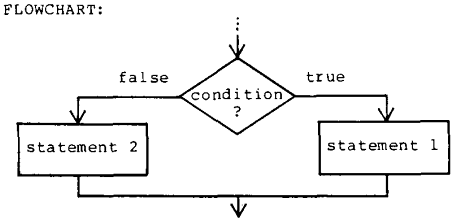
PSEUDO CODE:

```
WHILE [condition]
    DO [statement]
```

```
REPEAT [statement]
    UNTIL [condition]
```

Fig. 3. The loop structure is used to repeatedly execute a block of code. The REPEAT variation executes the statement block at least once, whereas the WHILE variation checks the terminating condition before the statement is executed.

FLOWCHART:



PSEUDO CODE:

```
IF [condition]
    THEN [statement 1]
    ELSE [statement 2]
```

Note: statement 2 is optional

Fig. 4. The choose structure executes one of two statement blocks depending on a true/false condition. The ELSE statement is optional.

which is used in courses at Seneca College. The benefit of using machine-independent terminology becomes apparent when switching to a different microprocessor. Notice that the Pseudo Code solutions do not suggest a particular target machine.

Example 1: Write a program to add a series of bytes and store the result. The length of the series is in memory location 41H and the series begins starting at memory location 42H. Store the SUM in memory location 40H.

Example 2: Search a null-terminated string of ASCII codes and count the number of digit characters ("0"–"9"). The string is stored in memory beginning at location 50H. Put the count in the accumulator.

The Pseudo Code and assembly language solutions are shown in Figs. 5 and 6. Note that the solution to Example 1 works for a zero-length series since the WHILE/DO structure checks the terminating condition before an addition is performed.

With a bit of coaching and more exercises, students should be able to design the Pseudo Code solutions easily. The translation to assembly language, however, requires considerable focus and, to assist, Pseudo Code statements should be used as comments in the assembly language program. This establishes a line-for-line correlation between the Pseudo Code and the assembly language program. For those students using a personal computer and word processor, the assembly language program can be written by editing the Pseudo Code file and inserting assembly language instructions into each line while pushing the Pseudo Code statements to the right into comments. This approach greatly simplifies the translation to assembly language.

The conditional sections of the structures are the most critical. It is in their use of conditional branch instructions that students falter, a problem arising out of the disparity between a microprocessor's instruction set and the way humans think and use language. This is particularly evident in Example 2 where a compound condition is required. An effective approach uses language within the condition brackets that, although not machine dependent, suggests the type of instructions that will be used during the translation to assembly language. Hence, in Example 2, the IF condition is stated as

IF [character > = "0" AND character < = "9"]

rather than

IF [character is a digit].

Although the latter is more akin to the way we think, it does not give any hint of the instructions required to implement the condition.

A message to the "byte counters": the argument that most of these solutions can be rearranged with a slight reduction in size is conceded; however, this must be weighed against the loss of code clarity and the loss of structure. This method is intended primarily for students of electronics using microprocessors (or microcontrollers) for "small" applications. These students are not writing code for file servers and compilers; they are writing code that interfaces microprocessors to terminals, printers, and other I/O devices; they are writing code to read from inputs, manipulate bits and bytes in some way, and write to outputs. With the high-capacity memory IC's available today, there is no need to shoehorn code into the smallest possible space. At Seneca College, students have designed microprocessor-based hardware and software for robotic arms, pen plotters, logic analyzers, etc.—the firmware required has never exceeded the capacity of a single EPROM.

Step 2) Modular Programming: Programming at the introductory level will likely be carried out in parallel with lectures introducing students to subroutines and parameter passing—the main ingredients of modular programming. Modular programming and structured programming are two mutually beneficial approaches to programming. Modules are subroutines with explicitly defined entry and exit conditions that exist in a hierarchy with complex modules building upon and using simple modules. Complex modules will "call" simple modules, passing parameters to them or receiving results back. At this level, all parameter passing uses the microprocessor's internal registers.

```
/* Example 1: Pseudo Code */

BEGIN
   [initialize pointer to 42H]
   [initialize counter from location 41H]
   [clear sum]
   WHILE [counter not equal zero] DO BEGIN
     [add @pointer to sum]
     [increment pointer]
     [decrement counter]
  END /* while */
  [store sum in location 40H]
END /* example 1 */


; Example 1: Assembly Language
;
          MOV  R0,#42H      ;initial pointer to 42H
          MOV  R7,41H       ;initialize counter from 41H
          CLR  A            ;clear sum
WHILE:    CJNE R7,#0,DONE   ;counter not = zero do begin
          ADD  A,@R0        ;add @pointer to sum
          INC  R0           ;increment pointer
          DEC  R7           ;decrement counter
          SJMP WHILE        ;end while
DONE:     MOV  40H,A        ;store sum in location 40H
HERE:     SJMP HERE
          END               ;example 1


          Listing 1
```

Fig. 5. Pseudo Code and assembly language solution for Example 1.

```
/* Example 2: Pseudo Code */

BEGIN
   [init pointer to 0050H]
   [init count = 0]
   REPEAT
      [char = @pointer]
      [increment pointer]
      IF [char >= '0' AND char <='9']
         THEN [increment count]
  UNTIL [char is 00H]
  [store count in accumulator]
END /* example 2 */


; Example 2: Assembly Language
;
EXAMPLE2: MOV  DPTR,#50H         ;init pointer to 0050H
          MOV  R7,#0             ;init count = 0
REPEAT:   MOVX A,@DPTR           ;char = @pointer
          INC  DPTR              ;increment pointer
IF:       CJNE A,#'0',$+3        ;if char >= '0' AND
          JC   UNTIL             ;
          CJNE A,#'9'+1,$+3      ; char <= '9'
          JNC  UNTIL             ;
THEN:     INC  R7                ;then increment counter
UNTIL:    CJNE A,#0,REPEAT       ;char is 00H
          MOV  A,R7              ;store count in acc
HERE:     SJMP HERE
          END                    ;example 2


          Listing 2
```

Fig. 6. Pseudo Code and assembly language solution for Example 2.

The following rules are added.

10) All modules begin with the module name followed by a pair of parentheses containing the names of parameters (if any) passed to the module.

11) All modules end with the keyword RETURN followed by a pair of parentheses containing the names of parameters (if any) returned by the module.

12) Module names are written using uppercase characters.

Although only the concept of modular programming has been added, a considerable leap forward has occurred. Students are now creating larger programs each centered around one main module with a hierarchy of modules below.

Many useful subroutines can be written by the students as exercises, leaving them with a library of routines useful to them as they advance towards high-level applications. Standard C functions [2] serve as excellent programming problems in Pseudo Code with subsequent translation to assembly language. A brief description of some of these follows. *Character class testing* subroutines are those that enter with an ASCII character in the accumulator, perform a test on the character, and return with a flag bit—typically the carry flag—set if the test passed or cleared if the test failed (Table I). *Code conversion* subroutines enter with a code in the accumulator, perform a conversion on the code, and return with the converted code in the accumulator (Table II). *String manipulation* subroutines perform operations on null-terminated strings—strings of ASCII codes terminated with a null byte (00H). These subroutines are entered with one or two pointers to strings and perform an operation on the string(s), such as a copy or compare (Table III).

As Pseudo Code is a learning tool rather than a real or "compilable" language, it does not include many of the features of high-level languages. The absence of data types, local versus global parameters, arrays, etc., is not a concern since the applications are usually small and hardware oriented, dealing with interfacing and control rather than with data processing.

Example 3: Write a subroutine called INLINE that inputs a line of characters from the console (echoing back each character as it is received) and places them in memory starting at location 60H. Maximum line length is 31 characters including the carriage return. Put 0 at the end of the line. Assume the existence of INCHAR and OUTCHAR subroutines that input and output characters to the serial port using the accumulator.

Example 4: Write a subroutine called HTOA that performs hex to ASCII conversion. A hex nibble is passed to the subroutine in the accumulator with the ASCII equivalent returned in the accumulator (example: input = 0BH, output = 42H).

The solutions are shown in Figs. 7 and 8. Since the students are now writing subroutines that are general in nature and useful for many applications, they should learn to include an appropriate comment block at the beginning of each subroutine. The comment block should provide the following information: 1) name of the subroutine, 2) entry conditions, 3) exit conditions, and 4) name of other subroutines used. Registers used for temporary storage should be saved on the stack at the beginning of the subroutine and restored from the stack at the end.

Step 3) Polishing the Syntax: When students are comfortable with structures and Pseudo Code, it is useful to develop a more cryptic and more consistent coding technique. The definition of Pseudo Code is now completed by supplying a set of operators and a precedence scheme. The operators are taken from standard C (see [2]), with some of the more esoteric ones omitted. Conspicuous variances include the use of the Commercial At (@) for address indirection, and the absence of the auto-increment and auto-decrement operators. These differences exist purely to fine tune Pseudo Code to the programming model of the target machine, in this case the 8051. This may seem in contradiction with the machine independence sought after; however, since the final objective is to generate assembly language programs, this liberty is justifiable. Undoubtedly, the coding rules should cater to some extent on the particular microprocessor used.

The operator set is given in Table IV and the precedence scheme is given in Table V.

A detail that confuses students initially is the difference between *relational operators* and *bitwise logical operators*. Bitwise logical operators are generally used in assignment statements such as the ampersand in

[lowernibble = byte & 0FH]

while relational operators are generally used in conditional expressions, such as the double ampersand in

IF [char != "*Q*" && char != 0DH] THEN · · · .

As well, the relational operator "==" should not be confused with the assignment operator "=". For example, the Boolean expression in

IF [j == 9] THEN · · ·

is either true or false, depending on whether or not j equals 9, whereas the assignment statement

[j = 9]

sets j equal to 9. This difference requires a slight adjustment in the coding practices used previously.

Example 5: Write a subroutine called PSTR to send a null-terminated string to the system printer expanding tabs with spaces. Assume tab stops after every eight columns, i.e., at columns 9, 17, 25, etc. Assume the existence of a subroutine called PCHAR to print the character in the accumulator. Assume the string is in memory at an address passed to PSTR.

The solution to this example is shown in Fig. 9. As can be seen, this subroutine is tricky enough that coding directly in assembly language without the assistance of a flowchart or Pseudo Code would be a formidable task. The Pseudo Code solution is concise and clearly shows the structure of the problem. Proceeding from the problem definition to Pseudo Code is one step in the solution; translating the Pseudo Code into assembly language is another. Each of these tasks is far simpler than the combined task of direct coding in assembly language.

TABLE I
CHARACTER CLASS TESTING SUBROUTINES

Name	Exit with carry = 1 if ...
ISALPH	char in range 'a' to 'z' or 'A' to 'Z'
ISDIGT	char in range '0' to '9'
ISHEX	char in range 'a' to 'f' or 'A' to 'F'
ISGRPH	byte in range 20H to 7EH (ASCII graphic)
ISWHIT	char is tab (09H) or space (20H)
ISUPPR	char in range 'A' to 'Z'
ISLOWR	char in range 'a' to 'z'

Note: Enter with ASCII code in accumulator

TABLE II
CODE CONVERSION SUBROUTINES

Name	Operation	Example Enter	Example Exit
HTOA	Hex TO Ascii	A = 0FH	A = 46H
ATOH	Ascii TO Hex	A = 41H	A = 0AH
UTOL	Uppercase TO Lowercase	A = 5AH	A = 7AH
LTOU	Lowercase TO Uppercase	A = 62H	A = 42H
BCDBIN	BCD to BINary	A = 29H	A = 1DH
BINBCD	BINary to BCD	A = 0FH	A = 15H

Note: A is the accumulator

TABLE III
STRING MANIPULATION SUBROUTINES

Name	Enter	Exit
STRLEN	P1	length of string P1 in A
STRCPY	P1, P2	string P2 copied to string P1
STRCAT	P1, P2	string P2 catenated to end of string P1
STRCMP	P1, P2	lexicographical comparison: A = @P1 - @P2 until end of strings or A != 0

Note: P1 and P2 are pointers to null-terminated ASCII strings
Note: A is the accumulator

```
/* Example 3: Pseudo Code */

INLINE()
   [pointer = 60H]
   [length = 31]
   REPEAT
      [input char from serial port]
      [echo back to serial port]
      [@pointer = char]
      [increment pointer]
      [decrement length]
   UNTIL [length is 0 OR char is 0DH]
   [@pointer = 0]
RETURN()


; Example 3: Assembly Language
;
;**************************************************
; INLINE: input a line of characters
;
; ENTER:  -no conditions
; EXIT:   -ASCII codes in memory beginning at 60H
;         -maximum line length = 31 characters
; USES:   -INCHAR, OUTCHAR
;
XR0      EQU   0              ;direct add of R0
XR7      EQU   7              ; and R7
INLINE:  PUSH  XR0            ;save registers on
         PUSH  XR7            ; stack
         MOV   R0,#60H        ;pointer = 60H
         MOV   R7,#31         ;length = 31
REPEAT:  ACALL INCHAR         ;input char from port
         ACALL OUTCHAR        ;echo back to port
         MOV   @R0,A          ;@pointer = char
         INC   R0             ;increment pointer
         DJNZ  R7,SKIP        ;decrement length
         SJMP  EXIT           ;until length is 0 OR
SKIP:    CJNE  A,#0DH,REPEAT  ;char is 0DH
EXIT:    MOV   @R0,#0         ;@pointer = 0
         POP   XR7            ;retrieve registers
         POP   XR0            ; from stack
         RET


         Listing 3
```

Fig. 7. Solution for Example 3.

```
/* Example 4: Pseudo Code */

HTOA(code)
   IF [code >= 0AH]
      THEN [add 37H to code] /* in range A TO F */
      ELSE [add 30H to code] /* in range 0 TO 9 */
RETURN(code)


; Example 4: Assembly Language
;
;**************************************************
; HTOA: hex to ASCII conversion
;
; ENTER:  -A contains hex nibble in range 00-0F
; EXIT:   -ASCII equivalent code in A
;
HTOA:    CJNE  A,#0AH,$+3     ;if code > 0AH
         JC    ELSE
         ADD   A,#37H         ;then add 37H
         RET
ELSE:    ADD   A,#30H         ;else add 30H
         RET


         Listing 4
```

Fig. 8. Solution for Example 4.

Conclusion

A further benefit of Pseudo Code is that the students learn the rudiments of high-level languages while studying assembly language programming. There is a tendency for high-level language programmers to consider their language as a sort of black box or, perhaps more appropriately, a form of black magic. Little consideration is paid to the machine and how programs execute at the machine level. A compiler performs some inexplicable translation and somehow a program is generated that performs the desired operation. This is fine in many instances, but for students of electronics, it is important to remain in contact with the hardware. Time-dependent and I/O intensive operations often demand the closeness offered by assembly language programming. Pseudo Code allows programmers to retain this closeness while using a structured technique. When these students begin programming in Pascal or C, they make the transition smoothly, knowing how the language performs and how it drives the hardware.

What are the students' views on the approach presented here? This was discovered many months after the technique was taught when students, working on their term design projects, were found to be sketching out their software routines in Pseudo Code prior to coding in assembly language. It seems they were sold on the idea. The order brought to their software through structuring made the extra step worthwhile.

The method presented in this paper has proven to be effective in teaching assembly language programming. Students at Seneca College have successfully implemented microprocessor-based designs requiring complex software. The resultant code, all written in assembly language, is concise, simple to read and debug, and most

TABLE IV
OPERATORS USED IN PSEUDO CODE

Class of Operator	Symbol	Operation
Arithmetic	+	addition
	-	subtraction
	*	multiplication
	/	division
	%	modulus (remainder after division)
Relational	==	true if values equal
	!=	true if values not equal
	>	true if first value greater than second
	>=	true if 1st value greater or equal to 2nd
	<	true if first value less than second
	<=	true if 1st value less than or equal to 2nd
	&&	true if both values are true
	||	true if either value is true
Bitwise Logical	&	logical AND
	|	logical OR
	^	logical exclusive OR
	~	logical NOT (1's complement)
	>>	logical shift right
	<<	logical shift left
Assignment	=	set equal to
	op=	assignment shorthand where 'op' is any arithmetic or bitwise logical operator (e.g.: j = j + 4 can be coded j += 4)
Precedence Override	()	see Table V
Indirect Address	@	variable following is address of operand

TABLE V
OPERATOR PRECEDENCE

Operator	Precedence
()	highest
~ @	
* / %	
<< >>	
< <= > >=	
== !=	
&	
^	
|	
&&	
||	
= += -= *= etc.	lowest

```
/* Example 5: Pseudo Code */

PSTR(pointer)
   [cc = 0] /* cc is column counter */
   WHILE [ (char = @pointer) != 0) DO BEGIN
      IF [char == tab] THEN
         REPEAT
            [send space to printer]
            [cc += 1]
         UNTIL [cc % 8 == 0]
      ELSE BEGIN
            [send char to printer]
            [cc += 1]
      END /* if */
      [pointer += 1]
   END /* while */
RETURN()

; Example 5: Assembly Language
;
;**************************************************
; PSTR: print string expanding tabs with spaces
;
; ENTER:  DPTR points to string
; EXIT:   string send to printer
; USES:   PCHAR
;
XR7       EQU   7               ;direct add of R7
PSTR:     PUSH  XR7             ;save registers on
          PUSH  DPH             ; stack
          PUSH  DPL
          PUSH  ACC
          MOV   R7,#0           ;cc = 0
PSTR2:    CLR   A               ;while ...
          MOVC  A,@A+DPTR       ;char = @pointer
          JZ    PSTR5           ;!= 0 do begin
          CJNE  A,#09H,PSTR4    ;if char == tab then
PSTR3:    MOV   A,#20H          ;send space to prntr
          ACALL PCHAR
          INC   R7              ;cc += 1
          MOV   A,R7
          ANL   A,#07H          ;until cc % 8 == 0
          JNZ   PSTR3
          SJMP  SKIP
PSTR4:    ACALL PCHAR           ;else begin send char
          INC   R7              ;cc += 1
SKIP:     INC   DPTR            ;pointer += 1
          SJMP  PSTR2           ; ... repeat while
PSTR5:    POP   ACC             ;retrieve registers
          POP   DPL             ; from stack
          POP   DPH
          POP   XR7
          RET

          Listing 5
```

Fig. 9. Solution for Example 5.

importantly, the software was written using the technique of structured programming.

REFERENCES

[1] L. Leventhal, *6800 Assembly Language Programming*. Berkeley, CA: Osborne/McGraw-Hill, 1978, ch. 13.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.