LA-6634-MS
Informal Report

UC-15 and UC-32

Issued: December 1976

A Self-Patching Firmware Program

by

*Luciano Stanchi

*LASL Long-Term Visiting Staff Member. Euratom Joint Research Center, Ispra, Italy.

los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87545

An Affirmative Action/Equal Opportunity Employer

UNITED STATES
ENERGY RESEARCH AND DEVELOPMENT ADMINISTRATION
CONTRACT W-7405-ENG. 36

Printed in the United States of America. Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy $3.50 Microfiche $3.00

A SELF-PATCHING FIRMWARE PROGRAM

by

Luciano Stanchi

ABSTRACT

A method of branching to an auxiliary program without any hardware or software modification at the time the auxiliary program will be written is described. Suitable statements in the main program are already prepared to accept a future program that will be written in machine language. The main program is stored as firmware in PROM memory while the auxiliary program is loaded into RAM memory. The method is described for the Intel 8080 microprocessor but can be generalized for any microprocessor or minicomputer that uses ROMs or PROMs.

I. INTRODUCTION

Systems based on microprocessors normally have their program stored in firmware (ROMs or PROMs). Modifications of the program due to new requirements are normally awkward. In minicomputer practice, on the contrary, it is quite easy to add a new part branching to an unused block of memory, if any, because they normally have provision to alter core memories by front panel operation.

It should be amazing to have the same possibility with microprocessors but, unfortunately, read-only memories cannot be partially modified to insert the branch instruction and must be totally reprogrammed. In case of ROM no possibility is given. In case of PROM the memory can be utilized with the onerous price of reassembling completely the program with erasing and successive rewriting.

The problem deserves a more accurate attempt to search for a general solution. A simple method of providing an easy modification avoiding the need of changing the read-only memory (ROM or PROM) should be highly desirable.

ROM: read-only memory.

PROM: programmable read-only memory.

RAM: random access memory.

A hardware approach which provides some modification in the circuitry for skipping unwanted parts of the program and branching to RAM memories by means of a program interrupt has been described.[1]

Here a software method that uses some codes in the ROM to provide the possibility of a future addition or modification of the program is presented.

This program has been developed for a system based on the Intel 8080 microprocessor, but the concept can be generalized with some rearrangement for use with any device. Therefore, the following will describe the program that has been developed for a particular instrument, hoping that it will be useful for other designers that will conceive systems based on any firmware program.

II. GENERAL DESCRIPTION

The system has been outlined in a preceding report.[2] A more detailed description can be found in Ref. 3. The purpose of this report is to enhance the part of the program concerning the statements used for "patching" the firmware as an item of general application. For the sake of clarity a short description of the complete equipment is first presented so that the problem is properly framed.

The instrument is microprocessor-based equipment that operates a continuous monitoring of effluents from a nuclear facility. Detectors send data to the "microcomputer" via suitable analog chains. The CPU (Central Processor Unit) evaluates the data and takes actions accordingly. During normal operation the instrument is only watching at the effluent without taking any particular action. Only in case of alarm due to excessive radioactivity or malfunction discovered by the self-diagnosing capability does the instrument carry out some intervention. In the routine operation it performs only an evaluation that is expected to be within the prefixed limits. Several variables are accessible to the operator and stored in a RAM memory. The program is stored in a 4k byte-PROM memory.

The instrument utilizes an Intel single-board computer type SBC 80/10. PROM or ROM memories can be assembled on the board up to a total of 4k bytes, and a 1k RAM is already provided by the manufacturer. Serial and parallel input/output assembled on the board are well matching the necessity of the system.

The instrument has some degree of complexity and can presumably reveal some new requirements when the facility will be fully operated. For that purpose a means of easy modification of the program has been conceived.

III. THE AUXILIARY PROGRAM

The main program has been obtained by means of a PL/M compiler that operates on a source program written in PL/M language. The program has been developed as a series of declarations followed by some executable statements and a series of procedures that are called by program interrupt. The conception is to leave the processor in HALT condition waiting for external interrupts mainly due to pulses related to the radioactivity of the effluents or to clocks determining the various phases.

The interrupts will call the different procedures that will automatically perform the desired action. Some procedures are connected to the intervention of the operator via a keyboard.

One of these procedures that can be called by the operator typing "OPER" on the keyboard or simply the initial "O" is the object of the present report. This procedure will allow the operator to write an

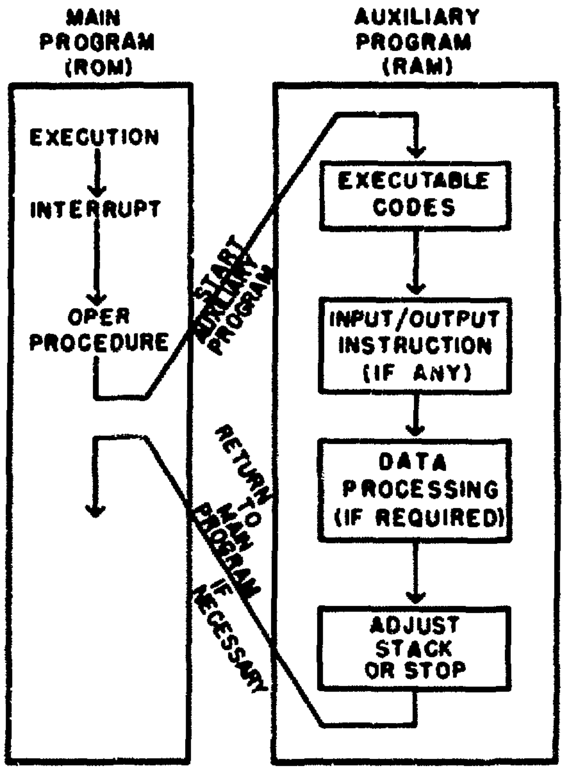

Fig. 1. Pictorial show of the auxiliary program operation.

auxiliary program that will be stored in the RAM of the single-board computer. Up to 511 bytes can be written by the operator. When the acquisition is terminated, the operator types a slash (/) and the auxiliary program is automatically executed.

Figure 1 shows schematically the operation. The OPER procedure acquires the auxiliary program as it will be described. Then, the auxiliary program is started and performs what it is instructed to do by its executable codes. Input/output and data processing are performed if requested. The end of the program can be either a stop or a return to the main program, having suitably loaded the stack before returning.

This auxiliary program can be a test program written for the setup of a future addition or a small modification of the hardware that does not affect the main program operation. It can also be used to test some parts of the hardware during the life of the instrument with the purpose of ascertaining the circuit behaviour.

More generally the auxiliary program can be considered as an integrating part that can be operated independently or in conjunction with the main program, having the sole difference of being stored into RAM instead of PROM (ROM) as for the main program.

IV. OPERATIVE FEATURES

The operator types a series of hexadecimal codes corresponding to the bytes in machine language. The Intel codes are referred to, but any other microprocessor will behave similarly. It is quite generally accepted practice in the microprocessor operation to write codes in hexadecimal form. In effect the hexadecimal form is more synthetic than octal form and well matched to 8-bit bytes. Unfortunately, input/output terminals are based on ASCII code that is not particularly suited for hexadecimal form.[4]

The main program allows the operator to type the hexadecimal digits consecutively or grouped by bytes (two-hex digits) intercalated by spaces, if desired, for better readability. The main program simply ignores the spaces as well as the carriage return and line feed so that the operator can write the program using the architecture that he likes, e.g., with lines of 16 adjacent bytes as made by the Intel compiler or with lines of 16 or 10 bytes with spaces.

If the operator presses accidentally a key different from one of the 16 hex digits (Ø to F) the program types immediately the message "ILLEGAL CHARACTER," and there is no store into memory. The program can be immediately resumed starting from the last correct byte.

If the operator recognizes a wrong code in the byte that he has just typed, he can go back of one position by pressing a period (.). This will allow the rewriting of the wrong byte over the already stored wrong code. Pressing repeatedly the "." key can be used to go back of some bytes to retrace a preceding wrong code. All the successive bytes have then to be rewritten.

When the series of codes constituting the auxiliary program is finished, the operator types a slash, as stated, and the program is immediately executed. The starting address of the program has been assigned by the compiler at the hexadecimal address 3CFØ. The source program only assigns the maximum length of the array.

V. DETAILED DESCRIPTION OF THE PROGRAM

Even if the section of the main program dealing with the acquisition of the auxiliary program is self-explanatory because of the comments generously inserted between statements, a minute guideline is reported here. To understand this chapter some knowledge of PL/M language is necessary.

Two procedures are used for the acquisition and start of the auxiliary program:

1) IB (input byte) which acquires the ASCII characters and composes them two by two into bytes.

2) OPER which forms the program assembling the bytes in a suitable array named OPARRAY. The calling of this procedure is made by the operator typing OPER (or simply O) plus CR (carriage return).

The two procedures are reported thoroughly in Table I and Table II from the part of list 1 of the compiled program that reflects the source program.

The OPER procedure (Table I) sets a pointer S for the array of bytes to be loaded. S starts from Ø and is incremented after the acquisition of each byte. The call of the procedure IB is simply made by the statement "OPARRAY(S)=IB" contained in line 774. Up to 511 bytes are allowed so that the last S can reach the value of 510. If this value is exceeded, there is a call to the "Message 17" (ME17) that warns the operator, typing "STRING TOO LONG." The operator must stop in this case and reconsider the program. There is no protection beyond this limit: the variable S is still incremented and the operator can invade the part of memory assigned to other variables. Note that S=511 is already too long because the compiler assigned the address of OPARRAY(Ø)+511 to another variable. As a matter of fact, OPARRAY starts at 3CFØ and goes up to 3CFØ+51Ø. If S=511 (Hex 1FF) the obtained value is 3CFØ + 1FF = 3EEF and this address is assigned for a variable in the following "PULSES" procedure. Thus in case the operator receives the warning of exceeding length, he must at least go back of one byte before typing the slash that fixes the end of the acquisition.

TABLE I

OPER PROCEDURE

```
00761  1:
00762  1    /* ************************************************** */
00763  1:
00764  1     /*14*/         OPER: PROCEDURE;
00765  2:
00766  2    /*OPER PROCEDURE ALLOWS THE OPERATOR TO WRITE AN EXECUTABLE
00767  2    PROGRAM IN MACHINE CODE. UP TO 511 BYTES ARE ALLOWED AND THE
00768  2    PROGRAM IS  TERMINATED BY PRINTING A SLASH +/+.*/
00769  2:
00770  2     DCL OPARRAY(511) BYTE;
00771  2:
00772  2                 S=0;
00773  2                 DO WHILE 1>0;
00774  2                 OPARRAY(S)=IB;          /*FIRST S: S=0*/
00775  3                 IF S=511 THEN        /*LAST S: S=510*/
00776  3           CALL PSCR(.ME17,LENGTH(ME17));   /*STRING TOO LONG*/
00777  3                 S=S+1;
00778  3                 END;
00779  2:
00780  2               END OPER;
00781  1:
00782  1    /*OPER HAS AN INFINITE LOOP THAT IS INTERRUPTED BY TYPING
00783  1    A +/+. THIS CAUSES A JUMP TO 3CECH WHERE THE OPERATOR PROGRAM
00784  1    WILL START. THE OPERATOR PROGRAM (IF NOT AN INFINITE LOOP
00785  1    ITSELF) WILL RETURN TO PROCEDURE RECO IF A RET STATEMENT
00786  1    IS FOUND. OR ANYWHERE IF IT CONTAINS A SUITABLE STATEMENT
00787  1    FOR JUMP OR FOR ALTERING THE STACK.
00788  1    ESCAPE FROM A TEST LOOP CAN BE DONE WITH POWER OFF.*/
00789  1:
00790  1:
00791  1:
00792  1    /* ************************************************** */
```

The IB procedure (Table II) is somewhat more complicated. In the program it appears before the OPER procedure. It contains an "Iterative Do" made of two steps that is used to acquire two ASCII characters and to combine them together into a byte.

The character obtained calling procedure IC (input character-call contained in line 715 of Table I") is first recognized and then one of the following actions is taken:

a) If the character is a CR (Carriage Return) or LF (Line Feed) or 2ØH (ASCII value for space) no action is taken. The stack pointer is increased by 2 as for a return from a call and an absolute jump to the calling point of the IB procedure in the OPER procedure is performed. This means that any one of the above characters is considered as not occurring, and the string can continue undisturbed.

b) If the character is not one of the above mentioned and lies outside of the groups listed on line 726, the action is again irrelevant as before, but here a message "ILLEGAL CHARACTER" is typed. The listed groups contain the hexadecimal digits "Ø" to "9" and "A" to "F" plus two auxiliary characters that have been chosen as "." and "/" for the only reason that they are listed in the ASCII table immediately before the value of "Ø" and therefore can be easily grouped.

c) If the character is "." (ASCII value 2E, see line 733), the action is to go back in the array assigned to the auxiliary program. This is the only action that is taken before returning to the calling point as for point a . The acquisition is not performed. The useful action due to this character is used for correcting wrong codes going back of as many bytes as dots are typed.

d) If the character is "/" (ASCII value 2F, line 742), the acquisition is terminated. The control is given to the auxiliary program that starts at address 3CFØ and is immediately executed.

e) If the character is a hexadecimal number (line 744), the "Iterative Do" performs the acquisition and assembly of two characters into one byte. The characters have to be stored as binary numbers so the ASCII value must be decreased accordingly. Numbers "Ø" to "9" contain an addition of 3ØH in ASCII value so this value is subtracted. Numbers

TABLE II

IB PROCEDURE

```
1:
1    /* ********************************************** */
1:
1     /*13*/          IB: PROCEDURE BYTE;
2:
2       /* IB STANDS FOR INPUT BYTE */
2    /*ONE BYTE IS ACQUIRED WHEN THIS PROCEDURE IS CALLED BY OPER.
2    THE OPERATOR TYPES '/' AFTER THE FINAL BYTE AND OPER WILL GIVE
2    CONTROL TO THE FIRST EXECUTABLE CODE STORED AS OPARRAY(0).*/
2    /*ASCII CHARACTERS ARE TRANSFORMED INTO HEX CHARACTERS.*/
2:
2      DCL I BYTE;
2      DCL PINC(2) BYTE;
2      DCL PALL BYTE;
2:
2       DO I=0 TO 1;            /*START ITERATIVE DO*/
2:
2     /*OPERATOR MAY TYPE CR,LF WHEN END OF A LINE IS REACHED.
2    OBVIOUSLY A 2-CHARACTER BYTE CANNOT BE DIVIDED.*/
2    /*OPERATOR CAN TYPE A SPACE BETWEEN BYTES BUT THIS IS NOT CON-
2    PULSORY. SPACES WILL BE NEGLECTED. IF OPERATOR TYPES A WRONG
2    LEGAL CODE, HE CAN IMMEDIATELY TYPE A PERIOD ('.') AT THE END OF
2    THE BYTE TO GO BACK OF ONE BYTE AND CORRECT THE CODE.*/
2:
2     IF (CHAR(I):=IC)=CR OR CHAR(I)=LF OR CHAR(I)=20H THEN
3         DO;
3         STACKPTR=STACKPTR+2;
4         GOTO 96BH;      /*THERE IS CALL IB*/
4         END;
3     ELSE DO;              /*START ELSE DO*/
3:
3    /*ACCEPT ONLY THE 16 HEXADECIMAL CHARS PLUS '.' AND '/' */
3    /*IF THE FIRST OR THE SECOND CHAR OF A BYTE ARE ILLEGAL THERE IS NO
3    STORE. THEN OPERATOR CAN CONTINUE WITH CORRECT BYTES*/
3:
3    IF CHAR(I)<2FH OR (CHAR(I)>39H AND CHAR(I)<41H) OR CHAR(I)>46H
4    THEN DO;
4         CALL PSCR(.ME23,LENGTH(ME23));    /*ILLEGAL CHARACTERS*/
5         STACKPTR=STACKPTR+2;
5         GOTO 96BH;      /*THERE IS AGAIN CALL IB*/
5         END;
4:
4      IF CHAR=2EH THEN DO;
4                         S=S-1;
5                         STACKPTR=STACKPTR+2;
5                         GOTO 96BH;      /*THIRD CALL IB*/
5                         END;
4:
4    /*THE ABOVE 'DO' IS STARTED BY A '.' AND CAUSES TO GO BACK OF ONE
4    PLACE IN THE OPARRAY AND TO CORRECT THE WRONG CODE.*/
4:
4    IF CHAR=2FH THEN GOTO 3CF0H;    /*2FH='/',CLOSE PROGRAM*/
4:
4      IF (CHAR(I) >=30H) AND (CHAR(I) <=39H)
4          THEN PINC(I)=CHAR(I)-30H;     /*STORE 0 TO 9 NUMBERS*/
4      IF (CHAR(I) >=41H) AND (CHAR(I) <=46H)
4          THEN PINC(I)=CHAR(I)-37H;     /*STORE A TO F CHARS*/
4:
4         END;          /*END ELSE DO*/
3:
3       END;                    /*END ITERATIVE DO*/
2:
2          PALL = ROL(PINC(0),4)+PINC(1);
2              RET PALL;
2            END IB;
1:
1     /*PALL IS RETURNED TO OPER. WHEN CHAR IS '/' THERE IS NO RETURN.
1    THE STORED PROGRAM IS EXECUTED IMMEDIATELY.*/
```

"A" to "F" have values "41H" to "46H." The value to be subtracted for this second group is "37H," e.g., A = 41H - 37H = 1C. The subtraction is evident using binary notation

```
        41H)      ø1øø øøø1  -
        37H)      øø11 ø111  =
        1ø)       øøøø 1ø1ø
```

If the iterative do is completed and no absolute jump is requested, the array of two characters listed as PINC(0) and PINC(1) is composed into the byte PALL that is returned to the call point in procedure OPER and assembled in the OPARRAY.

VI. USE OF THE ABSOLUTE GO TO STATEMENTS

The jumps listed in procedure IB and explained at the points a), b), c), and d) of the preceding chapter are set into effect both for the first or for the second character (of the bytes) typed by the operator.

The use of absolute GO TO statements instead of GO TO label can be surprising. In effect, GO TO label is used in the main program as GO TO EXIT where EXIT is before declared as label.[2,3] The opportunity of using GO TO label also here should have been great, particularly for two facts:

1) During the development of the program, the corrections or additions to the preceding statements were changing the following addresses so the absolute GO TO statement had to be replaced at each modification.

2) The program has been tested in two different manners before being assembled into PROM. First, the program has been tested with a software simulation using the INTERPB simulator. Secondly, a more comprehensive hardware simulation was performed using the Intel Monitor that compels the addresses foreseen for the program and for the RAM locations to move. The same addresses unfortunately were not accepted by the simulator. As a result, each time it was necessary to pass from simulator to monitor or vice versa, all the absolute GO TO statements had to be changed.

The GO TO label should be very convenient, but unfortunately this is not allowed by the PL/M language itself. As an example, if the three GO TO 9E8 were replaced by three GO TO BYTLAB statements, where BYTLAB is the label for the call IB procedure, the program should have been written as follows.

First, declaration of the label

```
        DCL BYTLAB LABEL;
```

Then the GO TO BYTLAB in IB procedure and finally line 774 should have been indicated as the destination for the jump (GO TO BYTLAB) writing

```
        BYTLAB: OPARRAY = IB;
```

This should have been valid if the successive statements: DCL BYTLAB LABEL, GO TO BYTLAB, BYTLAB:...., were all contained in the group of executable statements of the program or in the same procedure.

Unfortunately, the GO TO statements are in IB procedure, and the destination is in OPER procedure. So the label declaration must be at the beginning with the series of declaration for the main program. But writing BYTLAB:... in the OPER procedure is taken by the PL/M compiler as an implicit label declaration. The label declaration at the beginning of the program fails to be continuous and is valid for all the program with the sole exception of OPER procedure, which is the only place where the label should be used. The compiler assigns a second address for the label BYTLAB in the procedure OPER, but this address is unknown by the external world, and the GO TO statement in effect does not know where to go. The compiler assigns an address BYTLAB=ø corresponding to the first declaration and the GO TO statement is compiled as a JUMP to zero. But no statements are made in the source program for the label outside the OPER procedure, where on the contrary there is the correct statement, but it is unreachable without a modification of the compiled program.

Instead of modifying the compiled program, an absolute GO TO number statement has been written in the source program, with the consequence of many tedious replacements.

VII. CONCLUSION

The main program is now located in PROMs and so became invariant. Even if the PROMs can be erased and rewritten, they are considered as firmware that should normally not be altered. Even more, if a series of equal instruments has to be produced, the program can be stored in nonerasable ROMs so that no future modifications are possible.

The self-patching statements are stored as an integrating part of the main program. Future additions or modifications can be performed by the auxiliary program.

The part dealing with this feature is not related in any manner to the effluent monitoring system, for which it has been developed so that it can be generalized. Any program that will be written for an Intel 8080 microprocessor by means of the PL/M language can use the listed procedures and allow patching of an auxiliary program.

More generally, any program using firmware can use a similar section in the main program written in any language to perform the same task of avoiding erasing and rewriting of the PROMs.

REFERENCES

1. T. Travis, "Patching a Program into a ROM," Electronics Design 24, No. 18, 98 (Sept. 1976).

2. L. Stanchi, "Effluent Monitoring for Nuclear Safeguard" to be published in IEEE Trans. Nucl. Sci. 24, No. 1 (February 1977).

3. L. Stanchi and M. Vasey, "Effluent and Sanitary Sewer Monitors," Los Alamos Scientific Laboratory report LA-6638-M (1977).

4. L. Stanchi, "Is ASCII Code Old?" to be published.