AN IMPROVED HANDS-ON APPROACH TO TEACHING SYSTEMS PROGRAMMING AND THE IMPACT OF STRUCTURED PROGRAMMING

Roger T. Cooper and Malcolm G. Lane

Department of Statistics and Computer Science
West Virginia University
Morgantown, West Virginia 26506

ABSTRACT

The use of the hands-on approach for teaching systems programming presented at the 1974 SIGCSE Conference has proved to be even more successful in the past two years. The reasons for the increased success are given. An approach of using structured assembler language concepts as an integral part of the systems programming course is introduced and discussed. Specific examples of the use of several structured programming macros are presented.

INTRODUCTION

There has been a dramatic improvement in the success of the hands-on approach to teaching systems programming used at West Virginia University over the past two years.[5] The increased success can be attributed to several improvements which have been made in the pedagogical techniques used in the course.

This paper will concentrate on the improvements in the course assignments themselves and on the most recent improvement, the introduction of structured programming concepts into the course. Approaches for implementing structured programming macros for teaching systems programming will be presented, and specific examples of the use of several of the macros which have been implemented will be given. Finally, the impact of structured programming (structured assembler language) on teaching systems programming will be investigated.

AN OVERVIEW OF THE COURSE

The basic principle used in the hands-on approach to teaching systems programming is that students, working in groups, will implement a small multiprogramming executive using an IBM 1130 in order to learn the basic principles in the design of such a system. In 1974 [5], the success of the approach was demonstrated by a figure which contained the information found in column 1 of Figure 1.

Figure 1 summarizes several important success indicators by comparing the first three of the course years (1971-1973) against the past four semesters. The most significant changes in the course are the decrease in group sizes, the improved quality and thoroughness of the documentation, and most importantly the percentage of the systems which were successfully implemented. These indicators will be discussed later.

IMPROVEMENTS IN COURSE PROJECTS

The systems programming class is based on a multiprogramming executive (MPX) which the students must implement in groups. The MPX assignment includes the following:

A Task Scheduler
A Command Interpreter Task
An Input/Output Supervisor
An Error Message Handler
Memory Allocation Facilities

One of the major factors which increased the success of the hands-on approach to teaching systems programming was the redesign of assignments leading up to the MPX. It was discovered that students had difficulty implementing and debugging a task scheduler in the MPX assignment.

To remedy this problem, one of the first assignments that students now undertake is the implementation of a small round-robin scheduler. Students are provided with small test tasks (subroutines) to dispatch. They must give CPU control to each task in its turn, handle the saving and restoring of registers in control blocks maintained for the tasks (via linked lists), and finally "cancel" tasks upon request and remove the control blocks from the scheduler queue. When their scheduler detects an idle state (all tasks terminated), the student scheduler must end its execution.

Early design specifications of the student IBM 1130 Multiprogramming Executive (MPX) called for the implementation of an Input/Output Supervisor. However, the early assignments for writing I/O handlers for the card reader, line printer, and keyboard/console printer did not give specifications totally consistent with the I/O supervisor needed for the MPX.[5] The solution to this problem was to rewrite the specifications for all Input/Output handlers. The resulting assignment calls for the writing of a more general Input/Output Control System (IOCS) which can be used in the MPX with

	1971-1973 (3 Classes)	1974-1976 (4 Classes)
AVERAGE CLASS SIZE	27	22
AVERAGE GROUP SIZE	3.7/GROUP	2.9/GROUP
AVERAGE NUMBER OF GROUPS	7.0	7.5
AVERAGE LENGTH OF DOCUMENTATION (Typewritten Pages)	10	53
TOTAL NUMBER OF SYSTEMS	21	30
SUCCESS OF SYSTEMS IMPLEMENTED:		
A	4	10
B	7	11
C	6	7
D	4	2
PERCENT IN CATEGORIES A & B	52%	70%

A= Number of group projects with all features implemented.
B= Number of group projects with most features implemented.
C= Number of group projects which partially worked.
D= Number of group projects which did not work.

FIGURE 1. A COMPARISON OF THE FIRST THREE YEARS OF THE HANDS-ON APPROACH TO TEACHING SYSTEMS PROGRAMMING TO THE LAST TWO YEARS

only slight (if any) modifications. This project is a group project, as is the MPX.

The course improvements indicated in Figure 1 can be attributed to several things. The major improvement was due to the redesign of the assignments discussed above. Students more thoroughly understand schedulers before undertaking the MPX. Also, a thoroughly tested IOCS can be used in the MPX and thus IOCS should (theoretically) not create major debugging problems when interfaced to MPX.

Another factor was the reduction in the number of students assigned to a group. In the particular course being discussed, it has been observed that students working in groups of two or three seem to produce better projects than those working in groups of four. Thus, recent groups have been limited to a maximum of three students, and students are encouraged to work in groups of two.

Finally, more stringent standards have been adopted for design specifications, flowcharts, and documentation produced by the students. This has dramatically improved the quality of written documentation of the student Multiprogramming Executives while at the same time forcing students to put more thought into the design of the MPX. Figure 1 illustrates that more detailed documentation is now being provided by the students.

STRUCTURED PROGRAMMING IN ASSEMBLER LANGUAGE

The use of macros to facilitate structured programming in assembler language has been undertaken in various ways in the past few years [1,2,4]. Because of the particular need to avoid many of the common errors (discussed later) that students in the Systems Programming course make in assembler language and in order to facilitate the completion of the Multiprogramming Executive project, structured programming macros for the IBM 1130 were developed.[3] The development of the macros was undertaken with the basic assumption that they were for use in the Systems Programming course. The system configuration of the IBM 1130 used in the course and for macro development is shown in Figure 2.

Many of the macros developed are peculiar to the systems programming environment for the IBM 1130. Others are general macros typically implemented to accomplish structured programming.[1,2,4]

It should be pointed out that the extreme slowness of the IBM 1130 macro assembler in resolving a macro (primarily due to the slowness of the disk drive) could detract from the use of such macros. However, it is felt that the resulting improvement in coding, debugging and maintenance of assembler programs can offset the increased assembly time.

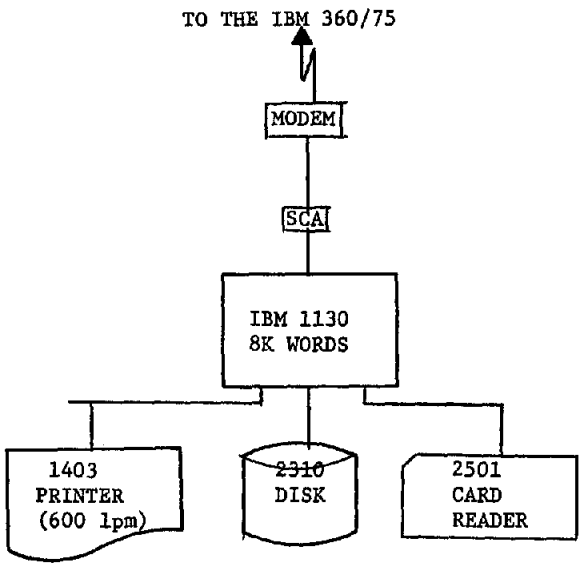

FIGURE 2. CONFIGURATION OF THE IBM 1130

FIRST IMPLEMENTATION OF MACROS

The earliest macros written for use by the Systems Programming class were called DSAB, ENAB, SAVE, and RSTO. These macros were designed to alleviate very specific problems encountered by students who were writing a Multiprogramming Executive (MPX) for the IBM 1130.[5] These macros were very successful in their restricted uses. However, they did not provide any assistance to the students who were trying to develop their first large assembler language programming assignment.

During the Fall semester of 1975, a new group of macros was added for experimental use. These macros were for defining loops and subroutines. An example of the looping control would be the DO...ENDDO block. The DO macro recognized one operand, which was the number of times that the loop was to be executed. The ENDDO closed the loop and performed the incrementing and testing. The loop count always was incremented by one. The blocks DO1...END 1 and DO2...END 2 were very similar to the DO...ENDDO block. However, the DO1 and DO2 loops also allowed for the automatic increment, by one, of the specified index register (1 or 2). The macro used to define the beginning of a subroutine was called PROC. The PROC macro saved registers upon entry to the subroutine; the code that restored the registers was also contained in the PROC macro.

Although these macros saw limited use, they proved successful when used. One of the major reasons for the lack of use of the early macros was that they were introduced after the students had begun programming. The students preferred to continue coding the way they knew, rather than learn something new. Also, the macros were not powerful enough to compensate for the fact that they slowed down assembly time. Students could not see a sufficient return for the extra assembly time.

In general, the early macros suffered from the following problems:

1) They were all post-test loops. This fact is not, in itself, bad; but the implementation was such that control information had to be divided between the beginning and the end of the block. Consequently, blocks could be difficult to understand.
2) The macros were inflexible. Users had very few, if any, options that they could use.
3) The names used for macros followed no pattern, and the formats for labeling and operands specified were not standardized in any way.
4) The block structure could not be viewed at a glance. The labeling of loop termination macros was unnecessary and at the discretion of the user. This made visual checking of code very difficult where nesting or large blocks occurred.

PRESENT IMPLEMENTATION OF MACROS

The macros written for the Fall semester of 1975 were supplanted by a more comprehensive set of macros for the Spring semester of 1976. With the exception of the PROC macro, the old macros were quietly discarded with no regrets. The best thing that could be said about the old macros is that they provided the authors with experience.

The structured programming macros in the new implementation are designed around the concept of a block. For the purposes of this discussion, a block is a unit of code that starts with a macro, contains one or more instructions that perform some easily definable function, and ends with another macro. Blocks can be combined to form larger blocks or modules. Blocks can also be nested within other blocks. A program may be one or more blocks. This approach lends itself both to Top-Down and Bottom-Up programming.

In this section, a brief description of all the macros will be given. Of the macros described, the macros @WHLE, @UNTL, @AND, @OR and @ASRT have the same condition specification formats as those described for the @IF macro in a later section.

Labels are necessary on any macros that define a block of code. These labels are used to pass information to following macros within a block. All of the macros that define a block must start with the same label.[1] The only macro that actually generates that label is the first macro in the block. All other macros in that block generate a derivation of that label that starts with an "@" (if they generate a label at all).

Since the macros of a block are designated by a common label, nesting of blocks can be done. The user is responsible for closing blocks with the proper macro. If a block is not closed, an undefined label error generated by the assembler will prevent execution of the program.

SELECTIVE BLOCK EXECUTION

@IF...@END

The @IF macro defines the beginning of a block that is executed if the condition specified in the operand field of the macro is true. The block is terminated by the @END macro.

@SLCT...@CASE...@DFLT...@END

The @SLCT macro defines the beginning of a "select-one-of-many" block. The accumulator is tested for a value between zero and the number of cases defined. A branch is made to the corresponding case. Each case is defined by an @CASE macro. If the accumulator contains a number outside the range of case numbers, the code beginning with the @DFLT macro is executed. The @END macro always terminates the @SLCT block.

REPETITIVE BLOCK EXECUTION

@DO...@END

The @DO macro defines the beginning of a pre-test DO loop that is similar to the DO loop of PL/1. The loop index must be specified by the user. It is incremented and then tested. An exit from the loop occurs when the index exceeds the limiting bound. The @DO block is always terminated by the @END macro.

@WHLE...@END

The @WHLE macro defines a pre-test loop that is executed while the condition specified is true. The @WHLE block is always terminated by the @END macro.

@RPET...@UNTL

The @RPET macro defines the beginning of a block that is repeated until the condition specified on the @UNTL macro is true. The @UNTL macro always defines the termination of an @RPET loop.

EXTENDING CONDITION TESTS

@AND

The @AND macro defines another condition that must be true before an @IF block will be executed. The @IF block could then appear as: @IF@AND...@END. There is no limit on the number of @AND macros that may be associated with one @IF macro. This macro may also be used in an @WHLE block.

@OR

The @OR macro defines an alternate condition that may be true to allow the execution of an @IF block. The present implementation allows only one @OR to be associated with an @IF block. Using the @OR macro, an @IF block would look something like: @IF@OR...@END; or perhaps: @IF@AND@OR...@END. This macro may also be used in an @WHLE block.

@ELSE

The @ELSE macro defines an alternate path that is executed when the conditions that allow execution of the @IF block are falso. The @IF block could then look something like: @IF@AND@OR..@ELSE..@END.

SUBROUTINE BLOCK EXECUTION

@PROC...@RETN

The @PROC macro defines the beginning of a subroutine and provides standard linkage for all subroutines. The @PROC macro saves all registers on entry to the subroutine. A subroutine is always terminated by the @RETN macro. The @RETN macro can specify that the contents of certain registers be transmitted to the calling routine. If no such registers are specified, all registers are restored before control is returned to the caller.

@EXEC

The @EXEC macro is used to call internal subroutines. The @EXEC macro allows for the passing of an argument list to a subroutine.

@CALL

The @CALL macro is identical to the @EXEC macro, except that it is for the calling of external subroutines. An external subroutine is one that is assembled separately from the calling routine.

@ARG#

The @ARG# macro is used internal to a subroutine to get the count of arguments in the argument list. This count is placed in the accumulator.

@GETA

The @GETA macro is used internal to a subroutine to get the value of an argument that was passed to the subroutine. The value of the argument is placed in the accumulator.

@PUTA

The @PUTA macro is used internal to a subroutine to change the value of an argument.

SYSTEM MACROS

PROGM, SETUP

The PROGM macro must be the first statement encountered by the assembler in any user program. This macro generates only global SET symbols that are necessary for the proper assembly of all other macros. The SETUP macro must be the first statement executed in a user program. The SETUP macro initializes certain necessary values that are used during execution of user programs.

@DSAB, @ENAB

The @DSAB macro is used to disable interrupts so that critical sections of code can be executed

without interruption. The @ENAB macro enables interrupts so that normal processing can continue. These macros are slightly modified forms of the DSAB and ENAB macros presented in Lane [5].

@INT4, @ILSW

The @INT4 macro defines the beginning of the interrupt handler. This macro saves the contents of all registers on entry to the interrupt handler. The save area specified is identical to that described for the @SAVE macro in a later section. This macro is also necessary to the proper functioning of the @DSAB macro. The @ILSW macro is used by the interrupt handler to determine which device caused the interrupt and to branch to an appropriate routine.

@SAVE, @RSTO

The @SAVE macro saves the contents of all registers in a standardized save area. The @RSTO macro accesses a standard save area and restores the contents of the registers to the saved values. (These two macros are generalized versions of the SAVE and RSTO macros.[5])

@ASRT

The @ASRT macro is used to test the truth of a specified condition. If the assertion is true, execution continues normally. Otherwise, the macro causes execution to enter an infinite loop. In this way, the programmer can catch errors before they cause other errors.

@BIT#

The @BIT# macro returns the bit position of the first bit on (counting from left) in the accumulator. Since bit zero may be on, a value of 31 is returned if no bits are on. This macro was designed to be used with @SLCT.

DESCRIPTION OF @IF AND @DO

In order to give the reader an understanding of the specific macros being discussed, the formats of the @IF and the @DO macros will be briefly presented. The descriptions of macros include example formats for macro calls. In these formats, words that are capitalized are macro keywords with special meanings. Words that are in small type indicate that the user may vary the word that appears in that field.

Many instructions are built from SET symbols as macros are expanded. This allows readability of the example expansions. Readers will also find that the examples show single quote characters where "@" symbols should be. Also, equal signs appear instead of the "#" symbol. This occurs because the 1130 system substitutes certain characters for some other characters that do not appear on its print chain.

@IF...@END

The @IF macro defines the beginning of a block of code that is executed if the condition test specified in the operand field of the macro is true. The @IF macro has several simple formats. They are as follows:

```
lab  @IF  var1,cond,var2,L
lab  @IF  var1,cond,var2
lab  @IF  var,@ZERO
lab  @IF  x
```

The word "lab" denotes a three character user label. The word "cond" denotes a comparison that can be written in any of the following ways:

cond	meaning
@EQ	var1 equal to var2
@NE	var1 not equal to var2
@GT	var1 greater than var2
@GE	var1 greater than or equal to var2
@LT	var1 less than var2
@LE	var1 less than or equal to var2

Wherever the expression "var1" occurs in the formats above, the word @ACC may appear to indicate that the comparison is to be done using the IBM 1130 accumulator.

When the letter "L" occurs as the fourth parameter in the above format, the expression "var2" is used as a literal value rather than an address. Otherwise, the expressions "var1", "var2", and "var" are regarded as addresses of the data to be compared. The word "@ZERO", when used as shown above, indicates that the block is to be executed if the variable "var" is zero.

The letter "x" in the short format above represents a simple comparison against the condition of the accumulator. The codes that can be used and their meanings are as follows:

x	meaning
P	ACCUM positive
N	ACCUM negative
Z	ACCUM zero
NP	ACCUM not positive
NN	ACCUM not negative
NZ	ACCUM not zero
ODD	ACCUM odd
E	ACCUM even
O	Overflow bit on
C	Carry bit on

The @END macro must always close an @IF block.

@DO...@END

The @DO macro creates a repetitive loop that is much like the DO loop in PL/I. @DO sets up a loop with a pre-test, so that the specified condition is always tested before the loop is entered. The forms of the @DO macro are as follows:

```
lab  @DO  var,bnd1,bnd2
lab  @DO  var,bnd1,bnd2,L
lab  @DO  var,bnd1,bnd2,BY,incr
lab  @DO  var,bnd1,bnd2,L,BY,incr
```

The operand "bnd1" specifies the starting value of the loop. This is considered to be a literal

value, unless replaced by the word "@ACC". @ACC indicates that the starting value for the loop is in the accumulator.

The operand "bnd2" specifies the terminating value of the loop. This operand is considered to be the address of the bound unless the letter "L" appears as an operand. When "L" appears, "bnd2" is taken as a literal value.

The operand "var" is the address of the loop index. Since the index registers of the IBM 1130 are located at memory locations 1, 2, and 3, the index registers can also be used as the loop index in any @DO loop. This feature allows flexibility for indexing through tables of data.

The loop index is modified by +1, unless the word "BY" appears as an operand, in which case the increment "incr" specified by the user is used to modify the loop index. Increments may be positive or negative. The block started by the @DO macro must always be closed by the @END macro.

DESIGN CONSIDERATIONS OF THE MACROS

The early macros were useful in testing out ideas. The two biggest problems in writing good macros were 1) making the macros both useful and easy to use, and 2) the IBM 1130 Macro Assembler. The method of labeling each macro in a block with the same label was implemented only after much thought as to its aspects from a user viewpoint. Due to restrictions imposed by the Assembler[3], any method of passing information to following macros through labels is both necessary and at the mercy of users. Other methods that did not depend on users could not be implemented on the IBM 1130.[2] The method used in this implementation is both easily remembered due to standardization, and easily desk-checked. It has the added attraction that assembly errors occur when certain critical macros like @END are omitted. This prevents user programs with an invalid block structure from executing.

The macro names have been thoroughly considered for readability and standardization where possible. Since the 1130 Macro Assembler allows only five-character names, explanatory names used by some authors [1,2,4] are immediately ruled out. The names of macros used to end blocks are of particular interest in themselves. The method of reversing the spelling of the loop-starting macro to produce the terminating macro has always been unacceptable to the authors. DNA makes a fine molecule but a poor macro name. And, RO is one way to make a boat move (and FI on you if you think otherwise). The combination of matching labels and standardized end statements [1] has proved to be reasonably readable and easy to use.

The looping and selective block execution macros now generate pre-test loops (except for @RPET, of course). By doing so, more overhead is generated. However, the resulting code is easy to follow and to use. There is no possibility that a programmer will terminate a loop with the wrong macro, and actually get into execution. In addition, the macros now generate error messages and informatory messages where applicable. This facilitates debugging.

THE IMPACT OF STRUCTURED PROGRAMMING CONCEPTS

Over the seven semesters since the Fall of 1971 that the Systems Programming course has been taught, there have been a great number of errors by students which keep occurring again and again. The errors generally deal with "clobbering" registers, save areas, control blocks, instructions, and pointers with incorrect data. In most cases the overall design of each student project is good. However, the tedious process of maintaining bits and bytes at the assembler language level and the inability of the IBM 1130 assembler to detect such things as misplaced operands (which result in address fields of zero and no errors) create errors which have a negative impact on the learning experience which the course has as its basic goal. This goal is to learn the principles of operating system design, not how to write IBM 1130 assembler language programs. The IBM 1130 and its assembler language are merely tools to implement a system that the students design themselves. (It should be noted that the IBM 1130 is the only computer system at the University available to computer scientists for using the hands-on approach to teaching systems programming [5].)

The addition of macros for structured programming yields a better and more powerful tool for the students to accomplish the above-mentioned goal. The structuring of the students' assembler language programs exposes students to improved programming techniques for the design of operating systems. It also eliminates many of the bit and byte errors in simple tasks such as programming loops within their programs. In this case, a single @DO...@END pair can accomplish what might require ten or so instructions without the macros. Thus, if one assumes that the number of errors made by the students is proportional to the number of lines of code written, fewer errors should be made, and hence, the MPX system should become operational earlier in the semester. Finally, the structuring techniques which the macros introduce should further reduce errors made by students.

Figures 3 and 4 illustrate the use of several of the macros in the implementation of the MPX projects. It should be pointed out that the examples are exactly as they appeared in the students' projects. (The appearance of AGO assembler instructions after several of the macros are the results of meaningless warning messages printed by a rather archaic assembler.)

One MPX project which was fully implemented in May 1976 used a very modular approach. However, most all macros (used earlier by the students) were eliminated from the MPX because of the extreme slowness of the IBM 1130 Macro Assembler in processing macros. The students implemented this MPX using external subroutines to structure the system; a data communication module was implemented as a subroutine to provide access to common data areas. Such sections as the first and second level interrupt handlers for all devices, the command

```
                         00137     CNT    'IF       FUNCT,'EQ,CONTR DO IF FUNCTION CONTRL
                         00147 Q +CONTR  AGO
0075 01 0C0C0188 00171                   XIO    L   SKIP   INITIATE THE CONTROL OPERATION
0077 01 C4000174 00172                   LD     L   ZERO   PUT CORRECT RETURN CODEIN ACC
0079 01 4C0C001E 00173                   B      L   RETRN  RETURN TO USER
                         00174     CNT    'END
                         00187     IWR    'IF       FUNCT,'EQ,INTWR DO IF FUNCTION INITWR
                         00197 Q +INTWR  AGO
008D 01 C4000172 00221                   LD     L   ACSAV  GET ADDRESS OF USERS DATA AREA
008F 01 84000176 00222                   A      L   TWO    GET ADDRESS OF USER PRINT LINE
0091 01 D40C021C 00223                   STO    L   CHAR   STORE THIS ADDRESS IN CHAR
0093 20 292570D6 00224                   LIBF       ZIPCO
0094 0  1100     00225                   DC         /1100
0095 1  021C     00226                   DC         CHAR
0096 1  0134     00227                   DC         DATA+2
0097 0  005C     00228                   DC         80
0098 30 084C78F3 00229                   CALL       HLPT3
009A 01 C4800172 00230                   LD     I   ACSAV  GET COUNT OF CHARS TO PRINT
009C 01 D400017E 00231                   STO    L   NNNNN  PUT THAT NUMBER IN NNNNN
009E 01 940C017C 00232                   S      L   CMAX   CHECK IF GREATER THAN 120
00A0 01 4C300122 00233                   BP         ILLCT  IF YES BRANCH TO ILLCT
00A2 01 C4800172 00234                   LD     I   ACSAV  GET COUNT OF CHARS TO PRINT
00A4 01 4C280122 00235                   BN         ILLCT  IF ) 0 BRANCH TO ILLCT
00A6 01 C40C0172 00236                   LD     L   ACSAV  GET FIRST WORD OF USER IOCC
00A8 01 8C000175 00237                   AD     L   ONE    ADD ONE TO IT
00AA 01 D4000170 00238                   STO    L   BITTS  STORE IT IN BITTS
00AC 01 C480017D 00239                   LD     I   BITTS  LOAD ACC WITH 2ND WORD OF USER
                         00240     * DATA AREA *
00AE 0  1804     00241                   SRA        4      ISOLATE CARRIAGE SKIP MASK
00AF 01 D4000181 00242                   STO    L   CHANM  PUT CHANNEL MASK IN CHANM
00B1 01 C4000172 00243                   LD     L   ACSAV  GET FIRST WORD OF USER IOCC
00B3 01 8C000175 00244                   AD     L   ONE    ADD ONE TO IT
00B5 01 D4000170 00245                   STO    L   BITTS  STORE IT IN BITTS
00B7 01 C480017D 00246                   LD     I   BITTS  LOAD ACC WITH 2ND WORD OF USER
                         00247     * DATA AREA *
00B9 01 E4000178 00248                   AND    L   FIFTN  ISOLATE SKIP BITS
00BB 01 D4000180 00249                   STO    L   SKIPB  PUT SKIP BITS IN SKIPB
00BD 01 C40C017E 00250                   LD     L   NNNNN  CHECK COUNT OF CHARS TO PRINT
00BF 01 4C180114 00251                   BZ         CONTL  IF ZERO BRANCH TO CONTL
00C1 01 C4000181 00252                   LD     L   CHANM  GET CHANNEL MASK IN ACC
00C3 01 4C1800D5 00253                   BZ         PRI    IF CHANMASK IS ZERO BRANCH PRI
00C5 00 650C0010 00254                   LDX    L1  /10    PUT A 16 IN REG ONE
00C7 0  1140     00255                   SLCA   1          CHECK IF CHANNEL MASK LEGAL
00C8 0  1001     00256                   SLA        1
00C9 01 4C200128 00257                   BNZ    L   ILMSK  BRANCH TO ILMSK IF NOT
00CB 01 C4000181 00258                   LD     L   CHANM  GET CHANNEL MASK IN ACC
00CD 01 D480018A 00259                   STO    I   RITE   PUT IT IN CHANMASK AREA OF IOCC
00CF 01 0C00018A 00260                   XIO    L   RITE   INITIATE WRITE COMMAND
00D1 01 C4000174 00261                   LD     L   ZERO   PUT CORRECT RETURN CODE IN ACC
00D3 01 4C0C001E 00262                   B      L   RETRN  RETURN TO USER
00D5 01 C4000172 00263     PRI           LD     L   ACSAV  GET ADDR OF USERS DATA AREA
00D7 01 84000176 00264                   A      L   TWO    GET ADDR OF USERS PRINT LINE
00D9 01 D4000183 00265                   STO    L   FPTR   STORE THAT ADDR IN FPTR
00DB 01 C400017E 00266                   LD     L   NNNNN  GET COUBT OF CHARS TO BE MOVED
00DD 01 8C000175 00267                   AD     L   ONE    ADJUST CHAR COUNT
00DF 0  1801     00268                   SRA        1      DIVIDE COUNT BY 2 TO GET WORD C
00E0 01 D4000170 00269                   STO    L   MMMMM  STORE WORD COUNT IN MMMMM
                         00270     BLN   'DO       INDEX,1,60,L DO THIS CODE 60 TIMES
                         00302 Q +       AGO
00FC 01 C400012E 00321                   LD     L   BLANK  LOAD 1403 PRINTER CODE FOR BLAN
00FE 01 D480012F 00322                   STO    I   TBUF   BLANK OUT MY BUFER INDIRECTLY
0100 01 7401012F 00323                   MDM    L   TBUF,1 INCREASE TBUF BY ONE
                         00324     BLN   'END
0104 01 C4000130 00338                   LD     L   JBUF   LOAD THE ADDRESS OF MY BUFER
0106 01 D400012F 00339                   STO    L   TBUF   STORE THIS ADDRESS IN TBUF
                         00340           'CALL     'MOVE,3,FPTR,TOPTR,MMMMM MOVE USER
                         00359     * BUFFER TO OURS *
010E 01 0C0C0190 00360                   XIO    L   COPY   INITIATE INITIATE WRITE COMMAND
0110 01 C4000174 00361                   LD     L   ZERO   PUT CORRECT RETURN CODE IN ACC
0112 01 4C0C001E 00362                   B      L   RETRN  RETURN TO USER
0114 01 C4000180 00363     CONTL         LD     L   SKIPB  GET SKIP COUNT IN ACC
0116 01 0C0C0188 00364                   XIO    L   SKIP   INITIATE SKIPPING OPERATION
0118 01 C4000180 00365                   LD     L   SKIPB  GET SKIP COUNT IN ACC
011A 01 94000175 00366                   S      L   ONE    DECREMENT SKIP COUNT BY ONE
011C 01 D4000180 00367                   STO    L   SKIPB  PUT REDUCED SKIP BITS IN SKIPB
011E 01 C4000174 00368                   LD     L   ZERO   PUT CORRECT RETURN CODE IN ACC
0120 01 4C0C001E 00369                   B      L   RETRN  RETURN TO USER
0122 01 C4000177 00370     ILLCT         LD     L   THREE  PUT CORRECT RETURN CODE IN ACC
0124 01 4C0C001E 00371                   B      L   RETRN  RETURN TO USER
                         00372     IWR   'END
0126 01 C4000176 00385                   LD     L   TWO    LOAD ACC WITH ILLEGAL FUNCT CODE
0128 01 4C0C001E 00386                   B      L   RETRN  RETURN TO USER
```

FIGURE 3. AN EXAMPLE OF THE USE OF THE @IF AND @DO IN STUDENTS' MPX PROJECTS

```
021D 01 C4000580 00589     KB     LD   L  RW120 PUT 120 IN ACC
021F 01 D4000542 00590            STO  L  RAREA STORE IN IOCC
0221 0  1010     00591            SLA     16    ZERO THE ACC
0222 01 D400053D 00592            STO  L  KBCPF RESET FLAG
0224 01 C4000173 00593            LD   L  ACSAV+1
0226 01 E4000531 00594            AND  L  AREAD IS IT A READ FUNCTION
0228 01 4C180232 00595            BZ      KEYBD IF YES, GO TO KEYBOARD ROUTINE
022A 01 E400C532 00596            AND  L  AWRIT IS IT A WRITE FUNCTION
022C 01 4C180288 00597            BZ      CONSL IF YES, GO TO CONSOLE ROUTINE
022E 01 C4000176 00598            LD   L  TWO   RETURN CODE=2 IF BAD FUNCTION
0230 01 4CCC053B 00599            B    L  ERRSB BRANCH TO ERROR SUBROUTINE
                 00600     ************************************************
                 00601     *     WHEN A READ REQUEST COMES IN, THE CHAR     *
                 00602     *COUNT AND BUFFER ADDRESS ARE SAVED.  KBFLG IS SET*
                 00603     *TO ONE(BEGIN READ FLAG),AND A CARRIAGE RETURN IS  *
                 00604     *ISSUED.                                          *
                 00605     ************************************************
0232 01 C40C0172 00606     KEYBD  LD   L  ACSAV GET FIRST WORD OF IOCC
0234 01 D40C0537 00607            STO  L  CCADR SAVE ADDR OF WHERE COUNT IS
0236 01 840C0175 00608            A    L  ONE
0238 01 D40C0583 00609            STO  L  UADDR STORE BUFFER ADDRESS
023A 0  1010     00610            SLA     16
023B 01 D4000540 00611            STO  L  POINT RESET THE POINTER
023D 01 C4800172 00612            LD   I  ACSAV LOAD ACC WITH CHAR COUNT
023F 01 D40C0538 00613            STO  L  CHARC STORE CHARACTER COUNT
                 00614     RRR    'IF      CHARC,'GT,120,L
                 00646     RRR    'OR      CHARC,'LT,0,L
0266 01 C40C0177 00679            LD   L  THREE RETURN CODE=3 IF BAD COUNT
0268 01 4CCC053B 00680            B    L  ERRSB BRANCH TO ERROR SUBROUTINE
                 00681     RRR    'END
                 00695     RRA    'IF      CHARC,'EQ,0,L
027C 01 74010530 00729            MDM  L  KBCPF,1 OPERATION COMPLETE FLAG ON
027E 01 0C0C0590 00730            XIO  L  CRGRT ISSUE A CARRIAGE RETURN
0280 0  101C     00731            SLA     16    RETURN CODE=0 IN ACC
0281 01 4C00001E 00732            B    L  RETRN RETURN FROM IOCS
                 00733     RRA    'END
0283 01 0C0C0590 00746            XIO  L  CRGRT DO A CARRIAGE RETURN
0285 0  101C     00747            SLA     16
0286 01 4C0C001E 00748            B    L  RETRN RETURN FROM IOCS
                 00749     ************************************************
                 00750     *     WHEN A WRITE REQUEST COMES IN, THE CONSOLE  *
                 00751     *IS CHECKED TO SEE IF BUSY AND IF READY.  THE CHAR*
                 00752     *COUNT AND DATA ADD OF THE MESSAGE ARE SAVED, CHAR*
                 00753     *COUNT CHECKED, CPRFG SET TO ONE(BEGIN WRITE FLAG),*
                 00754     *AND ISSUES A CARRIAGE RETURN.                    *
                 00755     ************************************************
0288 01 C48C0172 00756     CONSL  LD   I  ACSAV
028A 01 D4000538 00757            STO  L  CHARC STORE HERE
028C 0  101C     00758            SLA     16
028D 01 D4000540 00759            STO  L  POINT RESET THE POINTER
                 00760     RRZ    'IF      CHARC,'GT,120,L
                 00792     RRZ    'OR      CHARC,'LT,0,L
02B4 01 C40C0177 00825            LD   L  THREE RETURN CODE=3 IF ILLEGAL COUNT
02B6 01 4CCC053B 00826            B    L  ERRSB
                 00827     RRZ    'END
                 00841     RRX    'IF      CHARC,'EQ,0,L
02CA 01 74010530 00875            MDM  L  KBCPF,1 OPERATION COMP FLAG ON
02CC 01 0C0C0590 00876            XIO  L  CRGRT ISSUE A CARRIAGE RETURN
02CE 0  101C     00877            SLA     16    RETURN CODE=0 IN ACC
02CF 01 4C00001E 00878            B    L  RETRN RETURN FROM IOCS
                 00879     RRX    'END
02D1 01 C4C00172 00892            LD   L  ACSAV GET ADDRESS OF CHAR COUNT
02D3 01 840C0175 00893            A    L  ONE   GET ADDRESS OF DATA AREA
02D5 01 D4C00533 00894            STO  L  BFADR STORE ADDRESS IN BFADR
02D7 01 0C0C0590 00895            XIO  L  CRGRT DO A CARRIAGE RETURN
02D9 0  1010     00896            SLA     16    RETURN CODE=0
02DA 01 4C0C001E 00897            B    L  RETRN RETURN FROM IOCS
                 00898     ************************************************
                 00899     ************************************************
                 00900     *ILS04 SAVES THE ADDRESS OF THE NEXT INSTRUCTION  *
                 00901     *TO BE EXECUTED WHEN THE INTERRUPT OCCURED .      *
                 00902     *IT ALSO SAVES THE REGISTERS AND STATUS.          *
                 00903     ************************************************
                 00904     ************************************************
                 00905     ILS04  'INT4    PTSAV MACRO FOR ILS04
                 00933            'ILSW    INTPR,INTRD,INTCP SENSE AND SAVE ILSW
0305 01 C4000530 00967     RESTO  LD   L  KBCPF CHECK THE KB/CP FLAG
0307 01 4C18030F 00968            BZ      CHKPT CHECK THE PR FLAG
0309 0  1010     00969            SLA     16
030A 01 D40C053D 00970            STO  L  KBCPF RESET KB/CP FLAG
030C 00 44000202 00971            BSI  L  QIT   BRANCH TO QIT IF OPERATION COMP
030E 0  7016     00972            B         JUSRS SKIP NEXT SECTION
```

FIGURE 4. THE USE OF @IF, @OR, @INT4, AND @ILSW IN A STUDENT MPX PROJECT

interpreter, etc., were all implemented as external subroutines. This is not an unusual approach if one has a reasonably good assembler (with such facilites as GLOBAL symbols), but using the IBM 1130 system, the students had to carefully select techniques for using external subroutines which would work under the constraints of the assembler. The point here is that the structuring of this project greatly decreased debugging time. The fact that the project was a class A implementation (see Figure 1) attests to the fact that this particular structured approach was successful.

Students who completed the course in May 1976 were instructed in class on the use of the macros described in this paper. The use of these macros was highly recommended to the students and all students in that class used the macros in some or all of their assignments. The results of the recent introduction of the macros is perhaps best conveyed to the reader by comments of the students:

> "The macros were very helpful in the implementation of our MPX project. By using the macros we saved time which normally would be spent keypunching, coding, and debugging. Fewer wild branches were taken since the use of the @IF macro needs no branching."

> "The macros were extremely good for the modular approach to programming, their best points being looping control, case testing, conditional testing, and the ability to change core memory compared to the previous case of only being able to work off the accumulator. Conditional testing proved to be quite powerful, especially in the use of IF-THEN-ELSE tests and the IF-AND-OR-THEN tests.

> "We initially used the macros extensively in our first project (IOCS). They proved to provide quick, easy, error-free coding. But, because of the slow disk on the 1130, the assembly times for IOCS were unrealistic."

> "In writing the MPX, we have found that the macros can be a very effective and time saving approach. Several of the macros provided a means of writing code without getting bogged down with the details of every step of the program."

> "Use of macros made program implementation easier because of the following:
> 1. Programs were easier to write since a macro is a form of a higher level instruction. Many operations could be done with only one statement.
> 2. Programs were easier to organize and debug because blocks of code are naturally divided and isolated by the macro and its @END statement.
> 3. Problems were easier to isolate because the macros could be assumed correct, pointing to programmed code as the source of trouble.
> 4. Fewer statements had to be written and therefore fewer cards had to be keypunched, eliminating another source of error.

> "Given a faster machine which was designed to incorporate macros efficiently, a programmer would be foolish not to use them to his advantage. However, their use on the IBM 1130 is not attractive because of the great amount of time required to resolve a program incorporating macros due to the slow disk."

The conclusions given by the students were unanimous: 1) the macros are a valuable tool and proved to be very useful in eliminating programmer errors; 2) as projects got larger, the extreme slowness of macro resolution on the IBM 1130 forced many students to abandon the use of many of the macros. Hence, the machine being used for the course does not have sufficient power to make the use of macros attractive for larger projects.

It is hoped that a new system, more suitable to the structured approach described in this paper, can soon be acquired for use in the systems programming course. In the meantime, methods for improving the IBM 1130 macro assembler are being investigated.

SUMMARY

The hands-on approach to teaching systems programming has been successful in the past and is improving every semester. More stringent standards for specifications, flowcharts, and documentation have greatly improved the course. The use of structured programming concepts (e.g., via the use of macros) in writing student MPX projects will likely produce even more dramatic improvements when a more suitable computer system is acquired for use in the course.

Five years' experience in the hands-on approach to teaching systems programming has proved that students who finish the course discussed in this paper possess a thorough knowledge of the basic principles of operating system design. (Many have been successful as systems programmers in industry without further experience in systems programming.)

The hands-on approach indeed gives students enthusiasm and motivation which produce remarkable results. The addition of structured programming techniques to the students' hands-on experience will make them far more accomplished in the design of operating systems.

ACKNOWLEDGEMENT

The macro block labeling method described in this paper and other ideas were contributed by James F. Williams of the West Virginia University Computer Center.

REFERENCES

1. Harris Corporation (Data Communications Division), Remote Communications Processor Assembler Language User's Manual, Dallas, 1975.

2. Herman-Giddens, et al., "An Approach to Structured Programming for Users of Small Computers", ACM Southeastern Regional Meeting, April 14-16, 1975.

3. International Business Machines Corporation, IBM 1130/1800 Assembler Language, Boca Raton, Florida, 1971.

4. Kimura, Takayuki, "Structured Programming in PDP-11 Assembly Language", Unpublished Manuscript, University of Delaware, Newark, Delaware, 1974.

5. Lane, Malcolm G., "A Hands-On Approach to Teaching Systems Programming, SIGCSE Bulletin, Volume 7, Number 1, February, 1975.