Principles of Programming

Section 4: Symbolic Programming

IBM® Personal Study Program

Section 4: Symbolic Programming

The simple examples of computer instructions that we have seen so far have used actual machine addresses, which is the way the machine must have them. However, very few programs are actually written this way. Writing programs with actual (also called *absolute*) addresses leads to problems in assigning data to storage locations, makes it difficult to write cross references within a program, leads to difficulties when several people must work on the same job, and leads to programs which are very difficult to correct and to modify.

For these reasons most programming is done with a *symbolic programming system.* For the 1401 system there are three rather similar symbolic programming systems available, called SPS-1, SPS-2, and Autocoder. In this section we shall discuss the features of the use of SPS-1; all of this material will also be applicable to SPS-2 and to Autocoder, since these systems are extensions of SPS-1.

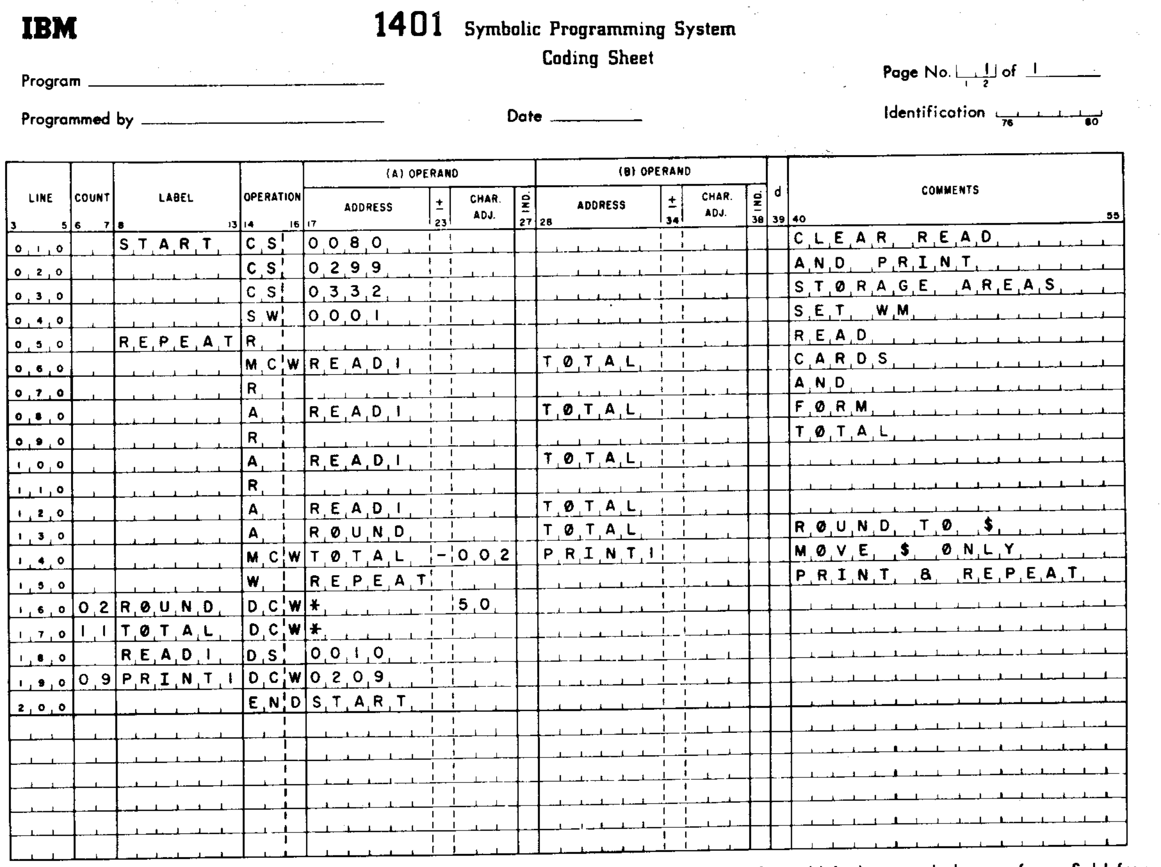
After establishing the fundamental ideas of symbolic programming in this section, almost all later examples will be written in the SPS language. This will allow the reader ample time to become thoroughly familiar with symbolic programming, bearing in mind that almost no absolute programming is done in applications.

4.1 Fundamentals of Symbolic Programming

The basic idea of symbolic programming is that *symbols* are written in place of actual machine addresses. After the entire program has been written in the symbolic language, the symbols are translated into absolute addresses by a processor. The processor is itself a large program, which can be run on the same machine as the eventual absolute language program. The program as initially written in symbolic language is called a *source program;* the processor program translates the source program into an *object program.* The object program may then be run to produce problem results.

It is worth emphasizing before proceeding further that (1) the processor is itself a program, not a machine, and (2) the processor *only* translates the source program into an object program—it does not cause the object program to be executed.

We may begin to get a clearer idea of how symbolic programming is used by considering an example. Figure 1 is a program written on a



IBM **1401** Symbolic Programming System
Coding Sheet

Program ______________ Programmed by ______________ Date ________ Page No. 1 of 1 Identification ________

LINE 3–5	COUNT 6–7	LABEL 8–13	OPERATION 14–16	(A) OPERAND ADDRESS 17	± 23	CHAR. ADJ.	IND. 27	(B) OPERAND ADDRESS 28	± 34	CHAR. ADJ.	IND. 38	d 39	COMMENTS 40–55
010		START	CS	0080									CLEAR READ
020			CS	0299									AND PRINT
030			CS	0332									STØRAGE AREAS
040			SW	0001									SET WM
050		REPEAT	R										READ
060			MCW	READ1				TØTAL					CARDS
070			R										AND
080			A	READ1				TØTAL					FØRM
090			R										TØTAL
100			A	READ1				TØTAL					
110			R										
120			A	READ1				TØTAL					
130			A	RØUND				TØTAL					RØUND TØ $
140			MCW	TØTAL	-	002		PRINT1					MØVE $ ØNLY
150			W	REPEAT									PRINT & REPEAT
160	02	RØUND	DCW	*		50							
170	11	TØTAL	DCW	*									
180		READ1	DS	0010									
190	09	PRINT1	DCW	0209									
200			END	START									

Figure 1. Example of a Symbolic Programming System (SPS) program. Four cards are read, after which the rounded sum of one field from each card is printed.

Symbolic Programming System coding sheet. The purpose of this very simple illustrative program is to read four cards, each of which contains an amount in dollars and cents in columns 1-10. The program is to form the sum of these four amounts, round the sum to the nearest dollar, and print the total in dollars on the printer in print positions 1-9. This of course is vastly simpler than anything we would normally do with a computer, but it will serve to illustrate the symbolic programming principles that are our concern at the moment.

A glance at Figure 1 shows that all addresses are written as symbols, with the exception of a few at the beginning. The symbols used in this program happen to be either five or six characters. In general a symbol may be from one to six letters and digits; the first character must be a letter. The invention of symbols is completely under the control of the programmer. It is often convenient to choose symbols that are descriptive of the information referenced by them, such as using TOTAL to stand for the address of the field in storage where a total is stored. On the other hand, symbols are not *required* to have any such meaning, and none is attached to them by the processor.

We see that it is possible to use absolute addresses where convenient. The processor is easily able to distinguish between symbolic and absolute addresses by the fact that the first character of a symbol is always a letter, whereas the first character of an absolute address is always a digit. We note that absolute addresses are written in *four*-digit form. This is also true of addresses over 999. Using SPS we are not required to figure out the three character form of addresses. If we want to write the address 1234, we write it just that way rather than as S34. The processor will convert the four-digit addresses to the three-character form required inside the machine (and if an address like S34 were used, it would be misinterpreted as a symbol).

In looking at the program in Figure 1 it will be noted that the operation codes are written in a new way. These are *mnemonic operation codes.* "Mnemonic" means "aiding the memory." These substitute operation codes are used because they are easier to remember than the actual machine operation codes. CS is the mnemonic operation code for Clear Storage; this is indeed easier to remember than /, and SW is easier to remember than a comma. The mnemonic operation codes for the instructions that have been discussed so far are shown in Figure 2.

Instruction	Actual	Mnemonic
Move Characters to A or B Word Mark	M	MCW
Set Word Mark	,	SW
Clear Word Mark	⌑	CW
Read a Card	1	R
Punch a Card	4	P
Write a Line	2	W
Clear Storage	/	CS
Add	A	A
Subtract	S	S

Figure 2. Mnemonic operation codes for the instructions so far considered.

It is still permissible to use the actual operation codes. If this is done, the code should be written in column 16, whereas mnemonic operation codes are always written starting in column 14.

We may now investigate the program shown in Figure 1 in detail. We see that the first instruction has a label of START. This label becomes the symbolic address of the instruction. That is to say, when this source program is translated by the processor, the symbol START will always be associated with the location in storage to which the processor assigns the operation code of the Clear Storage instruction. Any instructions elsewhere in the program that must refer to this instruction may be written with the symbolic address START instead of an absolute address.

The addresses of the three Clear Storage instructions are absolute. This is done because these addresses could never change; no program modification or correction could ever involve changing the read and print storage areas. The address of the Set Word Mark instruction is also absolute, on the theory that the field to be read from the card will always start in column 1. We shall discuss later the consequences of this assumption.

The Read a Card instruction presents no new concepts. The Move Characters to A Word Mark moves the data field from its position in the read storage area to the locations where the total will be accumulated. The following six instructions read the other three cards and add their data fields to the locations where the total is accumulated. The next instruction adds a 50 to the total. Remembering that the data fields were assumed to represent dollars-and-cents amounts, 50 added to the least significant part of the total is, in effect, $0.50. This means that if the cents amount is 49 or less, adding 50 will not change the dollar amount. However, if the cents amount is 50 or over, adding 50 to it will increase the dollar amount by 1. This is exactly what we want in order to round the total to the nearest dollar.

The next instruction moves the dollars portion of the total to a section of the print storage area. *Character adjustment* is used on this instruction. When this instruction is processed, 2 will be subtracted from the address which is established as the equivalent of the symbol TOTAL. This approach is necessary because we do not know what the equivalent address will be—since the processor has not yet defined it. If we *did* know the actual address corresponding to TOTAL, we could write an address 2 less than that to get only the dollars portion. The effect of the character adjustment is just what we need: the eventual address will be 2 less than *whatever* address becomes the equivalent of TOTAL.

The last instruction writes the contents of the print storage area on the printer. We see here a variation of the Write a Line instruction: an address is given in the A-operand field. We recall that the Write instruction always refers to the print storage area, so that no address is required for the data. This is our first example of an instruction address which refers to another instruction, this being the Write and Branch instruction. When the line has been written, the control section of the machine will automatically take the next instruction from the location specified by the address in this Write instruction. This is why the first address of an instruction is referred to as the A/I address: it can refer either to a data address or to an instruction address. The idea here is that after the first group of four cards has been read and totaled, we would like to return to the beginning of the program to read another group. This process would be repeated indefinitely as long as cards remained in the hopper. (We shall consider in the next section how a test might be set up to detect the last card of the deck.)

The next four instructions are used to define symbols in the program and in one case to define a constant that is referenced by a symbol. They are not instructions to the computer, but to the processor; they will not result in the creation of any instructions to be executed in the object program.

DCW stands for Define Constant with a Word Mark. Taking the first of these, we have an instruction to the processor to set up a constant two characters long, as specified by the number in the *count* field, columns 6-7. The constant is shown, starting in column 24, to be 50. The asterisk in column 17 says to the processor that the constant may be assigned to any convenient locations in storage. As we shall see later, the constant in fact would be assigned to the two locations immediately following the last instruction of the program. The symbol ROUND will be associated with the low-order character position of this two-character field. The DCW instruction that defines the symbol TOTAL is slightly different. It is specified as eleven characters, which is the number needed to hold the sum of four ten-digit numbers. However, nothing is written starting in column 24. This, in effect, defines

```
PG LIN  CT  LABEL    OP    A OPERAND          B OPERAND        D     LOC   INSTRUCTION COMMENTS

1 010   4   START    CS    0080                                      0333  / 080        CLEAR READ
1 020   4            CS    0299                                      0337  / 299        AND PRINT
1 030   4            CS    0332                                      0341  / 332        STORAGE AREAS
1 040   4            SW    0001                                      0345  , 001        SET WM
1 050   1   REPEAT   R                                               0349  1            READ
1 060   7            MCW   READ1              TOTAL                  0350  M 010 411    CARDS
1 070   1            R                                               0357  1            AND
1 080   7            A     READ1              TOTAL                  0358  A 010 411    FORM
1 090   1            R                                               0365  1            TOTAL
1 100   7            A     READ1              TOTAL                  0366  A 010 411
1 110   1            R                                               0373  1
1 120   7            A     READ1              TOTAL                  0374  A 010 411
1 130   7            A     ROUND              TOTAL                  0381  A 400 411    ROUND TO $
1 140   7            MCW   TOTAL -002         PRINT1                 0388  M 409 209    MOVE $ ONLY
1 150   4            W     REPEAT                                    0395  2 349        PRINT & REPEAT
1 160   2   ROUND    DCW   *                                   50    0400
1 170  11   TOTAL    DCW   *                                         0411
1 180       READ1    DS    0010                                      0010
1 190   9   PRINT1   DCW   0209                                      0209
1 200                END   START                                           / 333 080
```

Figure 3. Assembly listing of the program of Figure 1.

the constant to consist of eleven blanks. The situation here is that we need to specify the length of this field and to have the symbol TOTAL established as being equivalent to the low-order character of the field, but we do not actually need to enter a constant there. Here we are only setting up a storage area with a word mark and defining the meaning of the symbol associated with it.

The next instruction is a DS, for Define Symbol. It establishes 0010 as the absolute equivalent of the symbol READ1, but *without* causing anything to be loaded into storage with the object program. This is necessary here because we are dealing with the read area, which is used during object program loading; it is not permissible to use a DCW to set a word mark in this area. The DS, combined with the Set Word Mark instruction in the object program, accomplishes the same result, but does not set the word mark until *after* the object program is loaded. The DCW defining the symbol PRINT1 is acceptable, since the print area is not used during loading.

The last "instruction" is again strictly an instruction only to the processor. The END specifies to the processor that the end of the program has been reached and that the processor may proceed to complete the production of the object program. We write in the A-operand address portion of the END instruction the address of the first instruction that should be executed when the object program is later executed.

The way this program has been written, the processor would put the first character of the program in storage location 333. All succeeding characters would be stored in sequential locations, in this example.

The translation of the source program into an object program, which is also called *assembly*, may be outlined as follows. The source program cards are punched from the coding exactly as shown in Figure 1, with one card per line. The processor program must be in storage and will have complete control of the computer during the assembly; the source program is not executed during assembly. The processor reads the source program cards and translates the program into absolute form. The procedure varies somewhat depending on whether or not the machine on which the assembly is done has tapes. On a card machine there is an additional card-handling step during the assembly. In either case the result of the assembly by the processor is a deck of cards containing the object program. It is also possible to get a *post listing* or *assembly listing*, which shows both the original source program and the final absolute object program produced from it.

The assembly listing for the program in Figure 1 is shown in Figure 3. Note that the listing shows the instructions and data as originally written in the source program, and also the assembled object program input. The count field is seen to contain a value for all lines, including instructions; the latter is provided by the processor for the programmer's convenience. Notice that the addresses shown for the assembled input are correct for both instructions and constants: high-order for instructions and low-order for constants.

The program has not been executed yet! All that has been accomplished so far is to translate the symbolic source program into an absolute object program and to produce a deck of cards containing the object program. Now the object program may be loaded into the machine and run. It is only at this point that the cards containing problem data are placed in the hopper and read. In short, it is only now that the program that we have written is in control of the computer system.

Let us now consider what would be involved in making a change in this program. Suppose that after the program has been written and assembled, the problem specifications change so that it is necessary to form the sum of the dollars-and-cents amounts on *five* cards and that the fields are in columns 14-23 instead of 1-10.

To incorporate these changes in the program requires adding a Read a Card and an Add instruction, and changing all of the addresses that refer to the read storage area. If the program had been written in absolute, it would mean inserting the two instructions at some appropriate place, such as just before the rounding, and changing a number of absolute addresses. The insertion of the two instructions would require moving all instructions following the insertion, and the changing of the addresses would require rewriting all those instructions and repunching the instruction cards. Even in such an elementary program as this, we see that a small change can result in program changes requiring nearly as much work as the initial programming.

To change the symbolic program we start with the source program deck. Since almost nothing in the source program commits us to specific locations in storage, changes in the source program are much easier. The two instructions can be punched on cards and inserted at the proper place in the source program. At this point we may note a feature of line numbers that are preprinted on the form: they all end in zero. This means that up to nine instructions can be inserted between any two original instructions without destroying the sequence of line numbers. For instance, if the Read a Card and the Add instructions were to be inserted between lines 120 and 130, they could be given the line numbers 121 and 122 without in any way disturbing the line number sequence. This is valuable because by using a page number (at the upper right of the form) and a line number, the sequence of the source program cards can be defined as a protection against mistakes in handling of the source program deck. While on the general subject we may note also that the program identification can be punched in column 76-80 to provide an identification of the program deck, further reducing the possibility of mixups.

With the program written in symbolic form, the change in the location of the card field is almost completely solved by changing the address of the DS instruction that defines the field. On line 180 it is

necessary only to change the 0010 to 0023, which will change the absolute equivalent of the symbol when the program is reassembled. However, remember that a word mark was set in the high-order position of this field, and that an absolute address was used. If this address is not changed, the field will be incorrectly defined. This could be handled by changing the address of the Set Word Mark instruction to 0014, but a better procedure would have been to write the address in symbolic, with character adjustment, so that the change in field position will not create this particular problem.

Now when the program is reassembled, which is a simple matter, all of the addresses in the program that are written as READ1 will be changed. As a matter of fact, we note further that the insertion of the two new instructions changes the storage assignments of the ROUND and TOTAL fields, so that the reassembly changes virtually every address in the program. This is of no concern to us, since the processor takes care of the whole matter in a few minutes. The new assembly listing is shown in Figure 4.

It may seem a little strange to put so much emphasis on designing programs so that modifications are easy to make. It might be thought that once the program is written it can be forgotten. The actual fact is, however, that virtually all programs change constantly in use, either because improvements in the program are possible or because the problem specifications themselves change. It is not unusual for one programmer to be assigned the exclusive responsibility of making program changes. The wise procedures designer and programmer give considerable thought to ease of modification *before* the programming is done. Symbolic programming, properly used, is of great value in providing this simplicity of modification.

Review Questions

1. *Which of the following are allowable SPS symbols? CAT, K, F67YN, 674N, ABCDEF, H&89, GROSSPAY, NET PAY, NETPAY.*
2. *Explain the relation between the source program and the object program. When is the object program executed, in relation to the assembly?*
3. *Could SPS be used to write a program with no mnemonic operation codes and no symbolic addresses?*
4. *When character adjustment is used, do the symbolic address and the character adjustment ever get into the object program* separately?
5. *Absolute addresses are written on the SPS coding form as four digits. Does this mean that they appear as four digits in the object program?*

PG	LIN	CT	LABEL	OP	A OPERAND	B OPERAND	D	LOC	INSTRUCTION	COMMENTS
1	010	4	START	CS	0080			0333	/ 080	CLEAR READ
1	020	4		CS	0299			0337	/ 299	AND PRINT
1	030	4		CS	0332			0341	/ 332	STORAGE AREAS
1	040	4		SW	READ1 -009			0345	, 014	SET WM
1	050	1	REPEAT	R				0349	1	READ
1	060	7		MCW	READ1	TOTAL		0350	M 023 419	CARDS
1	070	1		R				0357	1	AND
1	080	7		A	READ1	TOTAL		0358	A 023 419	FORM
1	090	1		R				0365	1	TOTAL
1	100	7		A	READ1	TOTAL		0366	A 023 419	
1	110	1		R				0373	1	
1	120	7		A	READ1	TOTAL		0374	A 023 419	
1	121	1		R				0381	1	
1	122	7		A	READ1	TOTAL		0382	A 023 419	
1	130	7		A	ROUND	TOTAL		0389	A 408 419	ROUND TO $
1	140	7		MCW	TOTAL -002	PRINT1		0396	M 417 209	MOVE $ ONLY
1	150	4		W	REPEAT			0403	2 349	PRINT & REPEAT
1	160	2	ROUND	DCW	*		50	0408		
1	170	11	TOTAL	DCW	*			0419		
1	180		READ1	DS	0023			0023		
1	190	9	PRINT1	DCW	0209			0209		
1	200			END	START				/ 333 080	

Figure 4. Assembly listing of a slightly modified version of the program of Figure 1.

4.2 Further Information on the SPS Language and Processor

DCW and END are only two of the "instructions" to the processor. After considering some of the other *symbolic instructions* or *pseudo instructions*, we shall consider in a little more detail how the processor translates from a source program to an object program.

DCW automatically puts a word mark in a high-order character position of the constant that is defined with it. The DC pseudo instruction, which stands for Define Constant (no word mark), performs exactly the same functions as DCW but does not enter a word mark. It is ordinarily used to define the value of a symbol and the length of a field, in a situation where the word mark will be specified on the other field in a two-address instruction.

Both DCW and DC create constants which are punched on cards in the object program deck and are actually loaded into storage when the object program is loaded. This is true even if the constant is all blanks. The DS pseudo instruction, on the other hand, performs the functions of defining the length of a field, reserving space in storage for this field, and, if desired, associating a symbolic address with the field—but does not enter a constant into storage with the object program. It can be used when it is necessary to set up a storage location for intermediate or final results when a word mark is not needed in the field. The DS pseudo instruction can also be used for the *sole* purpose of defining the absolute equivalent of a symbol, by not putting anything in the count field. In the example of Section 4.1, for instance, a DS instruction was used to specify to the processor that READ1 was to stand for 0010. Situations will often arise where this is useful.

Origin is a pseudo instruction which has the symbolic operation code ORG. The only other field on such an instruction should be an absolute address in the A-operand portion. The processor will interpret this as an order to place the next character of the program in the location specified by the absolute address. This is most commonly used to indicate where the first instruction of the program should be located. In the absence of such an ORG at the beginning of the program, the first instruction is automatically placed in 333, which is the first location beyond the print area. It is also permissible, however, to have an origin instruction someplace other than the beginning, or even to have several of them. The latter might be useful, for instance, if it were desired to place the constants and the working storage in a group separated from the program.

A clearer understanding of the mechanics of the assembly process will be useful in writing correct SPS programs and avoiding certain types of errors. The operation of this processor program can best be explained in terms of an example. Let's see what the processor program would do in assembling the program of Figure 1.

With the program punched on cards having the column assignments shown on the coding sheet, the assembly can begin. The operation of the processor in doing this assembly consists of two rather distinct phases or *passes*. In the first pass the processor does little more than establish the "meanings" of the symbols and translate the mnemonic operation codes to actual. The processor does this by determining the storage location to be associated with each symbol, as it reads the entire program.

At the beginning of the program, the processor assumes that the first character of the program will later be loaded into location 333 unless it finds an origin card which specifies some other starting location. In the program in Figure 1, therefore, the label START would be entered into a *label table*, along with its absolute equivalent of 333. In order to keep track of the amount of storage required by the instructions and data in a program the processor must inspect each instruction or data word to determine its length. The first instruction would be found to require four characters (the operation code and one address). If there were a label on the next instruction, therefore, it would be given the absolute equivalent of 337. Proceeding in this manner, we see that symbol REPEAT would be entered into the label table with the absolute equivalent 349. The technique by which the processor keeps track of the location to which each symbol is equivalent, involves what is called the *location counter*. This is a field within the processor program which is started at 333 or whatever location is specified by the origin instruction, and is increased—as each card is read—by the number of characters needed to store the object program information created by that card. Any time another origin card is detected, the location counter is given the value specified on the origin card, without regard for previous contents of the location counter.

In our example, the location counter would start at 333 and be increased by the length of each instruction, as all of the instructions are read. Finally it would reach the first constant at the end of the instructions; now, the location counter must be increased by the length of the constant as given in the count field. Furthermore, constants and data are addressed by their low-order characters rather than the high-order by which instructions are addressed. Therefore, the label ROUND is associated with the address 400, not 399. Proceeding similarly, TOTAL would be entered in the label table as equivalent to 411. READ1 is made equivalent to 0010 and PRINT1 is made equivalent to 0209, since the actual addresses are specified in the source program. When the processor detects the END card, it stops reading cards and prepares for the second pass.

Notice that the processor has really not done much with the instructions so far. No symbolic addresses have been changed to absolute; this would clearly be impossible. For instance, the processor could not translate the symbolic address READ1 into an absolute address because at the time it finds this address it has not yet established the absolute equivalent of the symbol. This is the basic idea behind the two-pass operation.

On the second pass the processor uses the information in the label table to assemble absolute instructions as the source program cards are read again. The information in the label field (columns 8-13) is not used on the second pass; this information was needed only to define the absolute equivalents of the symbols. This time, as the first card is read, the four-digit address is converted to the three-character form. The assembled instruction is then punched into a card along with information to tell a subsequent loading program where in storage to put the instruction and its word mark. This process is carried out for each instruction, as the cards are read. Whenever a symbolic address is found, the processor looks in the label table to find the absolute equivalent in order to assemble the instruction. When an instruction is found which has character adjustment, the amount of the adjustment is added to or subtracted from the absolute equivalent found in the label table.

The constants are recognized by their operation codes as being constants rather than as being instructions and are assembled properly. In our program the 50, which is referred to by the symbol ROUND and the absolute address 400, would be punched on a card for loading with the object program. The other DCW constants are all blanks; these would also be put on cards for loading. For the purpose of our program it would only be necessary that sufficient information be punched on the card for setting a word mark; however, there is no provision in the SPS system for doing anything but literally loading the blanks. On the second pass the END card in the source program would cause the creation of a *transition card* in the object program. This card would be the last of the object deck and therefore would be read after the entire program had been loaded. It later causes the object program to take control of the computer system starting with the instruction specified by the address in the END instruction.

It should be emphasized once again that the result of the assembly is only the creation of an object deck. The object program is *not* executed and it is not even left in storage ready to be executed. With the assembly complete (and ordinarily with some checking for correctness), the object program can *then* be loaded into storage and executed.

Review Questions

1. *What is the difference between DCW and DC? Does either of them allow a symbol to be defined without loading anything into storage with the object program deck? How can this be done?*
2. *Why must SPS use two passes in assembling a program?*
3. *Suppose the same origin were given before the instructions and before the constants. What would happen when the object program is loaded?*
4. *What would happen if a symbol were used in the address part of the instructions in the program, but never appeared in the label column? Would the processor have any way of establishing the absolute equivalent of the symbol?*
5. *What would happen if a symbol were written* two *places in the label column? Would the processor have any way of knowing which one establishes the definition of the symbol?*

4.3 Case Study

The following case study will give us another opportunity to see how symbolic programming is used and at the same time introduce three new instructions.

The problem is in a greatly simplified element of a payroll calculation. We are given an input deck which consists alternately of payroll master cards and labor vouchers. The first card of the deck is a master, the second is a detail for the same man, the third is a master for the next man and the fourth is that man's detail, etc. Master cards have the pay number in columns 1-5, the name in columns 10-29 and the hourly pay rate in columns 53-56. The pay rate is given in dollars per hour to three decimals. A detail card has pay number in columns 1-5 and the hours worked, to hundredths of an hour, in columns 10-13. Both cards in practice would contain much other information. Our job is to read the cards, compute the gross pay assuming no overtime, and print the pay number, name and gross pay (to the nearest penny) on the printer. This is to be done for each man in the deck, without consideration of how to detect the last card (this problem is considered in the next section). The gross pay is to be printed with a dollar sign and a decimal point and with any leading zeros suppressed.

The source program is shown in Figure 5, where separate pages have been used for instructions and constants. This, incidentally, is a common way to write a symbolic program; often, the constants are entered on the separate page as they are first used in the program.

The program begins in this case with an origin instruction, which is used here primarily to illustrate the technique. It might be used in practice to avoid some other standard routine in the first part of available storage. After that we clear the read and print storage areas and set word marks, as before. Then we read the first card of the data deck, which will be a master card. We move the information on it from the read area to the print area and to a working area. This is done with a new instruction called Load Characters to A Word Mark. This instruction is somewhat analogous to the Move instruction but with a significant difference in the treatment of word marks. This instruction requires that only the A-field have a word mark; it is this word mark which stops the transmission of characters. Any word marks in the B-field are cleared, then the word mark from the A-field is transferred to the corresponding position in the B-field. This instruction can obviously be used only if the field to which the data is being moved is the same length as the source field; however, this is often the case and, when it is, this instruction removes the necessity of setting a word mark in the B-field.

Load Characters to A Word Mark

FORMAT	Mnemonic LCA	Op Code L	A-address xxx	B-address xxx
FUNCTION	This instruction is commonly used to load data into the printer or punch areas of storage, and also to transfer data or instructions from the read-in area to another storage area. The data and word mark from the A-field are transferred to the B-field, and all other word marks in the B-field are cleared.			
WORD MARKS	The A-field must have a defining word mark, because the A-field word mark stops the operation.			
TIMING	$T = .0115\ (L_I + 1 + 2L_A)$ ms.			

The third LCA instruction, written with character adjustment, moves the hourly pay rate to the multiplier field. The next instruction reads the detail card, obtaining the hours worked in the HOURS field, and we are ready to multiply to get the gross pay.

Multiplication in the 1401 is a special feature which permits multiplication by built-in machine hardware. (In the absence of this special feature, multiplication can be programmed.) On a Multiply instruction the A-address specifies the units position of the multiplicand; this field must have a word mark. The B-address of a Multiply instruction addresses the units position of a rather special field which initially contains the multiplier and in the end contains the product. The multiplier must be in the *high-order* positions of this special field before

Page No. 1 of 2

LINE 3–5	COUNT 6–7	LABEL 8–13	OPERATION 14–16	(A) OPERAND ADDRESS 17	(A) ± 23	(A) CHAR. ADJ.	(A) IND. 27	(B) OPERAND ADDRESS 28	(B) ± 34	(B) CHAR. ADJ.	(B) IND. 38	d 39	COMMENTS 40–55
010			ØRG	0600									
020		BEGIN	CS	0080									CLEAR READ
030			CS	0299									AND PRINT
040			CS	0332									STØRAGE AREAS
050			SW	PAYNØ	-	004		NAME	-	019			SET
060			SW	PAYRTE	-	003		HØURS	-	003			WM
070		PRØG	R										MASTER
080			LCA	PAYNØ				PRINT1					PAY NUMBER
090			LCA	NAME				PRINT2					NAME
100			LCA	PAYRTE				MULT	-	005			PAY RATE
110			R										DETAIL
120			M	HØURS				MULT					GET GRØSS PAY
130			A	RØUND				MULT	-	002			RØUND TØ CENTS
140			LCA	EDIT				PRINT3					
150			SW	MULT	-	007							
160			MCE	MULT	-	003		PRINT3					EDIT GRØSS
170			CW	MULT	-	007							
180			W										PRINT
190			B	PRØG									BRANCH TØ REPEAT
200													

Figure 5. SPS program to compute and print gross pay from hours worked and pay rate read from cards.

Page No. 2 of 2

LINE 3–5	COUNT 6–7	LABEL 8–13	OPERATION 14–16	(A) OPERAND ADDRESS 17	(A) ± 23	(A) CHAR. ADJ.	(A) IND. 27	(B) OPERAND ADDRESS 28	(B) ± 34	(B) CHAR. ADJ.	(B) IND. 38	d 39	COMMENTS 40–55
010			ØRG	0800									
020		PAYNØ	DS	0005									
030		NAME	DS	0029									
040		PAYRTE	DS	0056									
050		HØURS	DS	0013									
060		PRINT1	DS	0205									
070		PRINT2	DS	0230									
080		PRINT3	DS	0243									
090	01	RØUND	DCW	*		5							
100	07	EDIT	DCW	*		$	0	.					
110	09	MULT	DCW	*									
120			END	BEGIN									
130													
140													
150													
160													
170													
180													
190													
200													

Figure 5. Continued

the instruction is executed. The field must be one character position longer than the sum of the number of digits in the multiplier and the multiplicand. For instance, in our case we have four digits each; therefore the field has been established as nine character positions long. (This requirement is based on the way the machine multiplies.)

Multiply

	Mnemonic	Op Code	A-address	B-address
FORMAT	M	@	xxx	xxx

FUNCTION The multiplicand (data located in the A-field) is repetitively added to the data in the B-field. The B-field contains the multiplier in the high-order positions, and enough additional positions to allow for the development of the product. At the end of the multiply operation, the units position of the product is located at the B-address. The multiplier is destroyed in the B-field as the product is developed. Therefore, if the multiplier is needed for subsequent operations, it must be retained in another storage area.

Rule 1. The product is developed in the B-field. The length of the B-field is determined by adding 1 to the sum of the number of digits in the multiplicand and multiplier fields.

```
   Example:
   1246          4-digit multiplicand
 × 543           3-digit multiplier
               + 1
                 8 positions must be allowed
                   in the B-field.
```

Rule 2. A word mark must be associated with the high-order positions of both the multiplier and multiplicand fields.

Rule 3. A- and B-bits need not be present in the units positions of the multiplier and multiplicand fields. The absence of zone bits in these positions indicates a positive sign. At the completion of the multiply operation the B-field will have zone bits in the units position of the product only. The multiply operation uses algebraic sign control:

Multiplier Sign	+	+	−	−
Multiplicand Sign	+	−	+	−
Sign of Product	+	−	−	+

Rule 4. Zone bits that appear in the multiplicand field are undisturbed by the multiply operation. Zone bits in the units position of the multiplicand are interpreted for sign control.

WORD MARKS A word mark must be associated with the high-order positions of the multiplier and multiplicand fields.

TIMING The average time required for a multiply operation is:

$T = .0115\ (L_I + 3 + 2L_C + 5L_CL_M + 7L_M)$ ms.

L_C = length of multiplicand field.

L_M = length of multiplier field.

Note: The first addition within the multiply operation inserts zeros in the product field from the storage location specified by the B-address up to the units position of the multiplier.

One of the numbers that are multiplied in this operation has three places to the right of the decimal point, and the other has two to the right. These decimal points are of course not punched on the card; they are *understood.* To interpret the result we must decide where we understand the decimal point of the product to be. This can be obtained by applying the usual rule: the number of places to the right of the point in the product is equal to the number of places to the right of the point in the multiplier plus the number of places to the right in the multiplicand. This means that the eight-digit product will have five decimal places. We want to round this product to the nearest cent, which requires adding a 5 one position to the right of the pennies amount. This turns out to be two characters to the left of the units position of the field, so we add the 5 to the product field with a character adjustment of -2.

The gross pay is now available in storage, rounded to the nearest penny. Before printing it, however, we would like to insert a decimal point between the dollars and cents, arrange to print a dollar sign, and delete any zeros in front of the first significant digit. All of this can be done with the Move Characters and Edit instruction. This instruction requires the use of an edit word which contains the characters to be inserted in the edited amount, along with (in our case) a character to signal the use of zero suppression. The edit word is first loaded into the print storage area. This edit word is $bb0.bb, where the b's stand for blanks, as shown in the constants in Figure 5. When the Move Characters and Edit instruction is executed, the data from the A-field is inserted into the character positions in the B-field occupied by blanks or zeros, and high-order zeros are replaced with blanks.

A few examples will show what can be done with this powerful instruction.

A-field	B-field before	B-field after
08828	$bb0.bb	$b88.28
08828	$bbb.bb	$088.28

The zero in the edit word calls for zero suppression, and also defines the rightmost character position to which it is to be applied, as this example shows:

00067	$b0b.bb	$bb0.67

Zero suppression applies to commas to the left of the first significant digit:

000294368	b,bbb,bb0.bb	bbbb2,943.68

This instruction performs certain other editing operations also, as described in the summary box.

Move Characters and Edit

FORMAT

Mnemonic	Op Code	A-address	B-address
MCE	E	xxx	xxx

FUNCTION The Move Characters and Edit instruction modifies the data in the A-field by the contents of the edit-control word in the B-field, and stores the result in the B-field.

Define the *body* of the edit-control word as the part beginning with the rightmost blank or zero, and continuing to the left until the A-field word mark is sensed. The remaining portion is called the *status* portion.

The following rules control the editing operation.
Rule 1. All numerical, alphabetic and special characters can be used in the control word. However, some of these have special meanings:

Control Character	Function
b (blank)	This is replaced with the character from the corresponding position of the A-field.
0 (zero)	This is used for zero suppression, and is replaced with a corresponding character from the A-field. Also the rightmost 0 in the control word indicates the rightmost limit of zero suppression.
. (period)	This is undisturbed in the punctuated data field, in the position where written.
, (comma)	This is undisturbed in the punctuated data field, in the position where written, unless zero suppression takes place, and no significant numerical characters are found to the left of the comma.
CR (credit)	This is undisturbed in the status portion if the data sign is negative. It is deleted if the data sign is positive. Can be used in body of control word without being subject to sign control.
— (minus)	Handled in the same way as CR.
& (ampersand)	This causes a space in the edited field. It can be used in multiples.
* (asterisk)	This can be used in singular or in multiple, usually to indicate class of total.
$ (dollar sign)	This is undisturbed in the position where it is written.

Rule 2. A word mark with the high-order position of the B-field controls operation.
Rule 3. When the A-field word mark is sensed, the remaining commas in the control field are set to blanks.
Rule 4. The data field can contain fewer, but must not contain more, positions than the number of blanks and zeros in the body of the control word.

TIMING $T = .0115\ (L_I + 1 + L_A + L_B + L_Y)$ ms.

The A-field on a Move Characters and Edit instruction is required not to have more characters than the number of zeros and blanks in the edit word. Since the multiplication process always puts a zero in the high-order character of the product, it is necessary to set a word mark one position to the right of the high-order character in order to satisfy this rule. After the editing has been performed the word mark should be removed so that it will not disturb later operations with this field, when the next card is read.

With the edited gross pay in the print storage area it is now possible to write the line on the printer and branch back to PROG to read another card and start over.

The constants are shown preceded by an origin instruction, which once again is used mostly for illustrative purposes.

The definitions of the read and print area fields are all made with DS instructions, since nothing can be accomplished with any of them by loading word marks into storage. Word marks are not needed in the print area, and it is not permissible to load constants into the read area.

The DCW with the label ROUND is used to enter a 5 for rounding; the EDIT DCW puts into storage the edit constant; and the MULT DCW sets up the working storage location for the multiplier and the product. These last three DCW instructions are shown with an asterisk in the address field, to indicate that the processor may assign these constants in sequence as the program is assembled. Notice on the assembly listing in Figure 6 that the rounding constant is to be loaded into character position 800; the seven pseudo instructions between the origin and this DCW had no effect on the location counter since they specified absolute locations for the symbols. The END instruction, as usual, specifies that no more source program cards follow, and the address will cause the object program to begin executing instructions at the address shown.

Notice that the comments which were written on the coding sheets have been transferred to the assembly listing. They have no effect on the assembly and are provided for the convenience of the programmer and for others who may have to read the program. The use of comments is strongly recommended.

Review Questions

1. *What is the difference between the instructions Move Characters to A or B Word Mark, and Load Characters to A Word Mark?*
2. *Describe the operation of the Multiply instruction.*
3. *What characters in the control word (edit word) are always replaced by characters from the A-field?*
4. *Discuss the reasons for using a combination of DS and DCW pseudo instructions in this program. Could DCW be used throughout? Could DS be used throughout?*
5. *How would the object program be changed if both ORG instructions were omitted? Would the execution of the object program give the same results?*

PG	LIN	CT	LABEL	OP	A OPERAND	B OPERAND	D	LOC	INSTRUCTION	COMMENTS
1	010			ORG	0600					
1	020	4	BEGIN	CS	0080			0600	/ 080	CLEAR READ
1	030	4		CS	0299			0604	/ 299	AND PRINT
1	040	4		CS	0332			0608	/ 332	STORAGE AREAS
1	050	7		SW	PAYNO -004	NAME -019		0612	, 001 010	SET
1	060	7		SW	PAYRTE-003	HOURS -003		0619	, 053 010	WM
1	070	1	PROG	R				0626	1	MASTER
1	080	7		LCA	PAYNO	PRINT1		0627	L 005 205	PAY NUMBER
1	090	7		LCA	NAME	PRINT2		0634	L 029 230	NAME
1	100	7		LCA	PAYRTE	MULT -005		0641	L 056 811	PAY RATE
1	110	1		R				0648	1	DETAIL
1	120	7		M	HOURS	MULT		0649	@ 013 816	GET GROSS PAY
1	130	7		A	ROUND	MULT -002		0656	A 800 814	ROUND TO CENTS
1	140	7		LCA	EDIT	PRINT3		0663	L 807 243	
1	150	4		SW	MULT -007			0670	, 809	
1	160	7		MCE	MULT -003	PRINT3		0674	E 813 243	EDIT GROSS
1	170	4		CW	MULT -007			0681	⌑ 809	
1	180	1		W				0685	2	PRINT
1	190	4		B	PROG			0686	B 626	BRANCH TO REPEAT
2	010			ORG	0800					
2	020		PAYNO	DS	0005			0005		
2	030		NAME	DS	0029			0029		
2	040		PAYRTE	DS	0056			0056		
2	050		HOURS	DS	0013			0013		
2	060		PRINT1	DS	0205			0205		
2	070		PRINT2	DS	0230			0230		
2	080		PRINT3	DS	0243			0243		
2	090	1	ROUND	DCW	*		5	0800		
2	100	7	EDIT	DCW	*		$ 0.	0807		
2	110	9	MULT	DCW	*			0816		
2	120			END	BEGIN				/ 600 080	

Figure 6. Assembly listing of the program of Figure 5.

Exercises

*1. Give the absolute equivalent of each symbol in the program of Figure 1 (before the corrections).

2. Give the absolute equivalent of each symbol in the program of Figure 5.

*3. "Assemble" the program shown in Figure 7 "by hand." That is, carry out the same analysis of the symbolic program that the SPS processor would do, ending with an absolute program.

4. Assemble the program shown in Figure 8 by hand.

5. What is wrong with the following reasoning? It is desired to set up a program to handle data cards on which the fields are of variable position. To handle this, the absolute addresses of the field-defining DS instructions will be given by additional numbers on the data cards.

6. Extend the program of the Case Study as follows. For each man, there are *three* cards: the first and second are as before, and the third gives the man's deductions. The format of the deductions card is:

Columns	1-5	Pay number
	6-8	Social Security
	9-12	Withholding tax
	13-16	Savings bonds
	17-20	Union dues

The processing will now consist of computing the gross pay and the net pay. For each man, a line should be printed, as follows:

Positions	1-5	Pay number
	11-30	Name
	36-41	Gross pay
	47-50	Social Security
	56-60	Withholding tax
	66-70	Savings bonds
	76-80	Union dues
	86-91	Net pay

The six dollar amounts should be printed with decimal points but without dollar signs.

Page No. 1 of 1

LINE 3-5	COUNT 6-7	LABEL 8-13	OPERATION 14-16	(A) OPERAND ADDRESS 17	± 23	CHAR. ADJ.	IND. 27	(B) OPERAND ADDRESS 28	± 34	CHAR. ADJ.	IND. 38	d 39	COMMENTS 40-55
010			ØRG	0500									
020		ABC	CS	0080									
030			CS	0299									
040			CS	0332									
050			SW	A1	-	004		A2	-	004			
060			SW	A3	-	004		B1	-	003			
070		BCD	R										
080			A	A1				TØT					
090			A	A2				TØT					
100			S	A3				TØT					
110			A	HALFD				TØT	-	001			
120			MCS	TØT	-	002		B1					
130			W	BCD									
140		A1	DS	0005									
150		A2	DS	0010									
160		A3	DS	0015									
170		B1	DS	0210									
180	06	TØT	DCW	*		000	0	00					
190	01	HALFD	DCW	*		5							
200			END	ABC									

Figure 7. SPS program for exercise 3.

Page No. 1 of 1

LINE 3–5	COUNT 6–7	LABEL 8–13	OPERATION 14–16	(A) OPERAND ADDRESS 17	± 23	CHAR. ADJ.	IND. 27	(B) OPERAND ADDRESS 28	± 34	CHAR. ADJ.	IND. 38	d 39	COMMENTS 40–55
010			ØRG	0700									
020		HERE	CS	0080									
030			CS	0180									
040			SW	DATA1	-	004		DATA2	-	006			
050		READ	R										
060			LCA	DATA1				PUNCH1					
070			LCA	EDIT				PUNCH2					
080			MCE	DATA2				PUNCH2					
090			P	READ									
100		DATA1	DS	0005									
110		DATA2	DS	0023									
120		PUNCH1	DS	0105									
130		PUNCH2	DS	0115									
140	10	EDIT	DCW	*		$	,	0.					
150			END	HERE									
160													
170													
180													
190													
200													

Figure 8. SPS program for exercise 4.