

EUR 2637.e

VOL. II

EUROPEAN ATOMIC ENERGY COMMUNITY - EURATOM

THE COMPILATION AND PROCESSING OF
IBM 1401 PROGRAMS ON IBM 7090

VOL. II : THE COMPILER PROGRAM DESCRIPTION

by

A.F.R. BROWN

1966



Joint Nuclear Research Center
Ispra Establishment - Italy

Scientific Information Processing Center - CETIS

LEGAL NOTICE

This document was prepared under the sponsorship of the Commission of the European Atomic Energy Community (EURATOM).

Neither the EURATOM Commission, its contractors nor any person acting on their behalf :

Make any warranty or representation, express or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this document, or that the use of any information, apparatus, method, or process disclosed in this document may not infringe privately owned rights; or

Assume any liability with respect to the use of, or for damages resulting from the use of any information, apparatus, method or process disclosed in this document.

This report is on sale at the addresses listed on cover page 4

at the price of FF 16,50

FB 165,—

DM 13,20

Lit. 2 060

Fl. 11,95

When ordering, please quote the EUR number and the title, which are indicated on the cover of each report.

Printed by Guyot, s.a.
Brussels, January 1966

This document was reproduced on the basis of the best available copy.

EUR 2637.e

VOL. II

**THE COMPILATION AND PROCESSING OF IBM 1401 PROGRAMS ON
IBM 7090**

VOL. II : THE COMPILER PROGRAM DESCRIPTION by A.F.R. BROWN

European Atomic Energy Community - EURATOM

Joint Nuclear Research Center

Ispra Establishment (Italy)

Scientific Information Processing Center - CETIS

Brussels, January 1966 - 130 Pages - FB 165

In the field of non-numerical data processing it is often more profitable to use a medium-size computer instead of a big one. Compilation, however, may better be done on a bigger machine.

The four volumes of this report describe a symbolic programming language, its compiler for the IBM 7090 which produces IBM 1401 object programs, and a simulator permitting the execution of these programs on the IBM 7090.

EUR 2637.e

VOL. II

**THE COMPILATION AND PROCESSING OF IBM 1401 PROGRAMS ON
IBM 7090**

VOL. II : THE COMPILER PROGRAM DESCRIPTION by A.F.R. BROWN

European Atomic Energy Community - EURATOM

Joint Nuclear Research Center

Ispra Establishment (Italy)

Scientific Information Processing Center - CETIS

Brussels, January 1966 - 130 Pages - FB 165

In the field of non-numerical data processing it is often more profitable to use a medium-size computer instead of a big one. Compilation, however, may better be done on a bigger machine.

The four volumes of this report describe a symbolic programming language, its compiler for the IBM 7090 which produces IBM 1401 object programs, and a simulator permitting the execution of these programs on the IBM 7090.

EUR 2637.e

VOL. II

**THE COMPILATION AND PROCESSING OF IBM 1401 PROGRAMS ON
IBM 7090**

VOL. II : THE COMPILER PROGRAM DESCRIPTION by A.F.R. BROWN

European Atomic Energy Community - EURATOM

Joint Nuclear Research Center

Ispra Establishment (Italy)

Scientific Information Processing Center - CETIS

Brussels, January 1966 - 130 Pages - FB 165

In the field of non-numerical data processing it is often more profitable to use a medium-size computer instead of a big one. Compilation, however, may better be done on a bigger machine.

The four volumes of this report describe a symbolic programming language, its compiler for the IBM 7090 which produces IBM 1401 object programs, and a simulator permitting the execution of these programs on the IBM 7090.

This volume explains the structure of the IBM 7090 compiler program. Comments are given on the flow charts of this program and of some subroutines in the 1401 program package that handle magnetic tape input and output. The flow charts themselves are published in the fourth volume of the report.

This volume explains the structure of the IBM 7090 compiler program. Comments are given on the flow charts of this program and of some subroutines in the 1401 program package that handle magnetic tape input and output. The flow charts themselves are published in the fourth volume of the report.

This volume explains the structure of the IBM 7090 compiler program. Comments are given on the flow charts of this program and of some subroutines in the 1401 program package that handle magnetic tape input and output. The flow charts themselves are published in the fourth volume of the report.

EUR 2637.e

VOL. II

EUROPEAN ATOMIC ENERGY COMMUNITY - EURATOM

THE COMPILATION AND PROCESSING OF
IBM 1401 PROGRAMS ON IBM 7090
VOL. II : THE COMPILER PROGRAM DESCRIPTION

by

A.F.R. BROWN

1966



Joint Nuclear Research Center
Ispra Establishment - Italy

Scientific Information Processing Center - CETIS

TABLE OF CONTENTS

	page
Comments on the Flow Charts of the Compiler Program	1-113
Comments on the Flow Charts of some 1401 Magnetic Tape Input and Output Subroutines	114-129

S U M M A R Y

In the field of non-numerical data processing it is often more profitable to use a medium-size computer instead of a big one. Compilation, however, may better be done on a bigger machine.

The four volumes of this report describe a symbolic programming language, its compiler for the IBM 7090 which produces IBM 1401 object programs, and a simulator permitting the execution of these programs on the IBM 7090.

This volume explains the structure of the IBM 7090 compiler program. Comments are given on the flow charts of this program and of some subroutines in the 1401 program package that handle magnetic tape input and output. The flow charts themselves are published in the fourth volume of the report.

In the flow charts of the compiler system (see pp. 2-66 of the fourth volume) there are three kinds of boxes:

(1) with top and bottom lines made of minus signs, and straight ends. Such a box represents an action to be taken, occasionally with a comment or explanation.

(2) with top and bottom lines made of equal signs, and curved ends. Such a box represents a call on a subroutine, whose name is given inside the box. Occasionally, instead of the simple name of the subroutine, the whole calling sequence is given, beginning with a TSX instruction. And also occasionally, such a box may include an explanation of what the subroutine is supposed to achieve. If the subroutine has more than one exit, these will be represented by two or more numbered lines coming from the box. The numbers correspond to the numbers in "RETURN 1", "RETURN 2", etc., or "EXIT 1", "EXIT 2" etc. in the flow charts for the subroutine themselves.

(3) with top and bottom lines made of asterisks, and pointed ends. Such a box represents a question or a switch test.

Normally it will have two exits, labelled plus and minus, i.e. yes and no. Occasionally the exits may be labelled more explicitly, such as "MATCH" and "NO MATCH". Also occasionally, there may be more than two exits, as in the case of some boxes that contain the sentence "BRANCH ON FILE TYPE", from which there are as many exits as there are types of file, each one labelled with the name of one type of file.

A symbol sitting immediately above a box is usually the 7090 program label of the first instruction involved in doing what the box calls for. An isolated symbol from which an arrow leads to

Manuscript received on October 27, 1965.

a box has the same meaning, but indicates that branches come to the box from elsewhere in the flow chart, or from other flow charts. Such a symbol will not have an arrow leading to it. Instead, there must somewhere be the same symbol, isolated, with an arrow leading to it from a box.

Sometimes a call on a subroutine is not shown by a subroutine box, but by something like " USE =BT4D= " within an action box. "BT4D" is the name of the subroutine, and in this context it is put between equal signs to emphasize that it is the name of a subroutine. In very many flow charts, the phrase "OUTPUT SPS CARDS" occurs. This means that there is a call on subroutine SPS in the program, but the subroutine as such is not mentioned.

Single parentheses are used to mean "the contents of"; e.g. (S&3) would mean "the contents of location S&3". Double parentheses are used as quotation marks to enclose literals.

Occasionally expressions like "E.5", "E.M" or "E.M&2" occur. These refer to "elements" in a statement, and the three expressions above would refer to the fifth, Mth, and M-plus-twoth elements in the current statement. Recall that each compiler-language statement is broken up into elements, each consisting of a word, a punctuation, a literal, or a series of one or more blanks. If a statement has a label, this is E.1; the blank after it is E.2; and the first word of the function code is E.3. If a statement has no label, its initial series of blanks is E.1; then a fictive blank is assumed as E.2, so that the first word of the function code will again be E.3.

Similarly, expressions like "C.8" and "C.I" are sometimes used to indicate the 8th, or the Ith, in some series of characters read from left to right.

Flow chart 1 shows subroutine BREAD, which is used to "read" card images from the monitor input tape. Fundamentally, reading is done by the IOCS subroutine ".READ" . However, the card images may be of mixed BCD and binary cards, so subroutine "BREB" (see flow chart 2) is provided to examine the look-ahead bits, stored in location LOOKA , and decide how to use ".READ" . Subroutine BREB cannot be used directly by the compiler, however, because in compiling one statement it is sometimes necessary to know whether the next statement has a label, and if so what. The compiler must be able to "peek" at the next statement, which might be separated from the current statement by up to eight comment cards, without formally reading it.

So subroutine BREAD initially reads ten records from the input tape, and then supplies the first one as the record that has been logically read by the single calling of the subroutine. The next time BREAD is called, it will present the second record as the one that has been logically read, and it will also physically read the eleventh record. Thus BREAD uses ten card-length buffers, logically arranged in a ring. One of them always contains the card most recently "read", and the other nine contain the following nine cards from the input tape.

The calling sequence for BREAD is:

TSX BREAD,4

PZE V

where V is the address to which the program is to branch if the end of the input file is read. This is called "return 1", and the AC contains zero. Otherwise, "return 2" takes place, with a branch to the second word after the TSX instruction, with PZE Y,,M in the AC and stored at INAD. Y is the address of the first word of the card image, and M is 0 for a BCD card or 1 for a binary card.

Flow chart 2 shows the subroutine BrEB. It is used only by subroutine BREAD. The flow chart is self-explanatory.

Flow chart 3 shows the subroutine PREAD, which stands between the main program and BREAD. The main program calls on BREAD only at WUNA -- see flow chart 131 --, when one program has been compiled and the compiler is looking for the beginning of the next one on the input tape. While compiling a program, the main program calls on BREAD via PREAD. PREAD filters out

- (a) binary cards, which cause a branch to ERR56.
- (b) cards in SPS language, which get their labels handled exactly like labels of compiler language statements, so that compiler language statements and SPS instructions can branch to each other. However, if an SPS card has an op-code beginning with D, it is a storage-definer, not an instruction, and so its label is not treated like the label of a compiler language statement. The storage area or constant defined by such an SPS card cannot be referenced by compiler language statements, but only by other SPS cards.

After subroutine "LAB" has been used to handle the label, if necessary, subroutine "WRATE" is used to write out the SPS card on file WINTER, just as if it were an SPS card into which a compiler language statement had been translated. For subroutine WRATE, see flow chart 5. After that, subroutine BREAD is called again, until a card that is not an SPS card is found.

If the input file ends before a non-SPS and non-binary card is read, return 1 takes place, and the program branches to V as given in the calling sequence. Otherwise, return 2 is taken, with PZE Y in the AC and stored at INAD, Y being the address of the first word of the card image.

The use of PREAD to filter off SPS cards from the cards read by BREAD is an unnecessary complication. It would be simpler to read a card through BREAD, handle its label whether it was in SPS or in compiler language, compile it (a trivial task in the case of the SPS card), and then repeat the process for the next card.

The reason for PREAD is historical. Originally, the compiler did not accept SPS cards, and read the input directly through BREAD. Then it was modified to allow SPS cards, but it still took no notice of their labels, so that an SPS instruction could branch to a compiler language statement, but not vice versa. At this stage the use of PREAD as a filter, or a gate which by-passed SPS cards around the compiler proper and sent them straight to subroutine WRATE was the simplest procedure. Then it was decided that the labels of SPS instructions should go into the symbol table, and that the compiler might sometimes construct a label and force it on a label-less SPS card. But rather than modify the compiler proper,

giving it a simple section for SPS cards, it seemed safer to leave it unchanged and put the processing of SPS cards in PREAD.

Flow chart 4 shows subroutine D4V, used for converting 4-digit decimal numbers into binary form. The subroutine is entered with a word in the AC whose leftmost four characters are supposed to be decimal digits, and it is exited with the equivalent as a positive binary number in the AC. There is no protection against any of the supposed decimal digits being letters or other characters.

Flow chart 5 shows subroutine WRATE, whose principal function is to take card images in SPS language, put any symbols they may define into a symbol table, and write them out on a work tape, file WINTER. From there they will later be read back by a part of the program that functions as the second pass of an SPS assembler.

A counter at LCTR initially contains 0628, as the first available address for instructions to be assigned to. This counter is maintained by subroutine WRATE. Similarly, a counter at NAFL initially contains 7949 as the highest address to which the right-hand end of a constant can be assigned. Cells 7950-7999 are kept as a 50-character word of blanks for use by compiler language statements containing the word BLANKS. NAFL is not maintained by WRATE. It is counted downwards, as space is assigned to constants, while LCTR is counted upwards for instructions. Obviously, if they meet, the 1401 memory has been completely assigned and cannot contain the program being compiled.

WRATE uses subroutines D4V, for which see flow chart 4, BT4D, for which see flow chart 8, and FPASS, for which see flow chart 14. ERR50 is in the program as a subroutine, but there is no flow chart for it. It merely writes out the error message "STATEMENT USED ZERO LENGTH CONSTANT TREATED AS ONE CHARACTER".

Flow chart 6 shows subroutine BT3D, which is used for converting binary numbers into the equivalent 1401 three-character addresses; e.g. for converting the binary form of 7949 into the address I9I . The subroutine is entered with x, the number to be converted, as a positive binary integer in the AC, and is exited with the three-character 1401 address in bits P,1-17 of the AC.

If the number being converted is above 7999, there will be an error. However, the main program never allows this to happen. Since the input to BT3D is binary, it cannot represent what a programmer has coded until it has been converted from the original input form to binary form, and it will at that time have been checked for being within the limit.

If one wished to modify the system so as to compile for a 1401 with 16000 positions of memory, one would extend the table beginning at BT3DC from eight words to sixteen words.

Flow chart 8 shows subroutine BT4D, which is just like BT3D, except that it is exited with a four-digit decimal number in bits P,1-23 of the AC. If the original binary number were above 9999, there would be an error, but this is prevented by the same safeguards as for BT3D.

If one wished to modify the system so as to compile for a

1401 with 16000 positions of memory, the complications would be greater here than for BT3D. BT4D could be modified so as to produce 5-digit decimal numbers, and these could be used in SPS card images without too much alteration of the compiler. But in the symbol tables used by the compiler, there are several cases in which one machine word contains a 3-bit number indicating the category of symbol, a four-decimal-digit number in 24 bits, giving the address of the thing symbolized, and nine bits giving the length of the thing symbolized. To fit a five-digit number into such a word, since the first digit would never be greater than 1, it would be sufficient to represent the first digit by one bit, which could be stolen from the nine-bit field containing the length of the thing symbolized. One would have to adjust the compiler wherever it refers to the symbol table.

Note that the four-character output from BT4D is often called, in other flow charts, the "four-digit decimal form" or "four-digit form" of the input. The three-character output from BT3D is analogously called the "three-digit decimal form" or "three-digit form" of the input, even though the first and third characters are not necessarily digits. Some naming convention had to be used on the flow charts, and something like "1401 machine-language address form" would have been too long.

Flow chart 7 shows subroutine DEV, which, like D4V, converts decimal numbers into binary form. The calling sequence gives an address A; the word at A must be of the form PZE B, where B is the

location of a word containing from one to six decimal digits, beginning with the leftmost character. The last decimal digit, if there are fewer than six, must be followed by at least one pure zero, i.e. binary 000000 . A zero, as a decimal digit, must be represented for DEV by binary 001010 .

DEV is used by the compiler thus: the programmer will have put a decimal number on a card, from one to six digits long, with a blank or punctuation at either end. Another routine, DECOM, will have broken up the contents of the card into words, blanks and punctuations; here numbers count as words. A series of consecutive 7090 memory words will contain representations of blanks and punctuations in the form of small negative numbers; locators for literals, with the form MZE A,,B where the literal is B characters long and is stored in the 7090 memory with its first character in the leftmost position of 7090 memory location A; and locators for words, having the form PZE A , where the word is stored with its first character in the leftmost position of 7090 memory location A. Any zeros that were part of the word as keypunched will be stored in the memory as BCD zeros (binary 001010). The word will be terminated in the 7090 memory by at least one pure zero following its last character. If the last character of the word happens to occupy the last character position of a 7090 location, the next word of 7090 memory will be set to pure zero, and reserved as part of the representation of this input word.

This digression may explain why the contents of A, as shown in the flow chart, must not be negative or zero; why the pure and BCD zeros are used as they are; and why there has to be a protection in DEV, since it deals with words not yet certainly known

to be numbers, against non-digital characters. If a non-digit is found, exit 1 is taken.

If there is no such error, exit 2 is taken, with the binary number in the AC as a positive integer.

Flow charts 9, 10 and 11 show subroutine DECOM. There is one frequent reference to this subroutine in the program. It is entered at DECOM in the top left-hand corner of flow chart 9. The calling sequence gives an address A, and a program card image is supposed to have been read into locations A to A+13. The first 72 characters of this are to be broken up into words, blanks, punctuations, and literals. 72 memory locations, beginning at DWD, are available for storing words and literals. They are always stored with their first characters in the leftmost character position of a memory location, the other characters to the right and into higher memory locations if necessary, with pure zeros as fillers if the last-used memory location is not completely filled. To define the end of a word, not a literal, its last character must be followed by at least one pure zero, so that an extra word of pure zeros may be needed after the last information character. To prevent confusion, zeros in the original card image are stored as BCD zeros (binary 001010).

If more storage space beyond DWD+68 is needed, there is an exit to ERR53; an error message is given, and the card is not compiled. This is hardly probable, given the difficulty of fitting more than so much into one program card. The only way to make an otherwise valid statement overflow in this way would be with a PRINT or COMPOSE statement. A section of the statement, describing something to be put in the print line, occupies two or three words in addition

to punctuation, and by using thirty or so sections to describe the print line, and extending the statement with a few continuation cards, one could make the statement uncompileable.

In DWD ff. the words and literals, items of uncertain length, are stored. 144 locations beginning at DRAD are reserved so that each item in a statement, including words, numbers, literals, blanks, and punctuations, can be represented in some way by exactly one word of 7090 memory. If storage space beyond DRAD&140 is needed, the same exit to ERR53 takes place as when the limit of DWD&68 is exceeded.

The successive items of a statement are represented by successive memory words beginning at DRAD as follows:

- (a) a series of one or more blanks by MZE 4.
- (b) a punctuation by MZE P, where P=1 for left parenthesis, 2 for comma, 3 for slash, 5 for asterisk, 7 for minus, 8 for right parenthesis, 9 for period, 10 for plus, or 12 for an equal sign.
- (c) a word or number by PZE A, where the first character of the word or number is stored in the leftmost position of cell A, and successive characters are stored rightwards and in successive cells, with at least one pure zero marking the end.
- (d) a literal by MZE A,,B , where the literal is stored approximately as in (c) above, but B is its length in characters.

If the last item in the statement is a series of one or more blanks, the representation of this is suppressed. The representation of the last item in the statement, apart from final blank, is followed by zeros in the memory up to and including DRAD&143.

If the first item in the statement is a series of one or more blanks, MZE 4 is put into both DRAD and DRAD&1, and the representa-

tion of the second item will go into DRAD&2. This is done because the choice at the beginning of a statement, from the programmer's point of view, is label or no label. This means that a statement with a label is one item longer than the same item without its label. To equalize them, and simplify the compiling of what follows the label, the extra "blank" is put in at the beginning to fill up the place of a label.

If the first character of a statement is an asterisk, it is considered to be a comment. It will not be compiled, and DECOM will not break it up. Instead, the exit near the top left-hand corner of flow chart 9 takes place, with zero in the AC. Otherwise, apart from error exits, the exit will be at the point marked "EXIT FROM DECOM", somewhat to the right of the middle of flow chart 11. The AC will contain PZE DRAD&P, where DRAD&P-1 is the location into which the representation of the last item in the statement has been stored.

From flow chart 9 there are exits to DCRC, which is the beginning of flow chart 11, and "DCAT OR DCDL", which are at the beginning of flow chart 10.

From flow chart 10 there are exits to:

- (a) ERR53, if the limits of DWD&68 and DRAD&140 referred to above are exceeded.
- (b) DCMD, which is on the top line of flow chart 9.
- (c) ERR56, if a binary card image is found in the middle of a program. Binary cards are allowed in the input file, but only as data, after the end of one program and before the next one if any. A binary card found within a program causes an error message and an immediate termination of the run.

(d) VEND -- but this should be impossible. If the card which subroutine DECOM is now working on contains the beginning of a literal but not its end, presumably the completion is to follow on a continuation card. At DCATD on flow chart 10, the program looks ahead at the next record, without formally reading it. If the next record is a tape mark, the compiler behaves as though the literal had been properly concluded, and the 72nd character of the current record were its last character. Only if this is not so; i.e. if it is known in advance that the next record on the input tape is not a tape mark, does the program enter subroutine PREAD on flow chart 10. Therefore the exit to VEND cannot occur.

Subroutines DECOM and PREAD are also called in flow charts 134 and 135, during the compilation of SORT FIRST PASS statements. Such statements must be coded as if they were four statements; this was thought to give more clarity on the page than putting all the material into one statement, which would certainly require continuation cards. The first statement is simply SORT FIRST PASS; the second begins with ENTRY and tells how records for the sort are obtained; the third begins with BUFFERS and names the save files to be used in the sort; and the fourth begins with EXIT and names the write files for outputting sorted blocks of records. For obtaining the second, third, and fourth of these, PREAD and DECOM are called in flow charts 134 and 135.

Flow chart 11 begins at DCRC, to which the branch comes from the lower right hand corner of flow chart 9. The exits are:

- (a) to DCMD, on the top line of flow chart 9.
- (b) to EXIT FROM DECOM, the normal exit from the subroutine.
- (c) to ERR56, for the same reasons and with the same consequences as in (c) at the bottom of the preceding page, describing flow chart 10.
- (d) to VEND, which is impossible for the same reason as in (d) at the top of the preceding page, except that nothing is broken by the end of the current card. If the next record is seen in advance to be a tape mark, the program does not enter PREAD and goes to DCRD, just as if it had seen that there was another card, but that it was not a continuation card.

Flow chart 12 shows subroutine ATTL, which constructs a symbol table in the form of a list beginning at location SNAL. This is one of several lists which are contained, apart from the very first word of each, such as the word at SNAL, in a workspace beginning at location WORK and ending at location WKLIM-1. For convenience in using index registers, the words in this workspace are addressed by their distances from WKLIM. If the workspace is exhausted, the compiler branches to location SYSER. This causes the error message "WKLIM OVERFLOWS", followed by a memory dump and the end of the run. This should never happen, since the size of the 1401 memory limits the number of symbols a program will contain. But if it did happen because of an enormous number of symbols, one could re-assemble the compiler and allow a bigger workspace by defining WKLIM as WORK&3000 or 4000 instead of 2000.

The calling sequence for ATTL is:

TSX ATTL,4

PZE A

BCI 1,ETC.

where location A contains PZE B, and location B contains the symbol (normally A will be an address in the series DRAD to DRAD&140, and B will be an address in the series DWD to DWD&68) and BCI 1,ETC. represents some word of information that describes the symbol.

Using the colon to mean "contains", we can describe the way the list works thus:

Initially,

SNAL: PZE 0,,0

After one symbol has been stored in the list -- call it SYMBOL1, and the word that describes it DESCR1,

SNAL: PZE X1,,X1

WKLIM-X1: PZE X1-1,,0

WKLIM-X1&1: SYMBOL1

WKLIM-X1&2: DESCR1

X1 will have been the number contained in location NWL; initially it is equal to WKLIM-WORK. When it reaches 0, the workspace is exhausted, and the branch to SYSER takes place. The various subroutines that use the workspace all refer to and alter NWL to keep track of the workspace available.

After a second symbol has been stored in the list:

```
SNAL: PZE X1,,X2
WKLIM-X1: PZE X1-1,,X2
WKLIM-X1&1: SYMBOL1
WKLIM-X1&2: DESCR1
WKLIM-X2: PZE X2-1,,0
WKLIM-X2&1: SYMBOL2
WKLIM-X2&2: DESCR2
```

After a third symbol has been stored in the list:

```
SNAL: PZE X1,,X3
WKLIM-X1: PZE X1-1,,X2
WKLIM-X1&1: SYMBOL1
WKLIM-X1&2: DESCR1
WKLIM-X2: PZE X2-1,,X3
WKLIM-X2&1: SYMBOL2
WKLIM-X2&2: DESCR2
WKLIM-X3: PZE X3-1,,0
WKLIM-X3&1: SYMBOL3
WKLIM-X3&2: DESCR3
```

and so on. In order to remove a symbol from the list -- say SYMBOL2 , it suffices to set the address of the word just before the word in the list, WKLIM-X2 in this case, to zero. That is, if the list is being examined, and WKLIM-X2 is found to contain a word with a

zero address, the routine will pass on immediately to WKLIM-X3 if the decrement in WKLIM-X2 is some number other than 0, which we call X3, or will decide that it has examined the whole list if that decrement is 0.

The things that "BCI 1,ETC." in flow chart 12 can represent are as follows:

(a) for symbols naming variables: a word whose first three bits are 101, next 24 bits contain the address, in four-digit decimal form, of the rightmost character of the variable, last 9 bits give the number of characters in the variable, expressed as a binary number.

(b) for symbols naming areas: a word of the form

PTH A,,B

where A and B are the binary forms of the addresses of the rightmost and leftmost characters of the area.

(c) for symbols naming statements that have not yet appeared in the program, but which have been mentioned in "branch to" addresses:

MZE 1

(d) for symbols naming statements that have appeared in the program, and to which a transition could occur from the preceding statements, without an explicit "GO TO" or other naming of the statements as possible branch addresses:

MZE

(e) for symbols naming statements that have appeared in the program, and to which no transition from their preceding statements is possible, but only some sort of programmed branch:

MZE X

where X is some very large address.

The only reason for the distinction between (d) and (e) above is that a symbol of type (d) will be shown in the automatically produced flow chart with two vertical lines joining it to the preceding box, while a symbol of type (e) will not be joined to the preceding box in this way. A symbol of type (e) may, however, be joined to the preceding box by a line representing an explicit branch.

A symbol is put into the table as type (e) if the preceding statement involves an unconditional branch, or a set of branches among which one must be chosen. That is, if the preceding statement is a GO TO, RETURN, or RELOAD statement, or a COMP statement in which none of the three branch addresses is "NXT", or an XEC statement in which there is at least one branch address after the subroutine name, so that "NXT" is not supplied automatically by the compiler, but no branch address after the subroutine name is "NXT", or finally, a PAUSE statement in which "PAUSE" is followed by a branch address.

(f) for symbols naming program switches:

MTW

(g) for symbols naming files:

a word whose first three bits are 000 for a read file, 001 for a write file, 010 for a copy file, and 111 for a save file;

next two bits are 00 for normal blocking, 01 for standard blocking, 10 for record-mark blocking, 11 for physical blocking

next four bits give the number of the tape unit as a four-bit decimal digit. (The number of the output unit of a copy file will not be stored; only that of the input unit.)

The right-hand 27 bits of the word are all 0 initially. If a DEFFLD statement defines field names for the file, a word we may describe as PZE A,,B will be or'd into this word in the symbol table. At that time, locations A-B through A-1 will contain the definitions of the field names. See the flow chart for function DEFFLD, number 110. B will have to be less than 2 to the 9th power; otherwise it would overflow to the field in which the tape number is stored. To exceed this limit, there would have to be more than 255 fields defined for a single file.

The exits from subroutine ATTL are the normal one, and SYSER if the workspace for lists is exhausted.

Flow chart 13 shows subroutine SCLAB, which searches the list beginning at SNAL for symbols. The calling sequence gives an address A, which will normally be somewhere between DRAD and DRAD&140. The word at location A must contain B in its address part, where the word at location B is the symbol that must be sought in the table. If the symbol is not in the table, exit 2 takes place; if it is in the table, exit 1 takes place, and the AC contains PZE Y , where WKLIM-Y is the address at which the symbol is stored in the symbol table. WKLIM-Y&1 will contain the information about the symbol, as it was stored by subroutine ATTL.

Flow chart 14 shows subroutine FPASS (short for "first pass" i.e. first pass in an SPS assembly). It is used in three different places by subroutine WRATE (see flow chart 5) and these are its only uses in the compiler.

When the subroutine is entered, locations SCARD to SCARD&13

are supposed to contain the image of an SPS card, with a symbol in columns 8 through 13. The AC contains a binary number equal to the 1401 address that has been assigned to that symbol. If the symbol consists of six blanks, there is really no symbol, and the subroutine exits after doing nothing further. Otherwise, the symbol and its binary equivalent address are stored in a list beginning at location ASTB. It will be noted in the flow chart that before being stored, the symbol has any record marks it may contain converted to dollar signs and any BCD zeros (binary 001010) into pure zeros. Obviously record marks do not occur in symbols; it is just that table RMTAB does both alterations. BCD zeros are converted to pure zeros because the symbols in the table will have to be matched with information that has been written in the BCD mode on file WINTER and then read back in BCD as file RINTER; thus any character originally coded as a zero will appear in memory as a pure zero at this time.

As with the list made by subroutine ATTL, the words in this list address each other according to their distance from WKLIM, and the location of the next available cell in the list work space is saved in location NWL in the form of PZE N, where WKLIM-N is the next available cell.

The list beginning at ASTB is effectively the symbol table of the SPS program into which the input program is translated by the compiler.

It is organized as follows, using the colon for "contains":

initially

ASTB: PZE 0,,0

the first time a symbol is stored, let us call it SYMBOL1 and the

equivalent address in binary A1, and suppose that location NWL at that moment contains PZE X1

```
ASTB:   PZE X1,,X1
WKLIM-X1: PZE 0
WKLIM-X1&1: BCI 1,SYMBOL1
WKLIM-X1&2: PZE A1
```

the second time a symbol is stored, let us call it SYMBOL2 and the equivalent address in binary A2, and suppose that location NWL at that moment contains PZE A2. The the list becomes this:

```
ASTB:   PZE X1,,X2
WKLIM-X1: PZE X2
WKLIM-X1&1: BCI 1,SYMBOL1
WKLIM-X1&2: PZE A1
WKLIM-X2: PZE 0
WKLIM-X2&1: BCI 1,SYMBOL2
WKLIM-X2&2: PZE A2
```

The third time a symbol is stored, using the same naming conventions, the list becomes this:

```
ASTB:   PZE X1,,X3
WKLIM-X1: PZE X2
WKLIM-X1&1: BCI 1,SYMBOL1
WKLIM-X1&2: PZE A1
WKLIM-X2: PZE X3
WKLIM-X2&1: BCI 1,SYMBOL2
```

WKLIM-X2&2: PZE A2

WKLIM-X3: PZE 0

WKLIM-X3&1: BCI 1,SYMBOL3

WKLIM-X3&2: PZE A3

And so on. Symbols are never deleted from this list. It is consulted only at one point in the compiler -- at WACC in subroutine WAC, flow chart 129.

The only exits from subroutine FPASS are the normal one and the exit to SYSER if the list work space is exhausted.

Flow chart 15 has, in its top line, four addresses: FIRST, SECOND, VA, and VPA. FIRST is the address at which the compiler begins if it has been loaded as a complete binary deck, followed by a second file containing the 1401 input-output program package and the 1401 loader. SECOND is the address at which it begins if it has been loaded as a short binary deck, which counts on finding both the rest of the compiler and the 1401 program package and loader in disc storage. In that case the short binary deck is followed immediately by the file of cards to be compiled. VA is the address at which the compilation of each individual statement in an input program begins; thus, whenever a flow chart shows that the compilation of a statement has been completed, it shows a branch to VA, to begin work on the next statement. VPA is the point to which the program branches after assembling the condition in a conditional statement, if this is followed by something other than a GO TO statement.

E.g., consider the statements

```
IFSWITCH B GO TO MILANO
```

```
IFSWITCH B PRINT 1/$FIRST PASS FINISHED$
```

For both statements, processing begins at VA for the preliminaries, and then goes to IFSWIT (flow chart 78) for processing the condition IFSWITCH B . The same part of the compiler will take care of GO TO MILANO in the first statement above, but PRINT 1/\$FIRST PASS FINISHED\$ in the second will bring the program back to VPA. PRINT 1/\$FIRST PASS FINISHED\$ will be processed as a complete statement except for what lies between VA and VPA, namely the reading of the statement by subroutine PREAD, its decomposition into elements by subroutine DECOM (these have both been done already for the entire statement including its condition) and everything to do with labelling the statement, since a conditional statement cannot contain any label except the one that may come at the very beginning, before IFSWITCH.

Let us return to FIRST. When put on the computer for the first time, so to speak, the compiler reads the next file of the normal input tape into a large storage area, and then writes out the whole of itself plus the storage area, in disc storage, cylinder 249, module 0. The writing on disc is done by subroutine QWAD, which we do not attempt to describe here. The call on QWAD at FIRSF is the only time it is used in the program.

Note however, FIRSE, the second box in flow chart 15, and the box below and the box to the right of it. These concern "patch cards", which may be put at the beginning of the first file

read by the compiler. A patch card is recognized by having six asterisks in columns 73-78. But after the first non-patch card is read, no later card will be tested to see if it is a patch card. The format of a patch card would be tedious to explain, and can be easily deduced from the compiler program listing from FIRSB-1 to FIRSA-1. It is identical with the format of patch cards used by the 7090 program for simulating the 1401, which are fully discussed in the description of that program.

Patch cards are used to make minor corrections and modifications in the compiler program itself.

The short binary deck of the compiler program, whose preparation is described elsewhere, will cause a branch after loading to SECOND . Immediately, subroutine QRAD is executed, which should restore the computer memory to exactly the condition it was in just after QWAD was executed the last time; i.e. by reading back from disc storage just what subroutine QWAD wrote out.

After QRAD is executed, it is still necessary to open file BROWN, because the material on discs does not include the IBSYS service programs, including the IOCS; that is, QRAD and QWAD do not touch the basic input-output routines or the buffer areas.

The subroutines called in flow chart 15 are, apart from QRAD and QWAD, PREAD (flow chart 3), DECOM (flow charts 9, 10, 11), ATTL (flow chart 12), and LAB (flow chart 17). The exits are to VEND, when the input file is exhausted (not counting the file that is read between FIRST and FIRSF; what we call the input file is the one that follows that file, when the program begins at FIRST); to ERRO1 if the first character of a statement card is a punctuation

or the beginning of a literal (there will be an error message, and the card will be ignored by the compiler); to ERRO2 if the statement begins with a label, but this is followed by a punctuation or literal instead of a blank (there will be an error message in either case; in the first case the statement will be treated as if the punctuation were a blank, and in the second, as if the literal were separated from the label by a blank); to ERRO5 if the function code in the statement, or rather the first word of the function code, is not one of those in the table beginning at FUNT (i.e. if it is not a function code, in which case there is an error message and the statement is ignored by the compiler); to ERRO4 if the function is one shown in the table FUNT as demanding at least one number or symbol after it, and there is not one in the statement (error message, and the statement is ignored); and finally at VPH, there is an exit to one of the routines for compiling the different functions.

Here is a table of function codes, with the compiler program addresses where their routines begin, and the number of the flow chart for each.

ADD	ADD	39
AREA	AREA	33
BACKSPACE	BSP	45
BINARY CODE	BNR	76
BINARY DECODE		
CLOSE	CLOSE	117
COMP	COMP	40
COMPOSE	BUILD	34

COPY	COPY	118
DEFFLD	DEFF	110
END	MEND	20
ERASE	ERAS	45
GO TO	GO	107
IFSWITCH	IFSWIT	78
MOVE	MOVE	39
OPEN COPY	FOPEN	111
OPEN READ		
OPEN SAVE		
OPEN WRITE		
OUTPUT	PUT	120
PAGE	PAGE	26
PAUSE	PAUSE	121
PCBIN	PCB	99
PCD	PCD	97
PRINT	PRINT	34
RCBIN	RCB	98
RCD	RCD	96
READ	READ	118
RELOAD	RLOD	32
REPLACE SCAN BY	RPLC	31
RESETSW	RSTSW	22
RETURN	RTRN	104
REVSU	RVSU	21
REWIND	REW	95
SAVE	SAVE	122

SCAN	SCAN	30
SET	FSETC	102
SETSW	SETSW	22
SKIP	SKIP	94
SORT FIRST PASS	HA	134
SORT MERGE		
SORT FINAL MERGE PASS		
START SCAN LEFTWARD	START	29
START SCAN RIGHTWARD		
SUBTRACT	SUBT	39
TAPNUM	TAPN	27
UNCHECK	UNC	25
UNLOAD	UNL	95
UNSAVE	UNSAV	118
WREOF	WEFF	95
WRITE	WRITE	122
XEC	XEC	105
ZERSUP	ZERS	19
9CLOSE	NCL	117

Switch VLABF is one which may have been turned on by subroutine GNS (flow chart 77) during the compiling of the preceding statement, to show that if this statement does not have a label, a label must be supplied to it. The references to VLABF in flow chart 15 are the only ones in the compiler outside flow chart 77. VLABF is used indirectly, however; it is used to set switch LABF if necessary just before VPA in flow chart 15, and LABF is the switch that is

tested in subroutine LAB (flow chart 17) to see whether to supply a label to this statement. The reason for this is that in an IFSWITCH statement, the compiler may decide during the processing of the condition that the following statement will have to have a label. It may then return to VPA to process what follows the condition, and if it had set LABF to require a label for the next statement, it would cause subroutine LAB, called at the bottom right hand corner of flow chart 15, to give a label to this statement.

Flow chart 16 shows subroutine SPS, which takes characters chosen by other parts of the compiler and inserts them in an SPS card image in locations SCARD through SCARD&13. Whenever a word in the calling sequence for SPS has prefix PON or MON, SPS calls subroutine WRATE after processing that word; the SPS card image is written out on file WINTER, and the area SCARD through SCARD&13 is blanked again. But if the last word in the calling sequence for SPS has prefix PZE or MZE, something will be left behind in the SPS card image area. The next time subroutine SPS is called, it will not start to work on a fresh card image, but will continue a card image already begun.

This remark is meant to point out an inaccuracy in many flow charts. For instance, at the end of flow chart 19, for function ZERSUP, it says "Output SPS cards

MCS	PQR -001	XZSP
MCW	XZSP	PQR -001 "

but if subroutine LAB, called early in the processing of the statement and before flow chart 19 was reached, decided that the statement had or

needed a label, it will have left that label at the beginning of the SPS card image field. If the label was FROWN, for example, the routine in flow chart 19 would really: "Output SPS cards

```

                FROWN  MCS    PQR -001    XZSP
                    MCW    XZSP          PQR -001 "
```

Whenever a box in a flow chart says "Output SPS cards....." this is done in the compiler program through subroutine SPS, but subject to whatever may already be in the card image area. If something is already there, usually a label, it will be added to the first card described after "Output SPS cards".

Flow chart 17 shows subroutine LAB, which does the housekeeping for labels in compiler language statements and SPS-language instructions. LAB is called at two points in the program: in subroutine PREAD (flow chart 3) for dealing with SPS-language instructions coded by the programmer; and just before VPH, at the end of flow chart 15, for dealing with compiler language statements.

The meaning of switch LABF has just been explained in connection with flow chart 15. The other switches used in subroutine LAB are BCUT and SECND. Switch BCUT will have been set if and only if the preceding statement was one from which either control could not pass to the present statement, or could pass only by naming the label of the present statement. The only other reference to switch SECND is at IFU, near the upper right-hand corner of flow chart 80. It is set there to inhibit, in subroutine LAB, the action of switch BCUT for any statement that follows an

IFSWITCH statement.

E.g. after the statement GO TO MILANO switch BCUT will be set for the next statement, because no transition from the former to the latter is possible unless the latter happens to be the one whose label is MILANO. But during the compiling of the statement IFSWITCH B GO TO MILANO , switch SECND will be set during the processing of IFSWITCH B, and switch BCUT will be set during the processing of GO TO MILANO . SECND will cancel the effect of BCUT because a transition from IFSWITCH B GO TO MILANO to the next statement is possible, since it will occur whenever sense switch B is off.

BCUT and SECND control what is put into the symbol table, namely the value of Q in the call on subroutine ATTL. This is referred to above in the description of subroutine ATTL, flow chart 12. It has significance, finally, only for the flow chart of the compiled program, which the compiler automatically produces.

Subroutine LAB also calls on subroutine SCLAB (flow chart 13) to see whether the label of the current statement has previously appeared in the program. If it has appeared, then there is an exit from LAB to ERR12 (error message; current statement ignored by compiler) unless the description of the label in the symbol table is that of a label mentioned as a branch-to address, but whose statement has not yet appeared. If the label is in the symbol table with any other sort of description, its appearance as the label of the current statement is an attempt at giving it a second definition, and this is not allowed.

Subroutine LAB also calls on subroutine LABG (flow chart 18).

Flow chart 18 shows subroutine LABG, which is called only twice in the program, once by subroutine LAB (flow chart 17) and once by

subroutine TWIG (flow chart 106). LABG constructs a label for the current statement or SPS instruction, puts it into the card image of the SPS instruction itself, or of the first or only SPS instruction that will be deduced from the present statement, and insures that the label is in the symbol table.

Location TWIGY initially contains BCI 1,X11111 . There are only two places in the program where TWIGY is referenced: here in LABG, and in subroutine GNS (flow chart 77) which looks ahead to see what the label of the next card will be, and if it has none, to set switch VLABF so that the contents of TWIGY will be its label. Here in LABG is the only place where the contents of TWIGY are used up, and it is necessary to put a new value in TWIGY. ("Step up TWIGY".) This is done with subroutine JACK, which is not actually named on the flow chart for LABG. See flow chart 92 for JACK.

Then the contents of TWIGY are put in TWIGY&2, and PZE TWIGY&2 is put in DRAD, so that any routine which locates the label by looking at DRAD will find it. Then subroutine SCLAB (flow chart 13) is used to see if the label is already in the symbol table; if not, it is put in the table by subroutine ATTL (flow chart 12). If the label from TWIGY is already in the symbol table it will be as one that has been named in a previous statement as a branch-to address (named indirectly, under the guise of NXT or an implied NXT LABG does not bother to rectify this and put it in the table as the label of a statement that has appeared; the point is of no importance. Any label that begins with X Y or Z is assumed to be the result of the workings of the compiler, since the programmer is not supposed to use such labels, and thus mistakes cannot occur with them.

Flow chart 19 shows the routine for a ZERSUP statement. This uses subroutine SCLUB, for which see flow chart 24, to make sure that the first word after ZERSUP in the statement is the name of a variable, and to get its description from the symbol table. Exit 1 from SCLUB occurs if the word is not in the symbol table. The exits from flow chart 19 to ERR58, ERR59, VA, ERR60 should be self-explanatory. The three error exits cause an error message and the ignoring of the statement. If it turns out that the variable to be zero-suppressed is a single character, the statement is also ignored.

In the box headed "Output SPS cards", we have the first instance of an uneasy mixture of different categories of symbol. XZSP is what will actually appear on SPS cards as the B-address of the first and the A-address of the second; it is the name of a word in the 1401 program package included in the compiler. PQR, on the other hand, is a symbol arbitrarily used in this flow chart to represent the name of the variable, whatever it is. In effect, we say "Suppose the statement to be ZERSUP PQR."

Note also that the "Output SPS cards" box neglects the possibility that the first card may have a label, left there in the card image area by subroutine LAB. This is true of all "Output SPS cards" boxes in these flow charts.

Flow chart 20 shows the routine for dealing with an END statement. All preceding compiler language statements have already been translated into SPS instructions on file WINTER, and this END statement must now be translated into an SPS END card. (Note that an END card coded by the programmer in SPS format rather than

compiler language format will not work; i.e. the END card must not have a digit in its first column. But there would be no temptation to do this, as a compiler language END card works exactly like an SPS END card, and has a freer format.)

But first the SPS program must be completed with the input-output and other subroutines from the compiler package, as required by the statements that have been compiled previously. Switches, whose location symbols all begin with Q, have been set by the compiler when certain sorts of statements were compiled. The cards in the compiler's 1401 program package, or rather the card images, have codes in columns 40-45, and these codes, together with the Q-switch settings, determine which card images are copied onto file WINTER before the SPS END card. Here is a list of the Q-switches, with the occasion on which each of them can be set:

QRAN	any OPEN READ statement
QRST	any OPEN READ statement with ST blocking
QRRM	any OPEN READ statement with record-mark blocking
QRNO	any OPEN READ statement with normal blocking
QRPH	any OPEN READ statement with physical blocking
QRMU	any OPEN READ or OPEN COPY statement containing MULTI
QCAN	any OPEN COPY statement
QSAN	any OPEN SAVE statement
QSNO	any OPEN SAVE statement with normal blocking
QSST	any OPEN SAVE statement with ST blocking
QSRM	any OPEN SAVE statement with record-mark blocking
QWNO	any OPEN WRITE statement with normal blocking
QWST	any OPEN WRITE statement with ST blocking

QWRM any OPEN WRITE statement with record-mark blocking
 QWPH any OPEN WRITE statement with physical blocking
 QWAN any OPEN WRITE statement
 QCNO any OPEN COPY statement with normal blocking
 QCPH any OPEN COPY statement with physical blocking
 QCRM any OPEN COPY statement with record-mark blocking
 QCST any OPEN COPY statement with ST blocking
 QOP any OUTPUT statement writing from a save file;
 or any SORT FIRST PASS statement
 QOR any OUTPUT statement writing from a read or copy file
 QPRN any PRINT or COMPOSE statement
 Q9CL any 9CLOSE statement
 QBCD any BINARY CODE statement
 QBDC any BINARY DECODE statement
 QZSP any ZERSUP statement
 QSCAN any START SCAN LEFTWARD, START SCAN RIGHTWARD, SCAN, or
 REPLACE SCAN BY statement
 QFP any SORT FIRST PASS statement
 QME any SORT MERGE statement
 QFM any SORT FIRST PASS or SORT MERGE statement.

As set on these occasions, the switches do not correspond one-to-one with the codes in the card images. The correspondences are given in a table in flow chart 20. In order to make them work, the switches are logically combined in various ways until there is one switch corresponding to each card code. The table in the program listing from PARTS to PARTE gives the correspondences after this combining.

The program initializes subroutine DSKR and then uses it to read through the 1401 program package, from the beginning to the (EIOP) card. Each card is selected or not, according to what it has in cols. 40-45, and whether the corresponding Q-switch is set; and if selected it is processed by subroutine WRATE as if it had been part of the original input.

Beginning at MENDC in flow chart 20, the SPS "END" card is composed and output by subroutine SPS. If the fifth element (E.5) of the input statement is null, the compiler makes the A-address of the END instruction 0628, as this is the address of the first instruction into which the first imperative compiler-language statement was translated.

Flow chart 20 also calls on subroutine LABIG, for which see flow chart 103. It is used with all declarative statements to see if they have labels. If they do, an error message is given on each occasion and the label is ignored, but otherwise the statements are accepted.

The only exit from flow chart 20 is to VEND, on flow chart 124. The branch to VEND takes place when the input program has been completely translated into an SPS program, and the first pass of the SPS assembly has been done.

Flow chart 21 shows the compiling of a REVSW statement. Like flow chart 22, it calls on subroutine SUSW, shown in flow chart 23, to check on the switch name. If the switch name is that of a sense switch, or is "PAGEEND" referring to the sensing of a channel-12 punch in the printer carriage control tape, the

exit from flow chart 21 or 22 is to ERR38, where an error message is given and the statement is ignored. Otherwise the exit from both flow charts is to VA, as usual.

Flow chart 23 shows subroutine SUSW, used to check on switch names. It is called in flow charts 21 and 22, as mentioned above, and in flow chart 81 for subroutine IFSS, which is used in compiling IFSWITCH statements.

The calling sequence for SUSW will give an address A, normally one of the addresses DRAD to DRAD+70. The word at A must contain PZE B, where B is the location containing the switch name. If the word at A is negative or zero, the statement must have been wrongly coded, and there is an exit to ERR38: message, and statement ignored.

If the switch name is "XFIRST", there is an immediate exit 2 from the subroutine, as for a program switch. XFIRST is a program switch automatically provided by the input-output package, and does not have to be stored in the symbol table. It is considered to be set when either no READ, COPY or UNSAVE statement has been executed, or the most recently executed READ, COPY or UNSAVE statement was a READ or COPY statement which read the first or only record in its block.

If the switch name is "PAGEEND", this is replaced with the single character @ .

If the switch name is a single character, exit 1 from the subroutine is taken.

Otherwise, subroutine SCLAB (flow chart 13) is used to see if the name has already occurred in the program. If so, and it has occurred

as a switch name, exit 2 takes place. If it has occurred as some other sort of name, ERR39 occurs -- message and statement ignored. If the name has not already occurred, it is put in the symbol table by subroutine ATTL (see the explanation of flow chart 12) after a character has been reserved in the memory for the switch it names. Then exit 2 is taken.

This is the first instance we have mentioned so far in which the contents of location NAFL are used to find the address to assign to a constant or workspace in the 1401 memory. The initial value in NAFL is 7949, and this is counted downwards as the storage space is used up.

Flow chart 24 shows subroutine SCLUB, used in compiling statements with functions UNCHECK (flow chart 25), TAPNUM (flow chart 27), ZERSUP (flow chart 19), SKIP (flow chart 94), REWIND, UNLOAD, ERASE, BACKSPACE, and WREOF (flow chart 95), 9CLOSE and CLOSE (flow chart 117), READ, COPY and UNSAVE (flow chart 118), WRITE and SAVE (flow chart 122) and OUTPUT (flow chart 120). These are the functions whose codes are one word, and must always be followed in a statement by a blank and then the name of something that has been defined by a declarative statement: a variable for ZERSUP, and a file for the others. The fifth element in the statement, the word after the function code, is sought in the symbol table by subroutine SCLAB (flow chart 13). If it is not found, exit 1 from SCLUB is taken. If it is found, its description is put in the sense indicator register and in the MQ; the AC is cleared, and the first three bits of the description

are shifted into it. Exit 2 is taken, and the routine that called on SCLUB has the type number of the symbol immediately available in the AC.

Flow chart 25 shows the routine for compiling function UNCHECK. Subroutine SCLUB (flow chart 24) is used to check on the word after UNCHECK in the statement; if it is not in the symbol table, the program goes to ERR19. Then, if the symbol is not the name of a read or copy file, the program goes to ERR35. Otherwise, the appropriate SPS instruction is output, and the program goes to VA.

"PQR&012", when PQR is the name of a read or copy file, is the address of the last character before the first character of the buffer. The digital part of this character is blank for a read file, and contains the output tape number for a copy file. The zone bits are both 0 for a non-multi-reel file, and both 1 for a multi-reel file. The word mark is present if the file has never been read or copied, or if the last record copied or read did not belong to a redundant block, or if an UNCHECK statement has been applied to the file since the last time a block was read. Otherwise the word mark is absent.

Flow chart 26 shows the routine for compiling function PAGE. This is translated directly into an SPS instruction card with F in column 16 and @ in column 39.

Flow chart 27 shows the routine for compiling function TAPNUM. Subroutine SCLUB (flow chart 24) is used to see that the word after TAPNUM in the statement is in the symbol table; if not, ERR19. Then the program goes to ERR32 unless the word is the name of a read, write, or copy file. If it is a write file without blocking, or with physical blocking, the tape number is at the position of which the compiler language file name is the SPS address. Otherwise, the tape number is at the position 11 characters further right.

If the seventh element of the statement (the file name, its fifth element, should be followed by a comma and then a tape number) is not a number between 1 and 6, the program goes to ERRO7.

Flow chart 28 shows subroutine SETSC, which is used in the compilation of functions START SCAN LEFTWARD and START SCAN RIGHTWARD (flow chart 29), SCAN (flow chart 30), and REPLACE SCAN BY (flow chart 31). It makes sure that switch QSCAN in the compiler program has been set on (see flow chart 20 and its explanation) and that "SCAN" has been put into the symbol table as the name of a one-character variable located at 0618. Thus, the subroutine accomplishes nothing except on the first occasion of its use during the compilation of a program. If, on that occasion, it finds that "SCAN" is already in the symbol table, the program goes to ERR62, because the word has been improperly used.

Flow chart 29 shows the routine for compiling functions START SCAN LEFTWARD and START SCAN RIGHTWARD. It uses subroutine SETSC (flow chart 28) and subroutine TYPE (flow chart 73) to determine what kind of field has to be scanned, and so how to compile the statement.

The only special exits are:

- (a) to ERRO4 if the function code begins with "START" but is not properly completed.
- (b) to ERR47 if the statement calls for BLANKS or a literal to be scanned.

Flow chart 30 shows the routine for compiling a SCAN statement. This is quite straightforward; the only variable in the SPS output is the statement label that must follow "SCAN" in the statement, represented by "PQR" in the flow chart. This is the label of the statement to which the program must branch if the statement finds that the field has already been completely scanned.

The only subroutine called is SETSC (flow chart 28).

Flow chart 31 shows the routine for compiling function REPLACE SCAN BY . The only special exits are:

- (a) ERRO4 if the function code begins with "REPLACE" but is not properly completed.
- (b) ERR61 if the statement attempts to replace the scanned character with a record from some file.

The subroutines called are SETSC (flow chart 28) and TYPE (flow chart 73).

Flow chart 32 shows the routine for compiling function RELOAD. This is completely straightforward.

Flow chart 33 shows the routine for compiling the declarative function AREA. It calls subroutine DEV (flow chart 7) to get the binary equivalents of the numbers addressing the ends of the area; SCLAB (flow chart 13) to make sure the name of the area is not already in use as a symbol; ATTL (flow chart 12) to put the name of the area into the symbol table; and LABIG to give an error message if the statement has a label (but the statement will be accepted.)

The special exits are:

- (a) ERR23 if either of the words that should give the 1401 addresses of the ends of the area is not a number.
- (b) ERR21 if the right end of the area is supposed to have a lower address than the left end of the area.
- (c) ERR20 if the name given to the area has already been defined as the name of something else by an earlier statement.

Flow chart 34 shows the routine for compiling PRINT and COMPOSE statements.

Such a statement can be indefinitely long. If it is not "PRINT COMPOSITE", it consists of "PRINT" or "COMPOSE" followed by a blank and then an indefinite number of groups of elements of the form

number/word
number/literal

or

number/word.word

with the groups linked by commas between them.

When M is set = 5 in the flow chart, it means that "E.M" is E.5, the fifth element of the statement, and the number of the first group. Since a group may consist of three or five elements, and is followed by a comma unless it is the last group, M is increased by 4 or 6 as appropriate after each group has been processed.

The subroutines called are DEV (flow chart 7) to get the binary equivalent of the number of each group, E.M; TYPE (flow chart 73) to identify what is meant and named by the word(s) after the slash, E.(M&2) and possibly E.(M&4); PRINS (flow chart 36) and PRINH (flow chart 35) -- two subroutines used only for these functions, and called only in flow charts 34 and 37.

If a section names a file, flow chart 34 branches to PRINF, in flow chart 37, merely a continuation of flow chart 34. From the lower left-hand corner of flow chart 34 there is a branch to PRUNA, on flow chart 37.

The only special exits from flow chart 34 are to ERR46 and ERR51, if what should be the number of a group turns out not to be a number.

Flow charts 35 and 36 show subroutines PRINH and PRINS, which are called only in flow charts 34 and 37, and need no remark.

Flow chart 37 shows the continuation of flow chart 34, which is joined to it at PRINF and PRUNA, and from which the only exit is back to PRINE on flow chart 34. It calls on subroutines PRINH (flow chart 35) and LINC (flow chart 60).

Flow chart 38 shows subroutine COMQ, which is used in compiling ADD, SUBTRACT, and MOVE statements (flow chart 39) and COMP statements (flow chart 40). It calls subroutine TYPE (flow chart 73) to get information from the symbol tables about two data names in the statement.

The first data name begins with the word after the function code, which is the fifth element in the statement. If the sixth element (E.6) is a period, the fifth and seventh elements must be a file name and the name of a field in records of that file; so the second data name must begin with the ninth element; otherwise with the seventh element. This determines the calling sequence for the second use of TYPE: here A is DRAD&6 or DRAD&8. Similarly, the second data name may consist of one word, or of a word, a period, and a word, so that the first word after the second data name, if there is one, may be the ninth, eleventh, or thirteenth element. Accordingly, PZE DRAD&8/10/12 is stored in CMPB.

Subroutine TYPE furnishes a number for each of the types of data name it may be applied to (see column TYPQA in flow chart 75) and the numbers for the two data names in the statement are combined by COMQ and the complement put in index register 2, so that the routine that called on COMQ may use an address table immediately afterwards, and branch according to the types of the two data names.

Flow chart 39 shows the beginning of the routines for compiling ADD, SUBTRACT, and MOVE statements. Initially, the fundamental 1401 operation code is chosen and saved: A, S, or MCW. This is represented by III in flow charts 43, 45, 47, 51, 52, 55, 56, 57, 58, 66, 67, 68, 69 and 70 below. Then subroutine COMQ is called, to identify the data names. 7090 memory locations TYTAB to TYTAB&24 contain, in their decrement parts, the addresses of the various routines to which the program must now branch, according to the number left in index register 2 by COMQ. However, if the function code is ADD or SUBTRACT, the branch can only take place if the corresponding word in table TYTAB is negative; otherwise the program goes to ERR64, because an ADD or SUBTRACT statement may have only a literal or variable as its first data name, and only a variable as its second data name. In addition, the branch address taken from the table will be ERR47 if the statement attempts to move something into BLANKS or into a literal.

Apart from the errors, the branch out of flow chart 39 can be to MVPG (flow chart 65), MVPK (flow chart 43), MVTB (flow chart 51), MVPS (flow chart 70), MVPU (flow chart 66), MVTE (flow chart 67), MVPL (flow chart 69), MVPH (flow chart 45), MVTC (flow chart 47), MVPN (flow chart 69), MVTQ (flow chart 52), MVTD (flow chart 56), MVTG (flow chart 68), MVTL (flow chart 59), and MVTK (flow chart 55).

Flow chart 40 shows the beginning of the routines for compiling COMP statements. Subroutine COMQ (flow chart 38) is called, to identify the two data names and their types, and then, according to the number left in index register 2 by COMQ, the

address in one of the words in the table from TYTAB to TYTAB&24 is chosen to branch to. If the statement attempts to compare BLANKS or a literal with BLANKS or a literal, the branch will be to ERR44. Otherwise, the branch out of flow chart 40 is to one of
 CMPG (flow chart 61), CMPK (flow chart 42), CMTB (flow chart 50),
 CMPP (flow chart 61), CMPS (flow chart 70), CMPT (flow chart 66),
 CMPU (flow chart 66), CMTE (flow chart 67), CMPL (flow chart 66),
 CMPH (flow chart 44), CMTC (flow chart 53), CMPM (flow chart 42),
 CMPT (flow chart 66), CMPO (flow chart 48), CMTQ (flow chart 49),
 CMTD (flow chart 54), CMTF (flow chart 50), CMTG (flow chart 67),
 CMTL (flow chart 53), CMTL (flow chart 54), CMTK (flow chart 58).

Flow chart 41 shows subroutine CPREV, which is used in compiling some COMP statements to invert the order of the two data names, and interchange the high and low branch addresses. This is done because 1401 comparison requires a word mark to define the B-field, but not the A-field. It is more convenient, therefore, to have as the second data name in the statement a variable name, a literal, or BLANKS if possible.

Apart from the normal return, the program branches to ERRO4 if it finds that there are not enough elements in the statement to provide three branch addresses after the second data name.

CPREV is called in flow charts 42, 50, 53, 54, 61, 66 (twice), and 67.

Flow chart 42 shows the routine for compiling the comparison of a variable with BLANKS (CMPM) or BLANKS with a variable (CMPK). the only branch is to CMPY, where the branch-on-indicator instructions after the comparison are provided.

Flow chart 43 shows the routine for compiling a statement that moves BLANKS into a variable.

Flow chart 44 shows the routine for compiling a statement that compares a literal with a variable. AAAA will be the address of the rightmost character of the literal, which subroutine TYPE will have found in the symbol table and stored, and BBBB will be the symbolic name of the variable. The only branch is to CMPY, where the branch-on-indicator instructions after the comparison are provided.

Flow chart 45 shows the routine for compiling a statement that moves a literal into a variable, or adds a literal to a variable or subtracts a literal from a variable. The exit is to VAS, as from flow chart 52 also (these are the only two flow charts for ADD or SUBTRACT statements). At VAS, there is a test for whether the 1401 operation was subtraction, and if so, two more SPS instructions are constructed, to eliminate the possibility of any zoning in the result field except minus-zone and no-zone.

Flow chart 46 shows subroutines MCTF and MCTB, which are called only once each, in flow charts 51 and 50 respectively.

Given the length X of an area named in the statement, each of them produces the 4-digit decimal form of the address of the right-end character of a word of blanks of the same length. Its left-end address is 7950.

Flow chart 47 shows the routine for compiling a statement that moves one area into another. The 1401 op-code is shown as III, as though it could also be "A " or "S ". But a statement calling for addition or subtraction of two areas would already have been rejected at the bottom of flow chart 39.

Flow charts 48 and 49 show the routines for compiling statements that compare a variable with a literal, and two variables, respectively. Compare the note on flow chart 44.

Flow chart 50 shows the routine for compiling statements that compare BLANKS with an area (CMTB) or an area with blanks (CMTF). In the former case, subroutine CPREV (flow chart 41) is used to reverse the statement so that the second data name will be BLANKS, which already has a word mark. In either case, subroutine MCTB (flow chart 46) is used to get the right-end address of the necessary blank word.

Flow chart 51 shows the routine for compiling statements that move BLANKS into an area. It uses subroutine MCTF (flow chart 46)

to get HHHH, the right-end address of the necessary word of blanks. Though the 1401 op-code is represented by III, a statement that called for BLANKS to be added to or subtracted from something would have been rejected at the bottom of flow chart 39.

Flow chart 52 shows the routine for compiling statements that move one variable into another, or add or subtract two variables. See the note above on flow chart 45.

Flow chart 53 shows the routine for compiling statements that compare a literal with an area, or an area with a literal. In the former case, subroutine CPREV (flow chart 41) is used to reverse the comparison and ensure that the second data name indicates something with a word mark.

Flow chart 54 shows the routine for compiling a statement that compares a variable with an area, or an area with a variable. In the former case, subroutine CPREV (flow chart 41) is used to reverse the comparison.

Flow chart 55 shows the routine for compiling a statement that moves an area into a variable. Though the 1401 op-code is represented by III, a statement that attempted to add an area to a variable, or subtract an area from a variable, would have been rejected at the bottom of flow chart 39.

Flow chart 56 shows the converse of flow chart 55.

Flow chart 57 shows subroutine MCTK, which is called only in flow charts 58 and 59. Where the two data names of a COMPARE or MOVE statement both refer to areas, MCTK provides for the "move" or "compare" instruction itself, and also for putting a word mark on the second area before the action is taken, and removing it afterwards.

Flow chart 58 shows the routine for compiling a statement that compares two areas. It uses subroutine MCTK (flow chart 57).

Flow chart 59 shows the routine for compiling a statement that moves one area into another (a statement that attempted to add or subtract two areas would have been rejected at the bottom of flow chart 39). It uses subroutine MCTK (flow chart 57.)

Flow chart 60 shows subroutine LINC, which is called in flow charts 37, 61, 65, 66, 67, 68, 69 and 70. The calling sequence (which is illustrated just to the right of the flow chart) always names a 7090 memory location and a number; the 7090 location contains the address of the name of a file, and the number is used to construct three B-address adjusters in the three SPS instructions produced by the subroutine. It is used whenever the entire current record of a file has to be moved somewhere or compared with something. If the name of a file is, for example, SHOAL, the address of the first character of the record is in 1401 cells SHOAL-2, SHOAL-1, and SHOAL. The length of the record, in the form of a 3-character 1401 address, is in cells SHOAL&1, SHOAL&2, and SHOAL&3. "0602" is always the address of a constant "I9I" , or the 16000-complement of 1.

Flow chart 61 shows the routine for compiling a statement that compares BLANKS with part or all of a record, or the converse. In the former case, subroutine CPREV (flow chart 41) is used to reverse the comparison so that the second data name is BLANKS, which already has a word mark. If a whole record is indicated, subroutine LINC (flow chart 60) is used to get the SPS instructions that will set up the right-end address of the record. If a field of a record is indicated, subroutines CPSA (flow chart 62) and MCPG (flow chart 63) are called. CPSA provides an SPS instruction for putting the address of the leftmost character of the record into index register 1. MCPG constructs the 4-character decimal address BBBB of the rightmost character of the blank word with a length equal to that of the field.

Flow chart 62 shows subroutine CPSA, which is called in flow charts 61, 66, 67, and 70 (twice). It is used, when the first data name of a statement names a field of the current record of some file, to produce the SPS instruction that will put the address of the first character of the record in index register 1.

Flow chart 63 shows subroutine MCPG, which is called in flow charts 61 and 65. Given G, one less than the length of a field of a record, it finds BBBB, the four-digit decimal form of the address of a word of blanks with the same length. If G is above 49, there is an exit to ERR45, because BLANKS cannot represent a word of more than 50 blank characters.

Flow chart 64 shows subroutine CPSB, which is called in flow charts 65, 68, 69, and 70 (three times). It is used, when the second data name of a statement names a field of the current record of some file, to produce the SPS instruction that will put the address of the first character of the record into index register 2.

Flow chart 65 shows the routine for compiling a statement that moves BLANKS into a record or a field of a record. If a whole record, subroutine LINC (flow chart 60) is used to get the SPS instructions for constructing the address of the right end of the record. If a field of a record, subroutine CPSB (flow chart 64) is used to get the SPS instruction that will put the address of the left end of the record into index 2, and subroutine MCPG (flow chart 63) is used to get the address of the right end of the necessary blank word. RRRR, mentioned in the bottom box of flow chart 65, will have been constructed during the last use of subroutine TYPE, during the execution of subroutine COMQ in flow chart 39.

Flow chart 66 shows the routines for compiling statements that compare a literal with a record or field of a record (CMPL), compare a record or field of a record with a literal (CMPT), compare a variable with a record or field of a record (CMPN), compare a record or field of a record with a variable (CMPU), or move a record or field of a record into a variable (MVPU).

CMPL and CMPN differ from CMPT and CMPU respectively only in beginning with subroutine CPREV (flow chart 41) to reverse the comparison and make the second data name represent something that already has a word mark.

CMPT differs from CMPU only in that the former gets XXXX, the right-end address of the literal, from where it was left by subroutine TYPE during the execution of subroutine COMQ in flow chart 39; while the latter uses the symbolic name of the variable, found in the statement itself, as XXXX.

MVPU differs from CMPU only as to YYY, the 1401 op-code of the crucial SPS instruction (compare or move) and ZZZZ, the exit from flow chart 66 (VA after setting up instructions for a move, CMPY after setting up instructions for a comparison, so as to set up the subsequent branch-on-indicator instructions.) Though III is used in the second box below the heading "MVPU", this must be "MCW", for any attempt to compile a statement using a record in addition or subtraction would have been rejected at the bottom of flow chart 39.

If a whole record is involved, subroutine LINC (flow chart 60) is used for the instructions that will set up the address of its right end. If a field of a record is involved, subroutine CPSA (flow chart 62) is used to get the instruction that will put the address of the beginning of the record into index 1.

Flow chart 67 shows the routines for compiling statements that compare an area with a record or field of a record (CMTG), or compare a record or field of a record with an area (CMTE),

or move a record or field of a record into an area (MVTE). CMTG differs from CMTE only in beginning with subroutine CPREV (flow chart 41). MVTE differs from CMTE only as to XXX, the op-code of the crucial 1401 instruction 2 (move or compare respectively) and in the exit from flow chart 67 (VA after setting up instructions for a move, CMPY after setting up instructions for a comparison, so as to set up subsequent branch-on-indicator instructions.)

If a whole record is involved, subroutine LINC (flow chart 60) is used for the instructions that will set up the address of its right end. If a field of a record is involved, subroutine CPSA (flow chart 62) is used to get the instruction that will put the address of the beginning of the record into index 1.

Though "III" is used to represent the op-code under MVTE, this will have to be "MCW", as any attempt to compile a statement adding or subtracting a record and something else will have been rejected at the bottom of flow chart 39.

Flow chart 68 shows the routine for compiling statements that move an area into a record or field of a record. If a whole record, subroutine LINC (flow chart 60) is used for the instructions that will set up the address of its right end. If a field of a record, subroutine CPSB (flow chart 64) is used to get the instruction that will put the address of the beginning of the record in index 2.

Though III is used at one point to represent an operation code, this must be "MCW" for the same reason as described for flow chart 67.

Flow chart 69 shows the routines for compiling statements that move a variable (MVPN) or a literal (MVPL) into a record or a field of a record. The only difference between MVPN and MVPL is that for the former, XXXX is the symbolic name of the variable, which comes directly out of the statement; while for the latter, XXXX is the four-digit decimal address of the rightmost character in the literal, which was constructed during the use of subroutine COMQ in flow chart 39.

Subroutines LINC (flow chart 60) and CPSB (flow chart 64) are called as by flow chart 68. The same remark on "III" as in flow chart 68 applies here.

Flow chart 70 shows the routines for compiling statements that move a record or field of record into another record or field of record (MVPS) or compare a record or field of record with another record or field of record (CMPS). The only differences between MVPS and CMPS are in the op-code "XXX", which will be "MCW" for the former and "C " for the latter (though III is mentioned for MVPS, this must be "MCW" in fact, for any attempt at addition or subtraction involving a record would be rejected at the bottom of flow chart 39); and the exit YYYY from the flow chart, which is VA after setting up instructions for a move, and CMPY after setting up instructions for a comparison, so as to set up the subsequent branch-on-indicator instructions.

The possible assortments of first and second data name are field-field (go to CMPSC), record-field (go to CMPSB), field-record (go to the box just below CMPSC in the case of a move,

or invert the statement with CPREV and go to CMPSB in the case of a comparison), and record-record (go straight down the left side of the flow chart). This gives four possible situations for naming a whole record, with a call on subroutine LINC (flow chart 60) for each of them; and four possible situations for naming a field, with a call on CPSA (flow chart 62) for two of them, and a call on CPSB (flow chart 64) for the other two. There is an additional call on CPSB near the lower left-hand corner of the flow chart: when a record is moved into or compared with a second record, a word mark has to be put on the second record beforehand and removed afterward; it is addressed through index 2, and CPSB is used for the instruction that gets the address of the current record into the index register.

Flow chart 71 shows subroutine CMPYU, which is used at various points in flow chart 72, and nowhere else. The calling sequence, as shown in flow chart 71, gives A, which is usually an address in the series DRAD to DRAD&70, and locates indirectly a statement label or the word "NXT"; and X, which represents blank, S, T, U, or / i.e. one of the D-characters that are significant after a comparison, or blank for an unconditional branch.

Below flow chart 71 are listed the abbreviations of the various calls on CMPYU. Where the compiler program has the calling sequence:

TSX CMPYU,2

PZE 0,4

BCI 1,T

index register 4 contains the complement of the address of the word in the range DRAD to DRAD&70 which locates the first of the three branch addresses given in the statement. Then 0,4 will indirectly locate the high transfer address, 2,4 will indirectly locate the equal transfer address, and 4,4 will indirectly locate the low transfer address; these labels are symbolized THIGH, TEQ, and TLOW in the flow chart. Each one of them must be either the label of a statement in the program, or "NXT".

Flow chart 72 shows the last section of the routine for compiling a COMP statement. Branches to CMPY come from flow charts 42, 44, 48, 49, 50, 53, 54, 58, 61, 66, 67, and 70. The exit is always to VA, after the necessary branch-on-indicator instructions have been written out. The only subroutine called is CMPYU, Flow chart 71, which is indicated in flow chart 72 by abbreviations as explained just above.

Flow charts 73 and 74 show subroutine TYPE, which is used for getting from the symbol table the necessary information about a symbol in a statement being compiled, or doing the house-keeping for a literal. "Flow chart 75" is a table of what is stored in TYPQA, TYPQB, TYPQC and TYPQD by the subroutine, as a function of the kind of symbol located in the calling sequence. Where a square in the table is blank, the compiler will not look in that location after it has identified the type of the symbol from the contents of TYPQA.

Note that at the beginning of flow chart 73, before subroutine TYPE does anything else, it automatically stores the contents

of TYPQA through TYPQD in TYPRA through TYPRD. Thus the main program can get from TYPQA through TYPQD the information about the symbol to which TYPE was most recently applied, and from TYPRA through TYPRD THE information about the symbol to which the subroutine was applied on the preceding occasion. Use of this is made in subroutine COMQ (flow chart 38) where TYPE is applied to the two data names in a COMP, MOVE, ADD, or SUBTRACT statement, and leaves the description of the first data name in TYPRA through TYPRD, and of the second data name in TYPQA through TYPQD.

Subroutine TYPE is called in compiling START SCAN LEFTWARD and START SCAN RIGHTWARD statements (flow chart 29), REPLACE SCAN BY statements (flow chart 31), PRINT and COMPOSE statements (flow chart 34), and COMP, MOVE, ADD and SUBTRACT statements (in subroutine COMQ, flow chart 38).

The junction between flow charts 73 and 74 is at TYPK, which is near the centre of flow chart 73, and at the top left-hand corner of flow chart 74. Flow chart 74 returns to flow chart 73 at TYPX, which occurs twice at the bottom of 74, and is at the bottom left corner of flow chart 73.

Apart from the normal return, the subroutine may branch to ERR28 if the element being looked up is a punctuation or blank instead of a word or literal; to ERR31 if it is a word not in the symbol table; to ERR41 if it is a file name followed by a period and what should be a field name, but was not mentioned in the DEFFLD statement for the file; to ERR29 if it is the name of a write file; and to ERR30 if it is a statement label or switch name.

Subroutines called in flow charts 73 and 74 are LIT (flow chart 108) for setting up a new literal, or getting the address and length of an old one; and SCLAB for seeking in the symbol table the word to which TYPE is being applied.

Let us explain a little further the table in "Flow chart 75". Where a square contains a single digit, say "2", this means that the word of 7090 memory contains it as a positive binary integer, "PZE 2" in the example.

PZE RB,,LG means that the word contains, in its address part, a binary number equal to the distance between the beginning of the record and the end of the field, and in its decrement part a binary number equal to the length of the field, minus 1.

LDLD means that the word contains, in bits S,1-23, a four-character decimal number equal to (a) in the case of a field, the distance between the leftmost character of the field and the first character of the record, or (b) in the case of an area, the address of the leftmost character of the area.

RDRD is the same as LDLD, but substituting "rightmost" for "leftmost" in the explanation given in the preceding paragraph.

RDRD-LGG means that the word contains, in bits S,1-23, a four-character decimal number equal to the address of the rightmost character of a literal or variable, and in bits 27-35, a binary number equal to the length of the literal or variable.

Flow chart 76 shows the routine for compiling BINARY CODE and BINARY DECODE statements. These simply become branches to the corresponding subroutines in the 1401 subroutine package. Besides the normal return to VA, there is an exit to ERR05 if the word after BINARY is not CODE or DECODE.

Flow chart 77 shows subroutine GNS, which is used whenever, in compiling one statement, it is necessary to know the label of the next statement. The next statement has to be examined, and if it does not have a label, a label in the series beginning with X11111 must be assigned to it. In that case, switch VLABF is set, so that when the next statement comes to be compiled, and subroutine LAB is applied to it, the X11111-label will be added to it.

GNS is called in the routines for compiling IFSWITCH statements (flow chart 78) and XEC statements (flow chart 105).

No other subroutines are called by GNS.

Besides the normal return, the only exit is to ERR48. This occurs if the next nine cards after the current statement are all comment cards. The tenth card after the current statement is not available yet. If a tape mark or binary record is reached before a statement, in the look-ahead, the procedure followed is as if a statement was found with no label. A label in the series X11111 ff. is assigned to the tape mark or binary record and used in one or more of the instructions compiled from the present statement. As this label is finally left undefined, the error is caught during the second pass of the SPS assembly.

Flow chart 78 shows the beginning of the routine for compiling IFSWITCH statements. It goes as far as IFW, at which point the first section of the condition has been delimited, and it is known whether the statement ends in a "GO TO ..." or in some other sort of statement.

Subroutine GNS is used to get the label of the next statement. Whether this is a label provided by the programmer, or one provided by the compiler from the series X11111 ff., we symbolize it "X111" for convenience in flow charts 78 through 91.

If the IFSWITCH statement ends with a GO TO statement, subroutine TWIG (flow chart 106) is used to record the branch involved, for use in making the automatic flow chart.

Besides the continuation to IFW, which will be found at the beginning of flow chart 79, there is an exit to ERRO4 if the element after IFSWITCH is not a word or a plus or minus sign, and an exit to ERR46 if the list of switches is followed by "GO" without a "TO" after it, or by "GO TO" without a word afterwards.

Flow chart 79 shows the next section of the routine for compiling IFSWITCH statements. It is entered only at IFW, at the top left-hand corner, to which there is a branch in flow chart 78 and two in flow chart 79 itself. The normal exits are to VA in flow chart 15, and to IFEL, IFT and IFU in flow chart 80. There is also an exit to ERR46 if the compiler expects a certain element to be a switch name and finds instead a punctuation or blank.

The function code IFSWITCH is followed by a blank, and then a series of switch names joined by plus signs, minus signs, and slashes. For each switch name, one or two SPS-language instructions are output. Each SPS-language instruction is output by one of the subroutines IFSA, IFSB, IFSC, IFSD, IFSE, IFSF, IFSG, IFSH and IFSK (flow charts 83 to 91 respectively.) The points in the compiler program at which this is done are as follows:

IPML: Positive switch, Medial (i.e. not last, therefore initial or medial really), in Last section of condition.

IPMM: Positive switch, Medial, in Medial (i.e. here again initial or medial is meant) section of condition.

IPLLF: Positive switch, Last in section, section is Last in condition, final statement involves a Function not "GOTO".

IPLLG: Positive switch, Last in Last section, final statement is "GO TO".

IPLMF: Positive switch, Last in Medial section, final statement involves a function not "GO TO".

IPLMG: Positive switch, Last in Medial section, final statement is "GO TO".

INML and INMM in flow chart 79, and INLLF, INLLG, INLMF and INLMG in flow chart 80, are the same as IPML, IPMM, IPLLF, IPLLG, IPLMF and IPLMG respectively, but with negative switches.

At each of these points, if one SPS instruction is to be output, the name of the subroutine and an indication of the instruction are given together within a frame of equal signs. If two SPS instructions, the two subroutines are named successively within frames of equal signs, and then the indications of the instructions are given afterwards within a single frame of minus signs.

In indicating the instructions in this way, the following symbols are used:

X is the switch name, which may be either a sense switch or PAGEND, or a program switch.

B is the 1401 op-code, which means "B" in an unconditional branch or in a conditional branch when X is a sense switch or the @ into which "PAGEND" is translated. But if X is a program switch, B symbolizes "BWZ", and X symbolizes the switch name as B-address, with a 1 as D-character.

In other words, B *005 X means

B *005	X	if X is a sense switch name, and
BWZ *005	X	1 if X is the name of a program switch.

X111 is the label which subroutine GNS, called at the beginning of flow chart 78, found in or assigned to the next statement.

Y111 is the label, in the series Y11111 ff., which will be given, at IFT in flow chart 80, to the first SPS instruction resulting from the first switch name in the next section of the condition.

Z111 is the label, in the series Z11111 ff., which will be given, at IFU in flow chart 80, to the first SPS instruction resulting from what follows the condition, as long as it is not a GO TO statement.

GOTO is the label which follows the words "GO TO" if they occur at the end of the current IFSWITCH statement.

From flow chart 79 there are normal branches to VA, if the compilation of the statement has been completed; to IFEL in flow chart 80, which is simply a continuation of flow

chart 79 at this point; to IFT in flow chart 80, when the last switch name in a non-last section of the condition has been processed; to IFU in flow chart 80, when the last switch name of the last section has been processed, and the IFSWITCH statement does not end in GO TO something; and to IFW, back at the beginning of flow chart 79, when a switch name that is not the last in its section has been processed.

Flow chart 80 is a continuation of flow chart 79, entered at IFEL, IFT, or IFU. The exits are to VA if the statement has been completely compiled; to IFV in flow chart 78 (the third box from the top in the second column from the right) when a new section of switch names has to be begun; and to VPA in flow chart 15 when all the conditions in the statement have been compiled, and the functional statement at the end must now be compiled.

The only thing to be noted in flow chart 80, not covered in the description of flow chart 79, is the calling of subroutine JACK (flow chart 92). When a label in the series Y11111 ff. is given to the first SPS instruction resulting from a non-first section of switch names, JACK is used to make ready the next label in this series. Similarly, when a label in the series Z11111 ff. is given to the first SPS instruction resulting from the functional part of an IFSWITCH statement, JACK is used to make ready the next label in this series.

Flow chart 81 shows subroutine IFSS, which is called in subroutines IFSA, IFSB, IFSD, IFSF and IFSG (flow charts 83, 84, 86, 88, and 89 respectively). It applies subroutine SUSW (flow chart 23) to the switch name at present being processed, and gives return 1 if this is a single character or PAGEND, and return 2 otherwise. In the former case, in flow charts 83 to 89, the switch name is represented by "X", and in the latter case by "SWIX".

Flow chart 82 shows subroutine IFSSS, which is called in the same flow charts as subroutine IFSS. Whenever a switch name is found to be a single character, IFSSS is used to make sure that it is a letter between A and G, or @ representing "PAGEND". If not, there is a branch to ERR49. If the switch name is acceptable, the normal return takes place.

Flow charts 83 to 91, showing subroutines IFSA to IFSK, should be more or less self-explanatory. Each one outputs one SPS instruction, during the compilation of an IFSWITCH statement. They are called only in flow charts 79 and 80.

"B" and "BWZ" symbolize themselves as 1401 op-codes. "X" symbolizes a sense switch name or @ replacing "PAGEND". "SWIX" symbolizes a program switch name.

X111 symbolizes the label which subroutine GNS, called at the beginning of flow chart 78, found in or supplied to the next statement.

Y111 symbolizes the label, in the series Y11111 ff., which

will be given, at IFT in flow chart 80, to the first SPS instruction resulting from the first switch name in the next section of the condition.

Z111 symbolizes the label, in the series Z11111 ff., which will be given, at IFU in flow chart 80, to the first SPS instruction resulting from what follows the condition, as long as it is not a GO TO statement.

GOTO symbolizes the label which follows the words "GO TO" if they occur at the end of the current IFSWITCH statement.

Flow chart 92 shows subroutine JACK, which advances the series of labels beginning at X11111 , Y11111 , and Z11111. The original intention was that the labels in these series should consist of X, Y, or Z followed by a five-digit octal number. But the occurrence of zeroes would have complicated the situation when such a label came to be moved into the SPS instruction field by subroutine SPS. So instead of using octal numbers with digits between 0 and 7, "octal numbers" with digits between 1 and 8 are used.

JACK is called for the Y- and Z-series in flow chart 80, and for the X-series in flow chart 18, subroutine LABG.

Flow chart 93 shows subroutine SPDA. The calling sequence supplies A, an address usually in the range DRAD to DRAD&70. This indirectly locates a symbol, and the subroutine outputs an SPS card, via subroutine SPS, that will eventually put the three-digit address of the first

instruction generated by that statement into the next three available positions of program storage.

SPDA is called in flow chart 94 (compiling SKIP statements), flow chart 105 (compiling XEC statements), flow chart 117 (compiling CLOSE and 9CLOSE statements), flow chart 119 (subroutine RWA, called in flow charts 118, 120, 122), flow chart 118 (READ, COPY and UNSAVE statements), flow chart 120 (OUTPUT statements), and flow chart 122 (SAVE and WRITE statements.)

Flow chart 94 shows the routine for compiling SKIP statements. It uses subroutine SCLUB (flow chart 24) to look up the name of the file to be skipped in the symbol table; and subroutine SPDA (flow chart 93) to output part of the calling sequence in the 1401 SPS version of the program.

Besides the normal exit to VA, there is a branch to ERR19 if the file named in the statement has not been defined, and a branch to ERR24 if what should have been the name of a read, copy, or save file has been defined as something else.

Flow chart 95 shows the routine for compiling REWIND, BACKSPACE, UNLOAD, WREOF, and ERASE statements. All such statements are translated into two SPS instructions, the first of which moves the tape number from a file's housekeeping section into the second instruction, and the second instruction performs the non-data function on the tape. The only difference among the five functions is in the D-character of the second SPS instruction; so this is stored before anything else is done, and then the routine is common to all functions.

If the file is a write file with no blocking, otherwise known as physical blocking, the tape number will be in the character named, in the SPS version of the program, by the name of the file in the compiler language version. For any other sort of read, write, or copy file, the tape number will be at that address plus 11.

The only subroutine used is SCLUB (flow chart 24), to look up the file name in the symbol table.

Besides the normal exit to VA, there is an exit to ERR19 if SCLUB shows that the file name has not been defined; to ERR32 if what should be the name of a read, write or copy file has been defined as something else; or to ERR27 if the statement attempts to erase or write a tape mark on a tape associated with a read or copy file.

Flow charts 96, 97, 98, and 99 show the routines for compiling RCD, PCD, RCBIN and PCBIN statements.

Flow chart 100 shows subroutine LIK, which is used to output the SPS card or cards that set up a literal or variable in the 1401 memory. This is a little complicated because there is no special limit to the length of such a word in the compiler language, but in SPS coding there is a limit -- here set arbitrarily at 30 characters -- so that for a longer word it is necessary to code the leftmost piece with a DCW card and the rest with one or more DC cards.

Subroutine LIKR (flow chart 101) is used for the outputting of each DC or DCW card.

When LIK is entered, the accumulator contains PZE A, where A is the binary form of the address which the rightmost character is to have. The MQ-register contains PZE B,,C where C is the number of characters in the word, and B is the address of the 7090 cell in which it begins.

LIK is used in compiling SET statements (flow chart 102) and by subroutine LIT (flow chart 108), which is itself used by subroutine TYPE (flow chart 73) and in compiling WRITE and SAVE statements (flow chart 122).

Flow chart 101 shows subroutine LIKR, which is called only by subroutine LIK (flow chart 100). It does most of the work of outputting an SPS card containing all or part of a constant.

Flow chart 102 shows the routine for compiling SET statements. If DRAD&6 contains -1 , because the seventh element of the statement was (, the statement is to define a word of blanks, whose length is given by a number between parentheses. In SPS language, this is provided for by a DC card giving the name of the variable in compiler language to the rightmost character of the word in SPS language, and a DCW card which provides a word mark on the leftmost character of the variable. This happens after the branch in the middle of the flow chart labelled "WORD OF BLANKS".

Otherwise, the statement defines the initial value of the variable by means of something written as a literal (the branch in the middle of the flow chart labelled "LITERAL"). One or more SPS cards, with op-code DCW or DCW and DC, are output by subroutine LIK (flow chart 100).

Other subroutines used are:

LABIG (flow chart 103) to give an error message if the SET statement begins with a label, but to allow the rest of the statement to be accepted.

DEV (flow chart 7) to get the binary form of the number of blanks.

SCLAB (flow chart 13) to see if the name which is to be given to this variable has already been used for something else.

ATTL (flow chart 12) to put the name of this variable into the symbol table.

Besides the normal exit to VA, there are exits to:

ERR36 if the seventh element in the statement is a word. The fifth element is the symbol of the variable, the sixth is assumed to be an equal sign, and the seventh must be either a literal or the opening parenthesis of a pair enclosing a number giving the length of an initially blank variable.

ERR32 if the name of the variable has already been used.

ERR04 if the word between the parentheses is not a number.

ERR04 (from SETCD) if what should have been the opening parenthesis turns out to be some other punctuation or a blank.

Flow chart 103 shows subroutine LABIG, which is called by the routines for compiling SET statements (flow chart 102), END statements (flow chart 20), AREA statements (flow chart 33), DEFFLD statements (flow chart 110), and OPEN READ, OPEN WRITE, OPEN COPY and OPEN SAVE statements (flow chart 111). If any of these declarative statements has a label, an error message is given and the label is completely ignored, but the statement is otherwise accepted.

Flow chart 104 shows the routine for compiling RETURN statements. Switch QSUB is set so that when the 1401 subroutine package comes to be read (see flow chart 20), cards labelled SU will be adopted for the program. Switch BCUT is set for a reason that has already been explained in connection with flow chart 12.

The only error exit is to ERR40, if the fifth element in the statement, what follows "RETURN" and a blank, is not a one-digit number.

Flow chart 105 shows the routine for compiling XEC statements. At the very beginning, switch BCUT is set, on the assumption that control cannot pass to the next statement after the subroutine has been executed. The switch will be reset at XECH, near the lower right hand corner of the flow chart, if "NXT" is found as one of the labels following the function code. The eventual significance of switch BCUT is explained in connection with flow chart 12.

The third element in the statement is "XEC", the fourth is assumed to be a blank, and the fifth is the label of the statement to which control is to pass -- the name of the subroutine as it were. If there is no sixth element, the compiler alters DRAD&6, so that the statement will appear to have "NXT" as a seventh element. If there is a sixth element and it is not a comma, there is a branch to ERRO4.

Switch QSUB is set for the same reason as above, in connection with flow chart 104.

Subroutine CHAD is used (flow chart 109) to put the fifth element of the statement into the symbol table as the label of an unmet statement, if it is not already there as the label of an encountered or unmet statement. Subroutine TWIG (flow chart 106) is not used on this element because it is not desired that the automatic flow chart should show branches into subroutines. The automatic flow chart will show, for an XEC statement, where the program may branch to the next time a RETURN statement brings control back from the subroutine.

Subroutine CHAD, rather than subroutine TWIG, is also used at the bottom right hand corner of the flow chart when dealing with "NXT" as an element in the statement. The fact that control may go to the next statement after the subroutine is executed will be shown, in the automatic flow chart, by an ordinary vertical, and not by an explicit branch line. The ordinary vertical will be produced because switch BCUT is reset if any "NXT" has been processed in this way. Note that if switch BCUT has already been reset, the question at XECH prevents the subroutine GNS (flow chart 77) from being used

more than once per compilation of an XEC statement. Otherwise, "NXT" might not represent the same thing each time it appeared in the statement.

A variable address "M" is used for all the statement labels following the name of the subroutine, in an XEC statement. Every time the word located by M is not "NXT", subroutine TWIG is applied to it so that the automatic flow chart will show a branch from the XEC statement to the statement named by the element.

Subroutine SPDA (flow chart 93) is applied to every word in the statement after the function code, to provide a DSA card in the SPS version of the program. If the word is "NXT", SPDA is applied not to NXT itself, but to the coded or supplied label of the next statement.

Subroutine GNS (flow chart 77) is called the first time "NXT" is encountered in the statement, to find out what explicit label can be equated with it.

Flow chart 106 shows subroutine TWIG, which is used to construct a list of possible branches that will be used in making up the automatic flow chart. The list begins at location TWIT, which initially contains zero, and is similar in some respects to the list of symbols beginning at location SNAL, described in connection with flow chart 12. However, the list that begins at TWIT is more complicated. From TWIT extends a list of all the statement labels to which control could branch, and from each item on that list extends a list of all the statement labels from which control could branch to it.

If the list has to show that control could branch to MILANO from LONDON, LEEDS, and WICK; to ROMA from LONDON and THURSO; and to TORINO from BALHAM; it will be set up as follows, using the colon to mean "contains":

TWIT:	PZE A
WKLIM-A:	PZE B,,C
WKLIM-B:	PZE D
WKLIM-B&1:	BCI 1,MILANO
WKLIM-D:	PZE E
WKLIM-D&1:	BCI 1,LONDON
WKLIM-E:	PZE F
WKLIM-E&1:	BCI 1,LEEDSO
WKLIM-F:	PZE
WKLIM-F&1:	BCI 1,WICKOO
WKLIM-C:	PZE G,,H
WKLIM-G:	PZE K
WKLIM-G&1:	BCI 1,ROMA00
WKLIM-K:	PZE L
WKLIM-K&1:	BCI 1,LONDON
WKLIM-L:	PZE
WKLIM-L&1:	BCI 1,THURSO
WKLIM-H:	PZE M
WKLIM-M:	PZE N
WKLIM-M&1:	BCI 1,TORINO
WKLIM-N:	PZE
WKLIM-N&1:	BCI 1,BALHAM

This list is used later between locations VEND&2 and LVWRX in the compiler program, as mentioned in flow chart 124. The manner of its use will be described to some extent in the discussion of that flow chart.

As well as making the list that begins at TWIT, subroutine TWIG calls on subroutine CHAD to put the label of the statement to which control may branch from the current statement, into the symbol table. See flow chart 109 for CHAD.

According to the flow chart, subroutine LABG (flow chart 18) is called if the current statement has no label. As far as the author can see, this is impossible. TWIG is called in the compilation of COMP statements (see subroutine CMPYU, flow chart 71), IFSWITCH statements (flow chart 78), XEC statements (flow chart 105), GO TO statements (flow chart 107), READ, COPY and UNSAVE statements (flow chart 118), SAVE statements (flow chart 122), and PAUSE statements (flow chart 121). For all these kinds of statements, at the third box from the bottom of the right-hand column of flow chart 15, switch LABF will have been set. Before the branch to the function routine at the end of that flow chart, subroutine LAB is executed. In flow chart 17, for subroutine LAB, it is seen that if the statement does not already have a label, subroutine LABG (flow chart 18) will be executed, and the statement will thereby get a label.

Therefore (returning to flow chart 106), it appears that the call on subroutine LABG, and the question that can branch to it, are unnecessary.

Apart from the normal return, the only exit from TWIG is to SYSER. This could happen if the workspace between locations WORK and WKLIM is exhausted. See the discussion in connection with flow chart 12.

Flow chart 107 shows the routine for compiling GO TO statements. Subroutine TWIG (flow chart 106) is used to record the branch for the later production of the automatic flow chart. Switch BCUT is set to prevent a vertical line in the automatic flow chart between this statement and the next one. The only special exit is to ERRO4 if the word after GO is not TO.

Flow chart 108 shows subroutine LIT. This is used by subroutine TYPE (flow chart 73) and in compiling WRITE and SAVE statements (flow chart 122), to deal with literals.

The subroutine consults a list beginning at location FLIT to see whether a literal identical with the current one has already been dealt with. If so, the address of the literal in the 1401 program in the form of four decimal characters, is taken out of the list and left in location LITC. If the literal is a new one, it is put into the list, while the word containing its address is also put in LITC, and SPS cards for the constant involved are output by means of subroutine LIK (flow chart 100).

Initially, location FLIT contains zero. The first time a literal is put into the list, PZE N,,N is put into FLIT, where N is taken from the address of location NWL, and shows that WKLIM-N is the next free location in the list workspace. The literal is put into

as many words as necessary, beginning at WKLIM-N&2. Into WKLIM-N is put PZE N-1, and into WKLIM-N&1 is put octal AABBCDDOEEE, where AABBCDD is the four-digit address of the rightmost character of the constant to which the literal refers, and EEE is the length, in binary, of the constant in characters. Supposing the first literal encountered to be \$VENEZIA\$, the situation is then (using colon for "contains"):

```
FLIT:      PZE N,,N
WKLIM-N:    PZE N-1
WKLIM-N&1:  OCT AABBCDD0007
WKLIM-N&2:  BCI 1,VENEZI
WKLIM-N&3:  BCI 1,A00000
```

Now supposing the next literal encountered is \$TRIESTE\$, and that when it is encountered location NWL contains PZE Q . The list becomes:

```
FLIT:      PZE N,,Q
WKLIM-N:    PZE N-1,,Q
WKLIM-N&1:  OCT AABBCDD0007
WKLIM-N&2:  BCI 1,VENEZI
WKLIM-N&3:  BCI 1,A00000
WKLIM-Q:    PZE Q-1
WKLIM-Q&1:  OCT EEFFGGHH0007
WKLIM-Q&2:  BCI 1,TRIEST
WKLIM-Q&3:  BCI 1,E00000
```

The only subroutine called by LIT is LIK (flow chart 100). The only exit, apart from the normal return, is to SYSER if the workspace between WORK and WKLIM is exhausted. See the discussion of this in connection with flow chart 12.

Flow chart 109 shows subroutine CHAD, which takes every statement label to which a possible branch is indicated and makes sure that it is in the symbol table. If the label is already in the symbol table but not as a statement label, there is a branch to ERR34. If the label is already in the symbol table as a statement label, whether its statement has been encountered so far or not, there is a normal return. If the label is not yet in the symbol table, it is put into it as the label of a statement that must be encountered later in the program.

The subroutines called are SCLAB (flow chart 13) to search the symbol table for the label, and ATTL (flow chart 12) to insert the label in the symbol table if it is not already there.

The only other exit is ERR04, if the element in the statement that is supposed to be the label of some other statement is actually a literal or punctuation, or if the statement is too short.

CHAD is called by subroutine TWIG (flow chart 106) and by the routine for compiling XEC statements (flow chart 105). The only reason for having CHAD as a separate subroutine (instead of including what it does in subroutine TWIG) is the necessity, while compiling an XEC statement, of getting the name of the subroutine and possibly the label of the next statement into the symbol table without recording branches to them.

Flow chart 110 shows the routine for compiling DEFFLD statements. The name of the file must already have been put into the symbol table (see flow chart 111 for the compilation of OPEN READ, OPEN

SAVE, and OPEN COPY statements), and there will be a word in the symbol table whose leftmost three bits indicate the type of file (000 for read, 010 for copy, 111 for save), whose next two bits to the right indicate the kind of blocking (00 for normal, 01 for standard, 10 for record-mark, 11 for physical), and whose next four bits to the right give the number of the tape, if any. The remaining 27 bits on the right are all 0. If they are not all 0, a DEFFLD statement has already been compiled for the file, and the current statement will be ignored.

The information about the fields will be stored in a continuous series of words in the workspace between WORK and WKLIM. Suppose the series of words to be in locations X through Y. Then PZE Y&1,,Y-X&1 will be or'd into the word with the 27 waiting zero bits on the right.

Between locations X and Y, each field will be defined by a pair of words of which the first is the name of the field, and the second is of the form PZE U,,V where U is the distance between the first character of the record and the rightmost character of the field, and V is the distance between the leftmost and rightmost characters of the field.

The subroutines called are LABIG (flow chart 103) to note the superfluity of a label; SCLAB (flow chart 13) to look up the file name in the symbol table; and DEV (flow chart 7) to convert to binary the character numbers of the two ends of each field, as coded in decimal by the programmer.

Besides the normal return to VA, there are exits to ERRO4 if the statement is too short to contain even one field definition;

ERR19 if the file name has not been defined by an OPEN statement; ERR57 if the file name is the name of a write file, or of something other than a file; ERR20 if a DEFFLD statement for the file has already occurred; ERR06 if what should be the character number of one of the ends of a field proves not to be a number; ERR22 if the character number for either end of a field is 0; and ERR21 if the character number for the left end of a field is higher than the character number for the right end. (Note that the word "offset" is used in the flow chart in what is probably an unclear way; if the leftmost character of a certain field is to be the third character of a record, the "left-end offset" would be 3 according to the flow chart. Probably the meaning of the word "offset" would be better satisfied if the offset were considered as 2 in such a case.) There is also an exit to SYSER if the workspace between WORK and WKLIM is exhausted. See the discussion of this possibility in connection with flow chart 12.

Flow chart 111 shows the beginning of the routine for compiling OPEN READ, OPEN COPY, OPEN WRITE, and OPEN SAVE statements. The subroutines called are LABIG (flow chart 103) for disregarding labels in such statements, and OPM (flow chart 112) for determining whether the statement ends with "MULTI" or not, and accordingly putting a plus in location MULTI and setting switch QRMU, or leaving zero in location MULTI and not setting the switch.

At the first decision point in flow chart 111, according to

the word following "OPEN" in the statement, index 2 is set to the complement of the address of QSAN, QCAN, QWAN, or QRAN. Each of these locations begins a series of 10 locations, one series for each class of file (save, copy, write, or read), with the following uses:

- 0,2 Switch location showing whether any file of the class has been opened.
- 1,2 Location containing the address to which the compiler will branch, according to the class of the file, when it leaves flow chart 111 (see the box in the lower right hand corner).
- 2,2 Switch showing whether any file of the class, with normal blocking, has been opened.
- 3,2 Switch showing whether any file of the class, with standard blocking, has been opened.
- 4,2 Switch showing whether any file of the class, with record-mark blocking, has been opened.
- 5,2 Switch showing whether any file of the class, with physical blocking, has been opened.
- 6,2 Location containing the prefix for the class of file, for use in the descriptive word that goes along with the name of the file into the symbol table (see the box in the upper right-hand corner of flow chart 114, and the box immediately to its left.)
- 7,2 Location containing the address to which the compiler is to branch after putting the file name and description into the symbol table (see the second box from the top in the right-most column of flow chart 114.)

- 8,2 Location containing the address to which the compiler is to branch at the end of flow chart 114 (see the lower right-hand corner of flow chart 114, and the beginning of flow chart 115.)
- 9,2 Location containing the address to which the compiler is to branch after OPWPH, when the file has been found to have physical blocking. (See the decision box near the center of flow chart 113.)

The normal exits from flow chart 111 are to OPWD and OPSD in flow chart 113. The other possible branches are to ERRO5 if the word "OPEN" is not followed in the statement by "READ", "COPY", "WRITE", or "SAVE"; and ERR14 if an OPEN COPY statement has something other than a digit between 1 and 6 where the number of the output tape should be.

Flow chart 112 shows subroutine OPM, which is called only in flow chart 111. It tests whether an OPEN READ or OPEN COPY statement ends with "MULTI", and if so sets switch QRMU and puts a plus instead of a zero at the right end of location MULTI. The normal return is the only one possible. If the statement has an element in the position where "MULTI" is allowed, but this element is not the word "MULTI", there is an error message, this element in the statement is ignored, and the statement is otherwise accepted.

Flow chart 113 shows the continuation from flow chart 111. It is entered only at OPWD and OPSD, which are the two points of normal exit from flow chart 111. One subroutine is called on Flow chart 113; DEV (flow chart 7) for getting the record length in case of normal blocking.

The normal exits are OPWT, for a write file with physical blocking, and to OPWE for all other sorts of file. The error exits are to ERRO4 if the statement is too short to be complete; to ERRO7 if the mode letter for the tape unit is not U or B, or if the tape number is missing or is not a digit between 1 and 6; to ERR15 if the input and output tape numbers of a copy file are the same; and to ERRO8 if the statement tries to open a save file with physical blocking, or contains something unintelligible where the indication of the type of blocking should be.

Flow chart 114 shows the continuation from flow chart 113. It is only entered at OPWE and OPWT, which are the two points of normal exit from flow chart 113. The normal exits are to VA for a write file with physical blocking, and otherwise to OPWP or OPWQ in flow chart 115 (see the lower right-hand corner of flow chart 114, and the beginning of flow chart 115.)

Subroutines called are DEV (flow chart 7), for finding the binary form of the buffer length; SCLAB (flow chart 13), for making sure the file name is not already in the symbol table as the name of something else; and ATTIL (flow chart 12), for putting the file name into the symbol table.

Besides the normal exits to VA, OPWP, and OPWQ, there are error branches to ERR04 if the statement is too short to be complete; to ERR09 if the word in the statement that should be the length of the buffer is not a number; to ERR10 if it is a number but is higher than 4000; to ERR12 if the file name has already been used as the name of something else. An error message is given if the file has normal blocking, and the buffer length is not an integral multiple of the record length. But the compiler accepts the statement, after effectively reducing the buffer length to the nearest lower integral multiple.

Flow chart 115 gives the rest of the routine for compiling OPEN statements. It is entered from flow chart 114 at OPWQ or OPWP. The only exit is to VA. The only subroutine called is STS (see flow chart 116), which is named in the lower box in the left-hand column, and in the box in the middle of the page.

Flow chart 116 shows subroutine STS, which is used at only two points in the compiler, both in flow chart 115. It is used to output a series of SPS cards putting constants in adjacent words of the 1401 memory. No subroutines (except SPS) are used, and the only exit is the normal return.

Flow chart 117 shows the routine for compiling 9CLOSE and CLOSE statements. The normal exit is to VA; there are also branches to ERR32 if the file named is not in the symbol table, and to ERR42 if what should be a write file name is in the symbol table as the name of something else.

If a 9CLOSE statement names a write file which does not have normal blocking, it is treated as a CLOSE statement, and no special indication is given.

The subroutines called are SCLUB (flow chart 24), to scan the symbol table for the file name, and SPDA (flow chart 93), to output the SPS card containing the address constant equivalent to the file name.

Flow chart 118 shows the routine for compiling READ, COPY, and UNSAVE statements. Apart from the normal exit to VA, there are error branches to ERR32 if the file name is not found in the symbol table; to ERR35, ERR33, or ERR32 if the file name is in the symbol table, but not as the proper sort of file for the function code; to ERRO4 if the branch-at-end-of-file address is not given. If a punctuation, literal, or non-final blank is found where the branch-on-repeated-redundancy address is allowed, an error message is given (ERR18) but the statement is accepted and compiled as if it ended after the branch-on-end-of-file address.

Subroutines called are SCLUB (flow chart 24), to find the file name in the symbol table; RWA (flow chart 119), to output the SPS card images containing a branch to the appropriate subroutine in the 1401 program package, and the name of the file ; TWIG (flow chart 106), to record the possible branches to the end-of-file address and the repeated-redundancy address, for later use in making the automatic flow chart; and SPDA (flow chart 93), to output the SPS card images containing these possible branch addresses as address constants.

Flow chart 119 shows subroutine RWA, which is called in flow charts 118, 120, and 122; i.e., in the compilation of all READ, WRITE, COPY, SAVE, UNSAVE, and OUTPUT statements. The branch to the proper 1401 subroutine is selected before entering RWA, and then RWA outputs the SPS card image containing this branch, followed by an SPS card image containing the address constant for the file named by the word next after the function code. It also checks whether the statement contains at least one more word after this file name, and branches to ERRO4 if not. This is the only special exit. The only subroutine called is SPDA (flow chart 93) for outputting the file name as an SPS address constant.

Flow chart 120 shows the routine for compiling OUTPUT statements. The only choice in the compilation concerns whether the second file named, whose buffer contains the information to be output, is a read/copy file or a save file. Two different subroutines in the 1401 program package are involved, and accordingly the branch to PUTB or PUTA in this flow chart is taken. On the significance of switches QOP and QOR, see the notes for flow chart 20.

Subroutines called are SCLUB (flow chart 24), to make sure that the write file on which the information is to be written is really in the symbol table as such; SCLAB (flow chart 13) to look up in the symbol table the name of the file containing the information; RWA (flow chart 119), to output the SPS card images containing the branch to the proper routine in the 1401 program package, and the

name of the write file as an address constant; and SPDA (flow chart 93) to output the address constant for the file containing the information to be written.

Besides the normal exit to VA, there are branches to ERR32, if the name of the write file is not in the symbol table; ERR42, if it is in the table but as some other kind of name; ERR63, if it is the name of a write file, but not one with physical blocking; ERR04, if the name of the file containing the information to be written is missing from the statement; ERR32, if it is in the statement but not in the symbol table; and ERR29, if it is in the symbol table but not as the name of a read, copy, or save file.

Flow chart 121 shows the routine for compiling PAUSE statements. Recall that the function code PAUSE may stand by itself in the statement (branch to PAUSB in the flow chart) or be followed by the label of some other statement (rightward branch from the second box in the flow chart) or be followed by a number in parentheses, which is to be the B-address of the halt instruction.

Subroutines called are DEV (flow chart 7) to convert the number between parentheses to binary (it is converted back to 4-digit decimal form at the bottom of the flow chart by subroutine BT4D, which here, as usually, is not explicitly mentioned); and TWIG (flow chart 106) to record the possible branch for the automatic flow chart.

Besides the normal exits to VA, there are branches to ERR04, if the word PAUSE is not followed by a blank and then either a word or an opening parenthesis; and ERR25, if the element after the opening parenthesis is not a number, or if the number is above 7999.

Flow chart 122 shows the routine for compiling WRITE and SAVE statements.

Subroutines called are SCLUB (flow chart 24) to make sure the word after the function code is in the symbol table; RWA (flow chart 119), to output SPS card images containing a branch to the appropriate write subroutine in the 1401 program package, and the name of the save or write file as an address constant; LIT (flow chart 108) to deal with a literal if that is what is to be written; SCLAB (flow chart 13) if any other sort of thing is to be written, to look up its name in the symbol table; SPDA (flow chart 93) to output SPS card images containing address constants for a read, copy, or save file name (at WRITM in the flow chart) if the material to be written is to be taken from such a file, and for the branch-on-buffer-full address if the statement is a SAVE statement; and TWIG (flow chart 106) to record such a possible branch for use in constructing the automatic flow chart.

Besides the regular exit to VA, there are branches to ERR32 if the word after the function code is not in the symbol table; to ERR52 or ERR42 if it is in the table, but is not the name of a file of the class appropriate to the function code; to ERR28 if the element of the statement that indicates what is to be written is not a word or literal; ERR55 if it is a word

not in the symbol table; ERR55A if it is the name of a switch or a statement label; ERR29 if it is the name of a write file; ERR46 if, in a SAVE statement, what should be the branch-on-buffer-full address is a blank, punctuation, or literal; or ERR04 if that branch address is missing altogether.

Flow chart 123 shows subroutine DSKR, which "reads" the 1401 program package, card by card, from the memory area into which it was read from disc by subroutine QRAD, just under "SECOND" at the top of flow chart 15. The address in location DSKRA is set = FAKE in flow chart 20, just before DSKR is first called in the compilation of any individual program.

There are two groups or "files" of card images in what begins at FAKE; the first ends with a card containing " (EIOP) " (end of input-output package), and the second with a card containing " (ELDR) " (end of loader). When either of these special cards is found by DSKR, return 1, a sort of end-of-file, is taken. Otherwise, return 2. The first of the special cards terminates the use of DSKR in flow chart 20, and the second terminates its use in flow chart 124. These are the only two places in the compiler where subroutine DSKR is called, and the complete set of card images beginning at FAKE is gone through once per program compiled, the first part up to "(EIOP)" in flow chart 20, and the remainder in flow chart 124.

Flow chart 124 shows the beginning of the last section of the compiler. It is entered at VEND, to which control can come from five places: two on the right side of flow chart 20,

after an END card has been read (these are the points from which VEND ought to be reached) and six at the six places where subroutine PREAD is called, in flow charts 10, 11, 15, 134 and 135 (these all represent a coding error, a program having been ended by the end of the input file without an END card.)

The exits are to WENT, at the beginning of flow chart 128, WCA, at the beginning of flow chart 125, WA, at the beginning of flow chart 126, and WBH, in flow chart 127.

From WENT, the program goes through flow chart 128 and branches to WENTA, at the top of flow chart 131. From WCA, the program goes through part or all of flow chart 125 and branches to WCBB, in flow chart 127, and through that flow chart and back to WBD, in the middle of flow chart 124. From WA the program goes through flow chart 126 and then to WB, in flow chart 127; after going through flow chart 127, it returns to WBD in the middle of flow chart 124. From WBH in flow chart 127, it returns immediately to WBD.

There are no error exits as such from flow chart 124.

The box that says, "make flow chart in listing (see VEND&2 to LVWRX)" covers a long section of the compiler program, which is not flowcharted because it is essentially independent of the production of a 1401 machine-language program. If one removed everything in the program from VEND&2 to LVWRX&1 inclusive, the compiler would work just the same but would not produce a flow chart automatically in the listing. However, a partial explanation is given here:

Each line of the printed flow chart is output by subroutine LVWR after being set up in locations SAVEA to SAVEA&21. The

subroutine can be entered at LVWR or LVWRA. The calling sequences are the following:

```
at LVAB-3:  TSX LVWR,4
            PZE LU
at LMDB:    TSX LVWR,4
            PZE LM
at LMDC-2:  TSX LVWRA,4
            PZE LM
at LMBA-2:  TSX LVWR,4
            PZE LU
at LMBA&25: TSX LVWR,4
            PZE LL
```

LU ("left upper"), LM ("left middle"), and LL ("left lower") are each at the beginning of a series of 66 words. Each of these words contributes its rightmost two characters to the printed line; hence 22 words are obtained.

The first 30 words, beginning at LU, LM, or LL, represent 30 columns on the left side of the page. Each column is two characters wide, and could contain (using "b" to represent a blank): bb lb *- *= -- or ==

A branch on the left side of the page will be represented by something like *-----

```
  1
  1
  1
  *=====
```

but the number of 1's forming the upward line is variable, and

so are the number of --'s to the right of *- , and the number of =='s to the right of *= .

The next four words, LU&30 to LU&33, LM&30 to LM&33, or LL&30 to LL&33, will contain either eight blanks, or six blanks with a parenthesis at either end, or the six characters of a 1401 instruction label (which will be either a statement label coded by the programmer, or one of the labels in the series X11111 ff., Y11111 ff., Z11111 ff.) with a parenthesis at either end, or ".....", "..////..", or " 11 " .

The thirty words beginning at RU ("right upper"), RM ("right middle"), and RL ("right lower") represent thirty columns on the right side of the page. Each column is two characters wide and could contain: bb bl -* *= -- or ==

A branch on the right side of the page will be represented by something like

```

=====*
      1
      1
      1
-----*

```

but the number of 1's forming the downward line is variable, and so are the number of --'s to the left of -* and the number of =='s to the left of *= .

Thus the flow chart consists of a central channel, eight characters wide, containing the labels of instructions, with boxes enclosing labels or groups of labels; thirty two-character columns to the left; and thirty to the right, containing the

representations of branches.

When subroutine LVWR is entered at LVWRA, whatever is in the 30 locations beginning at LU, LM, or LL, in other words the material for the left side of the chart, is used as it is. But when it is entered at LVWR, the contents of those 30 cells have their sequence reversed. The reason is that it was usually convenient to think of the columns on the left as numbered from the center outwards, but when the moment came, in subroutine LVWR, to assemble them along with the columns on the right, they had to be reversed. For the thirty columns on the right, the problem never arises, because the center-outwards order is the same as the usual left-to-right order.

To see how the material for the columns is constructed, we go back to just after VENDA, and before KVA. We have a list beginning at TWIT, showing all the possible branches. See the discussion of flow chart 106 for the organization of this list. It is a list of labels to which branches occur, and to each label of this kind is attached a list of labels from which branches to it can occur.

We also have a general symbol table beginning at location SNAL, which contains all the symbols that have appeared in the program, more or less in the order of their first appearances in the program. The exception to this order is that when a statement label has appeared as a branch-to address, and its own statement appears later in the program, the recording of the earlier appearance will be erased from the list beginning at SNAL.

Thus the list contains the labels of all labelled statements in the same order as that of the statements. When a label has occurred as a branch-to address but its statement has never turned up, it will be in the list in its original sequential position, as an undefined label, and will cause a group of slashes above it in the flow chart as well as an error indication in the second pass of the SPS assembly.

In short, we have a list from which we can extract all the labels that have occurred, in the same order as their statements. Undefined labels will be marked as such in the list, and their relative position in it does not matter much.

Assuming that there are not more than 299 statement labels in the program, we have 300 pairs of locations beginning at SYMBS. As we get each label out of the list beginning at SNAL, we put it in the first cell of the next available pair. In the second cell, we put PZE 0,0,0 for labels of statements to which control could pass without a branch from the immediately preceding statements; PZE 0,1,0 for labels of statements to which it could not so pass; and MZE 0,1,0 for undefined labels.

In the list that begins at SYMBS, every label in the program has an implicit position number, and every branch could be specified by the earlier label it involves, the distance in the SYMBS list to the later label it involves, and whether the branch is forwards or backwards in the list.

The program now constructs, for each label in the SYMBS list, two lists beginning at the word that follows the label itself and showing all the branches to or from later labels. This is done in three stages:

(1) Each label now has an implicit number, between 0 and a possible maximum of 596 (299 labels possible.) In the list beginning at TWIT, every label, whether a branch-to label in the primary list or a branch-from label in a secondary list, is replaced by its number.

(2) The secondary lists are attached to the odd-numbered words in the space beginning at SYMBS, using the addresses that were left vacant in the words stored above as PZE 0,1,0; or PZE 0,0,0 .

(3) For each label in the SYMBS list, all the lists attached in step (2) are scanned to see if that label occurs as a branch-from. If so, a second list is constructed in the workspace between WORK and WKLIM, giving the numbers of the labels to which that label branches. This new list is attached by means of the decrements of the words that were initially PZE 0,0,0 or PZE 0,1,0 and whose addresses were used in step (2) above to attach the list of the branches to the symbol.

Now to give an example, suppose the program contains only five labelled statements, occurring in the order TORINO, ROMA, MILANO, ZURIGO, EMPOLI; and that there are branches from TORINO to ROMA and MILANO, from ROMA to MILANO, ZURIGO, and EMPOLI, from MILANO to ROMA and EMPOLI, from ZURIGO to TORINO, and from EMPOLI to MILANO. Using the colon for "contains", we should have the following:

SYMBS&0: BCI 1,TORINO

SYMBS&1: PZE A,,B

SYMBS&2: BCI 1,ROMA00

SYMBS&3: PZE E,,F

SYMBS&4: BCI 1,MILANO

SYMBS&5: PZE N,,P

SYMBS&6: BCI 1,ZURIGO

SYMBS&7: PZE U,,V

SYMBS&8: BCI 1,EMPOLI

SYMBS&9: PZE X,,Y

SYMBS&10: PZE O,,O

WKLIM-A: PZE C

WKLIM-A&1: PZE O

WKLIM-C: PZE O

WKLIM-C&1: PZE 6 (to Torino from Zurigo)

WKLIM-B: PZE 2,,D (from Torino to Roma)

WKLIM-D: PZE 4,,O (from Torino to Milano)

WKLIM-E: PZE G

WKLIM-E&1: PZE 2

WKLIM-G: PZE H

WKLIM-G&1: PZE O (to Roma from Torino)

WKLIM-H: PZE O

WKLIM-H&1: PZE 4 (to Roma from Milano)

WKLIM-F: PZE 4,,K (from Roma to Milano)

WKLIM-K: PZE 6,,M (from Roma to Zurigo)

WKLIM-M: PZE 8,,O (from Roma to Empoli)

WKLIM-N: PZE Q
 WKLIM-N&1: PZE 4
 WKLIM-Q: PZE R
 WKLIM-Q&1: PZE O (to Milano from Torino)
 WKLIM-R: PZE S
 WKLIM-R&1: PZE 2 (to Milano from Roma)
 WKLIM-S: PZE O
 WKLIM-S&1: PZE 8 (to Milano from Empoli)
 WKLIM-P: PZE 2,,T (from Milano to Roma)
 WKLIM-T: PZE 8,,O (from Milano to Empoli)
 WKLIM-U: PZE W
 WKLIM-U&1: PZE 6
 WKLIM-W: PZE O
 WKLIM-W&1: PZE 2 (to Zurigo from Roma)
 WKLIM-V: PZE O,,O (from Zurigo to Torino)
 WKLIM-X: PZE Z
 WKLIM-X&1: PZE 8
 WKLIM-Z: PZE AA
 WKLIM-Z&1: PZE 2 (to Empoli from Roma)
 WKLIM-AA: PZE O
 WKLIM-AA&1: PZE 4 (to Empoli from Milano)
 WKLIM-Y: PZE 4,,O (from Empoli to Milano)

Every branch is recorded twice, once under the label the branch comes from, and once under the one it goes to. But the program will only be interested in the recording under the earlier label, as this will indicate the top of a branch line. The branch line will be continued down the page to the other end of the branch, on the

left of the page if the program is logically branching upwards, and on the right if logically downwards.

The program has now reached LVA in the listing. The addresses SYMBS in LVA and SYMBS&1 in LVBA were initialized just after VENDA, in flow chart 124. The address of both LVA and LVBA is increased by 2, after each label is dealt with, and when instruction LVA gets a zero, there are no more labels. The flow chart is terminated with a line of periods under the last box in the centre, and the program goes to VENX, on flow chart 124.

Initially, blanks are put in the four groups of 30 locations beginning at RU, RL, LU, and LL. The 30 locations beginning at RCOL, which initially contain zeros, represent the thirty columns on the right side of the page, and the 30 locations beginning at LCOL, also initially containing zeros, represent the thirty columns on the left. However, the ones on the left are numbered from right to left; thus LCOL and RCOL contain information about the columns nearest the center of the page, while RCOL&29 and LCOL&29 contain information about the columns at the edges of the page.

Whenever the part of the program beginning at LVBA finds a branch from the label now being considered to a later label, it looks for a word in the RCOL series which is now zero, indicating a free column, and then puts the distance covered by the branch into that word. The same is done by the part of the program beginning at LVFB&1; if there is a branch to the current label from a later label, it looks for a free word in the LCOL series, and puts the length of the branch into it. At the same time, the former part

of the program puts `=*` in the corresponding word of the RL series, showing the upper corner of the branch line, and the latter part of the program puts `*-` in the corresponding word of the LL series, showing the upper corner of the branch line. At RLHI, and LLHI, an indication of the corner farthest from the center on that line in each direction is stored.

Before each new label begins to be dealt with, the program prepares two lines full of blanks for the flow chart (4 groups of 30 words beginning at RU, LU, RL, LL). Then it checks the contents of all the words in the RCOL and LCOL series; wherever one of these is non-zero, 2 is subtracted from it. If the result is 0, the lower end of a branch has been reached, and must be noted for the current label. If the result is not zero, it is replaced in the word in the RCOL or LCOL series, and the program stores `b1` or `1b` in the corresponding word of the RU or LU series. If the result is zero, the program stores `-*` or `*=` in the corresponding word of the RU or LU series. `1b` or `b1` represents a continuation of a downward line, while `-*` or `*=` represents the lower corner of a branch. When a lower corner is stored, RUHI or LUHI is checked to see if it is further from the center than any other lower corner on that line, so that the farthest-out one on each side can be found afterwards from RUHI and LUHI.

When all the continuations of branches and lower corners of branches for the current label have been dealt with in this way, `==` or `--` is put into words in the RU and LU series as necessary so as to join the farthest lower corner on each side to the centre.

Then:

(1) (a) If the label now being dealt with does not begin with X, Y, or Z, it is one that was coded by the programmer, and so a new box on the flow chart is to begin with it. A line is set up in the sixty cells beginning at LM, with continuations of all vertical lines coming from corners that have been listed and going to corners that have not yet been listed, and eight periods in the middle to close the preceding box. Then this line is output by subroutine LVWR. Then blanks or two 1's with three blanks on either side are put in the middle of the LM-row, and it is output again; but this time LVWR is entered at LVWRA, because the thirty columns on the left have already been reversed once. Whether there are two 1's in the centre of the line depends on whether the current label is one to which control could come from the preceding statement without an explicit branch.

This is done between LMA and LMDC-1 in the compiler program listing. The two writings out of the LM-row are not done if the current label is the first in the program -- location LMDA is used as a switch for this purpose just before LMDB in the listing.

Then, between LMDC and LMBA-1, either or ..////.. is put in the centre of the LU-row, to form the top of the new box, and the LU-row is written out. The slashes are used if the label is an undefined one.

(b) If the label now being dealt with begins with X, Y, or Z, it is presumed to be one supplied by the compiler, and hence not important enough to the programmer to begin a new

box. The LL-row, which contains the label itself, will have to be output, but the LU-row may not have to be; it is useful only to provide room for branch lines to be drawn.

The lines that come horizontally to a label from the lower corner of a branch are put in the LU-row, while those that come in from the upper corner of a branch are put in the LL-row. If cell LU or RU contains part of a horizontal, the LU-row must be written out. If cell LL contains part of a horizontal, and cell LLP (which contains what was LL for the last label) also does, the LU-row is written out to provide separation between the two horizontals. Similarly for RL and RLP. Of course, however many reasons there may be for writing out the LU-row, it is only written out once. It has blanks in the middle.

This is done between LVK&4 and LMA-1 in the compiler listing.

(2) Beginning at LMBA in the compiler listing, the label is put in the middle of the LL-row, and it is output. The contents of LL and RL are saved in LLP and RLP respectively, for the test mentioned in (1) (b) above.

If, on either side of the diagram, the program finds itself having to find room for more than thirty vertical lines, it abandons the flow chart and branches to VENX. Otherwise, it branches to VENX, after completing the flow chart, at LVAB-1.

This brings us to VENX, on flow chart 124, and ends the discussion of the automatic flow chart.

Subroutines called in flow chart 124 are DSKR (flow chart 123) as already mentioned, and TRITE (flow chart 133), once to deal with the cards of the 1401 loader, and once to deal with assembled SPS instructions, one line per instruction. The cards of the loader are written out on the monitor punch tape, and also on file WENTER, where they are saved for inclusion in the listing after the assembled SPS instructions. The assembled instructions are written out directly on the monitor listing tape. (The absolute versions of the assembled instructions are meanwhile being gathered into card-length groups, and written on the punch tape and on file WENTER, just as the loader cards were, at WCBC on flow chart 125, WBC in flow chart 127, and at the very end of the program at two points in flow chart 128.)

Flow chart 125 shows a section of the last part of the compiler. It is entered only at WCA, and only from the lower right-hand corner of flow chart 124, when a constant is to be dealt with. The only two exits both go to WCBB, at the top right-hand corner of flow chart 127, and thence through that flow chart to WBD in the middle of flow chart 124.

Subroutines called are WAC (flow chart 129) and WAE (flow chart 130) for evaluating SPS addresses, and TRITE (flow chart 133) for writing the absolute constant card on the punch tape and on file WENTER. (The constant in symbolic form will be written on the normal listing tape just after WBD in flow chart 124.)

Flow chart 126 shows a section of the last part of the compiler. It is entered only at WA, to which the program can come from several points in flow chart 124, when the op-code of a 1401 instruction has been determined but the rest of the instruction remains to be assembled.

The exit from flow chart 126 is always to WB, at the beginning of flow chart 127. The only subroutines called are WAC (flow chart 129) and WAE (flow chart 130) for evaluating the A- and B-addresses of the instruction.

Flow chart 127 shows the continuation from flow chart 126, at WB, or from flow chart 125, at WCBB, through to the return to Flow chart 124 at WBD. It can also be entered at WBH from flow chart 124, when a "DS" card is encountered.

WB is reached when an instruction has been set up in absolute form in the record for listing. It is then put into the buffer beginning at PCARD, from which absolute program cards are written out on the punch tape.

WCBB is reached when a constant has already been set up and output on the punch tape, and has been set up ready to be output on the listing tape. If the SPS "instruction" that created the constant had * as its A-address, the constant is mixed up with instructions in the series of words constituting the program; since the constant has been output for punching from the CCARD buffer, the continuity of the words in the PCARD buffer has been broken, and so PCL is set = 81 so that the next time an instruction has to be put in the PCARD area, the previous card will be written out, and a new one will be begun with the address of

the first instruction after this constant.

At the conclusion of flow chart 127, WBH, the SPS-language part of the current record for listing is moved 36 characters rightward. This is intended to give them some visual separation from the compiler-language statements with which they are vertically mixed.

The only subroutine called in flow chart 127 is TRITE, for writing out absolute program cards on the punch tape and one file WENTER.

Flow chart 128 shows the last section but one of the compiler. It is entered only at WENT, to which the program comes from flow chart 124 after reading the end of file RINTER, or an END card on that file. If there is an incomplete card of absolute instructions in the PCARD buffer, this is output; then the END card is converted to a transfer card at the end of the absolute program cards.

File WENTER, on which the absolute program cards have been saved for later listing, is now closed and rewound, and the same tape is prepared for reading as file RENTER. These cards have already been output for punching, but their listing has been delayed so as not to confuse further the main assembly listing. The program then goes to WENTA, in flow chart 131.

Subroutines called are TRITE (flow chart 133), to output a possible last card of instructions, and the transfer card, on the punch tape and on file WENTER; and WAC (flow chart 129) and WAE (flow chart 130) to evaluate the A-address on the END card.

Flow chart 129 shows subroutine WAC, which evaluates the first part of an A- or B-address (blank, or a four-digit number, or an asterisk, or an input-output unit address, or a symbol) and leaves the result in binary form in S&2. This is where the symbol table that was constructed by subroutine FPASS (flow chart 14, called only by subroutine WRATE, flow chart 5) is used.

Flow chart 130 shows subroutine WAE, which evaluates the address adjuster, if any, of an A- or B-address, adds it to what was left in S&2 by subroutine WAC, converts this to 3-digit decimal form, and zones the middle digit if necessary according to the indexing of the address. An undefined address is converted to the three characters ===

Both subroutines WAC and WAE have no error exits, and call no other subroutines. They are always called together, four times in all, in flow charts 125, 126 (twice), and 128.

Flow chart 131 shows the last part of the compiler. It is entered only at WENTA, to which the program comes from flow chart 128. First, file RANTER is copied onto the monitor listing tape. This contains the card images of the absolute program deck that was also written on the punch tape, except that record marks have been changed to dollar signs. Then various parts of the compiler program are initialized, as shown in the large box in the middle of the flow chart.

Then subroutine BREAD is used (see the notes on subroutine BREAD, flow chart 1, and subroutine PREAD, flow chart 3, to see why BREAD and not PREAD is used here). Something from the ordinary monitor input file is read. This can be:

- (1) A BCD card image containing " (PROG) " in columns 1-6.

This signals the beginning of a new program to be compiled. The program goes through WUNF to VAN, on flow chart 15, and compilation of the new input program begins.

- (2) A BCD card image not containing " (PROG) ", or a binary card image. This is supposed to be a data card for the last-compiled program to use when it is tried out by the 1401 simulation program. It is copied onto disc by subroutine WDK (flow chart 132). However, if the card is BCD and has group marks in its first 6 columns, it is copied as if it were an ordinary 7-8 card. An ordinary 7-8 card could not have been included in the input file conveniently, as it would have become a tape mark.

- (3) A tape mark, which signals the end of the run. At GOUT, one more program card is copied onto the disc, such that if the last-compiled program is tried out and ends with a RELOAD statement, this will cause a programmed 1401 halt rather than some random action depending on what happens to be on the unused part of the disc. Then the disc-track buffer area is copied onto the appropriate track, since it probably contains a waiting incomplete track image, and the run is terminated at HOUT.

The only exits from flow chart 131 are to VAN and the final exit at HOUT. Subroutines called are BREAD (flow chart 1) and WDK (flow chart 132) three times.

Flow chart 132 shows subroutine WDK, which is used for copying absolute 1401 program cards, as produced by the compiler, and data cards that may be between the programs in the input file, onto the common tracks of the disc file. Thus there is set up the image of a file of cards, and it is possible to try the effect of loading this file into a 1401 and pressing the LOAD button immediately after the compilation. The compiler job may be followed immediately, in the monitor input, by a job using the 1401 simulator program with a U in the control card.

WDK uses the subroutine WRDISC, which is not part of the compiler program, to write the 462-word buffer beginning at DKBUF on successively higher-numbered tracks of the common disc storage cylinders.

WDK is called once in subroutine TRITE (flow chart 133) to write program cards on the disc, and three times in flow chart 131 to write data cards -- once for each BCD card, and twice for each binary card, since it moves only 14 words at a time.

Flow chart 133 shows subroutine TRITE, which has two different uses depending on the calling sequence. This is explained at the top of the flow chart. There are no exits except

the normal return, and the only subroutine called is WDK (flow chart 132).

TRITE is called only once with an IOCD prefix in the calling sequence, near the middle of flow chart 124. With an MZE prefix, it is called just below VEND in flow chart 124, twice in flow chart 128, at WBC in flow chart 127, and at the end of flow chart 125 at WCBC.

Flow chart 134 shows the beginning of the routines for compiling SORT FIRST PASS, SORT MERGE, and SORT FINAL MERGE PASS statements; and the completion of the latter two. The compilation of a SORT FIRST PASS statement is completed on flow chart 135.

Flow chart 135 is merely a continuation of flow chart 134; it is entered only at HBE in the upper left-hand corner, to which control goes from the upper right-hand corner of flow chart 134. We shall discuss 134 and 135 together.

A SORT FINAL MERGE PASS statement is merely translated into the SPS instruction B XFMER . If there is not, elsewhere in the program, a SORT MERGE statement, the 1401 package program in which XFMER occurs will not be selected, and XFMER will finally turn up as an undefined symbol.

A SORT MERGE statement is turned into:

- (a) the SPS instruction B XMERG, i.e. a branch to the package program that does the work.
- (b) four SPS "DS" instructions, equating "YTHREE", "YFOUR", "YONE", and "YTWO" to the absolute values of the names of the two read and two write files that must be named after the function code. This is done by the four calls on subroutine

HBV (flow chart 136) in the lower left-hand corner of flow chart 134.

(c) two pairs of SPS instructions that essentially equate "YRDJ" and "YWTJ" to the names of the read and write routines in the program package that are appropriate to the type of blocking in file "YTHREE". Since SPS coding does not allow the equating of two symbols, and since the absolute values of XRDN, XRDM, etc. are not yet known, we cannot equate them directly. And we cannot use a simple branch instruction to give the effect of equating them, because of the "store B-address register" instructions at XRDN, XRDM, XRDS, and so on. A pair of instructions, as shown in flow chart 138 for subroutine HBT, has to be used for each effective equation. This is all done by subroutine HBT, flow chart 138, called in the lower left-hand corner of flow chart 134.

(d) SPS "DC" and "DCW" instructions setting up two words of blanks, named XACH and XBDH, exactly long enough to contain the field called KEY in the file equated to YTHREE. These words are used as workspaces in which the preceding key in each of the input files can be found, so as to show whether the present key breaks the sequence. These are output by subroutine HKEY, flow chart 137.

(e) SPS "DS" instructions defining XKEY as the offset of the right end of field KEY in the file equated to YTHREE, and XLKEY as the offset of the left end of the same field. This is also done by subroutine HKEY, flow chart 137.

(f) An SPS no-op instruction with label YFIN, which gives the merge program in the package an address to return to when it has

finished. This is simpler for the merge program than having to behave as a subroutine in the matter of returning. Since the merge program can only be called once in a program, it need not return as a subroutine normally does.

A SORT FIRST PASS statement must have all its specifications coded as if in three following statements. It would have been possible to require the programmer to code, e.g.

```
SORT FIRST PASS FIND,GCARD,BUFA,BUFB,BUFC,BUFD,OUTA,OUTB
```

but it seemed, for once, better to have the information spread out as:

```
SORT FIRST PASS
```

```
ENTRY XEC FIND USE GCARD
```

```
BUFFERS BUFA,BUFB,BUFC,BUFD
```

```
EXIT OUTA,OUTB
```

This is the only type of statement with such an extended format, and consequently the only one in compiling which subroutines PREAD and DECOM (flow charts 3 and 9) are called anywhere except in flow chart 15.

The word following USE, symbolized by YYYY in flow chart 134, is treated rather in the way the name of the record to be saved is treated in compiling a SAVE statement -- compare flow chart 122. The address of the address of the leftmost character in the record to be sorted is, in the last box before HBE in flow chart 134, furnished to the package sort program as an address constant whose own label is XINPUT.

All in all, a SORT FIRST PASS statement with the three supplementary statements that must follow it is translated into:

- (a) a simple SPS branch instruction equating XRCARD with whatever follows ENTRY XEC in the statement; this must be the compiler-language label of a subroutine for getting the next record to be sorted.
- (b) an address constant labelled XINPUT, as explained ten lines above, for locating the record to be sorted.
- (c) four SPS "DS" instructions equating XBUFA, XBUFB, XBUFC and XBUFD to the four save file names given after "BUFFERS" in the second supplementary statement; or rather to the absolute addresses equivalent to those names. This is done by the four calls on subroutine HBV (flow chart 136) in the leftmost column of flow chart 135.
- (d) two pairs of SPS instructions dealing with XRDJ and XWTJ just as YRDJ and YWTJ were dealt with in section (c) for a SORT MERGE statement, according to the type of blocking in the save file whose name is equated with XBUFA. This is done by subroutine HBT (flow chart 138).
- (e) and (f) -- the same as (d) and (e) above for a SORT MERGE statement, but based on the field named KEY in the file whose name is equated with XBUFA. These are done by the call on subroutine HKEY (flow chart 137) in the leftmost column of flow chart 135.
- (g) SPS "DS" instructions equating XTHREE and XFOUR to the absolute addresses equivalent to the write file names given after EXIT in the third supplementary statement. This is done by the two calls on subroutine HBV (flow chart 136) on the right side of flow chart 135.
- (h) An SPS no-op instruction with label XFIN. See the explanation in (f) above for a SORT MERGE statement.

To summarize, the subroutines called in flow charts 134 and 135 are PREAD (flow chart 3), DECOM (flow chart 9), SCLAB (flow chart 13), HBV (flow chart 136), HKEY (flow chart 137), and HBT (flow chart 138). The only exits are to VA, if the statement compiles correctly, or to ERR65 in case of any ERROR.

Flow chart 136 shows subroutine HBV, which is called several times in flow charts 134 and 135 to equate some file name as given by the programmer with the fixed name under which a package sort program will have to refer to it. There are error branches to ERR65 if the file name given by the programmer is not in the symbol table, or if it does not name the proper type of file (read, write or save) required at that point in the compilation.

The symbol table which SCLAB uses is not helpful in constructing the SPS "DS" instructions, as it does not contain the absolute address of the key point in the file, the address to which the file name is equivalent in the SPS version of the program. Therefore the symbol table used by subroutine WAC has to be used, because the file name must already have been put into that table, when the relevant OPEN FILE statement was compiled, sending several SPS cards through the subroutine FPASS. This is the only place where subroutine WAC is used, outside the second pass of the SPS assembly in flow charts 125, 126, and 128.

It would be simpler if one could code, in SPS, something like

```
03 XBUFA DS BUFA
```

and leave it until the second pass of the SPS assembly to find out the numerical equivalent of BUFA. But we have followed the

ordinary conventions of SPS in not allowing such a DS instruction, and so this bit of the second pass has to be done ahead of time by the compiler.

Notice also that subroutine SPS is used rather oddly to put "BUFA", or whatever the file name given by the programmer is, into location SCARD so that it will appear, to subroutine WAC, as BUFA with blanks after it, and not as BUFAOO . SCARD is blanked again immediately afterward.

Subroutines called in flow chart 136 are SCLAB (flow chart 13), SPS (flow chart 16) and WAC (flow chart 129).

Flow chart 137 shows subroutine HKEY, which is called once each in flow charts 134 and 135. It sets up two necessary work spaces, named XACH and XBDH, and equates XKEY and XLKEY to the right and left offsets of the key field in the file indicated by the calling sequence. If SORT FIRST PASS and SORT MERGE occur in the same program, the compiler presumes that the latter continues the sort which the former began, and that therefore the length and location of the key field is the same. So subroutine HKEY will be fully executed only once per program, even if called twice; this is ensured by switch QFM, which also serves for the selection of a few cards in the 1401 program package, common to both phases of sorting.

The only subroutine called is SCLAB (flow chart 13). The only exit, besides the normal return, is to ERR65 in

case the file name indicated in the calling sequence is not in the symbol table; or if no field called KEY has been defined for that file by a DEFFLD statement.

Flow chart 138 shows subroutine HBT, which is called once each in flow charts 134 and 135. See the explanations in section (c) for a SORT MERGE statement and section (d) for a SORT FIRST PASS statement.

When subroutine HBT is entered, subroutine HBV (flow chart 136) has recently been executed and has left the description of some file in location HBVD. There are two bits in this word which indicate the type of blocking for the file, and they are used to get two branch instructions, appropriate to that type of blocking, from locations READL&n and WRITEL&n. The symbolic addresses to which these instructions branch are then equated, effectively, to either XRDJ and XWTJ, or YRDJ and YWTJ, according to the calling sequence by which HBT was entered.

No subroutines are called in flow chart 138, and there are no abnormal exits.

Flow charts 201 to 224 describe not the compiler, but various subroutines in the 1401 program package that accompanies the compiler.

Flow chart 201 shows subroutine XRDA as it operates when called in reading or copying a file. It is called for this purpose in flow chart 204 (subroutine XRDN for reading or copying a record with normal blocking), flow chart 206 (ditto with record-mark blocking), flow chart 207 (ditto with physical blocking), and 209 (ditto with standard blocking).

The address of the last-read record is put in index 2. But if the third character of this address is) it is not the address of the first character of the current record, but merely a sign that a block must be physically read before a record can be logically read. In that case control goes to XRDAD, otherwise to XRDAF.

If the name of the file is X, the address of the file is X, and what is described in the preceding paragraph is stored in cells X-2, X-1, and X. In cells X&1, X&2, and X&3 will be the length of the last-read record. This, at XRDAF, is added to the address of that record. If the address is now equal to the address in X&4, X&5, and X&6, the block is exhausted, and control goes to XRDAF. Otherwise switch XFIRST is reset, to show that the new record is not the first of a block, and

the address of the first character of the new record is put into index 3. Then the program takes exit 2 from the subroutine, showing that a block was not read during the execution of the subroutine.

At XRDAD the program sets up a read instruction to get the next block of the file. Then, if the file is a copy file, not a read file, and if the address of the last-read record does not end in) , which would show that the current physical file had not yet begun to be read, it writes out the last-read block on the output tape.

This brings the program to XRDAE, from which point the flow chart should be self-explanatory. If an end-of-file is read, the program goes to XRDAKS, puts) at the file address, writes two tape marks on the output tape if the file is a copy file, and branches to the end-of-file exit that was given in the calling sequence for the routine that called XRDA.

Flow chart 202 shows subroutine XRDA as it operates when unsaving a file. It is called for this purpose in flow chart 205 (unsave record with normal blocking), flow chart 206 (unsave record with record mark blocking), and 210 (unsave record with standard blocking.)

If X is the name of the file, there is a character at X&11 giving the status of the file -- N for neutral, R for read, W for write, and H for hold. If this is R, the program goes to XRDAF, adds the length of the preceding record to the

address at X, and tests for equality with the address at X&6. That address was stored when the file was last moved from write status to read or hold status, and indicates the limit up to which information was stored when the file was last in write status.

If the addresses are equal, the program goes to XRDAKT, puts the file in hold status, and takes the end-of-file exit. This was included in the calling sequence of the subroutine that called XRDA in its turn. Otherwise, return 2 from XRDA occurs, showing that the record just unsaved is not the first in the buffer.

If the file is in hold status, it is put into read status, and subroutine XSAVD is called (flow chart 203) to set the file address equal to the beginning of the buffer. This is X&25 if the file has standard blocking, and X&13 otherwise. It appears impossible to the author that (FILE) should then =(FILE&6), but timidity prevents him from removing the decision box just above XRDAY from the program. The address is then put in index 3, and return 1 occurs, showing that the record just unsaved is the first in the buffer.

If the file is in neutral status, nothing has ever been saved in it, and the end-of-file exit is taken at once.

If the file is in write status, X-2, X-1, and X contain the address plus 1 of the last character stored in the buffer by the most recent save instruction. This address is stored in X&4, X&5, and X&6 to mark the end of the saved file; then the file is put in read status, subroutine XSAVD is executed and so on as if the file had been in hold status.

Flow chart 203 shows subroutine XSAVD, which puts the address of the first character of a buffer into the location whose name is the same as the name of the file. It is only used for write and save files, and is called in subroutines XRDA (flow chart 202), XWTR (flow chart 214), and XWTA (flow chart 218). If the file has standard blocking, the address is the address of the name of the file, plus 25; otherwise plus 13. If standard blocking, it is also necessary to put this address, in four-digit form, at X&13, X&14, X&15, and X&16; and to put "0004" in X&21, X&22, X&23, and X&24.

If the program contains no write or save files with standard blocking, the compiler will have left out the instructions corresponding to the left side of the flow chart.

Flow chart 204 shows subroutine XRDN, as it is executed for a read or copy file with normal blocking. Most of the work is done by subroutine XRDA (flow chart 201). If the return from XRDA is on line 2, a new block has not just been read, and all that remains to be done is to test the switch for non-redundancy in the block. If this is on, the normal exit from XRDN occurs. This may be because the current block was not redundant, or because an UNCHECK statement has been applied to the current file since the block was read. The redundancy check branch in the calling sequence to XRDN may be 000 (i.e. left uncoded in the READ or COPY statement) but if not, the program now branches to that address, if the non-redundancy switch is off.

If the exit from XRDA is on line 1, a new block has just been read, and the program checks whether its length is such as to contain an integral number of records. If the address of the end of the block is the same as that of the end of the buffer, this must be so, because the length of the buffer was checked during compilation. Otherwise, the program constructs all the possible record addresses in the buffer; if none of them coincides with the end of the block, the block is bad and there is a halt. If the length of the block is acceptable, the program goes to XRDEND, as if the exit from XRDA had been on line 2.

Flow chart 205 shows subroutine XRDN, as it is executed for a save file with normal blocking. Most of the work is done by subroutine XRDA (flow chart 202). If the exit from XRDA is on line 2, a normal return from XRDN takes place immediately. Otherwise, XRDA has obtained the first record in the file, which has just been moved from write or hold status to read status. The buffer is checked to see that it contains an integral number of records, though the author cannot see a possibility of error in this, and then the normal return takes place.

Flow chart 206 shows subroutine XRDM as it is executed for a read or copy file with record-mark blocking (XRDM on the left) and also for a save file with record-mark blocking (XRDM on the right). In either case, most of the work is done by subroutine XRDA (flow chart 201 for XRDA on the left, and flow chart 202 for XRDA on the right.)

If the file is read or copy, and the exit from XRDA is on line 1, a new block has just been read, and the program puts a record mark in the last character of the block. There should be one there in any case, but the block might be one prepared by some other program, divided into records by record marks, but without a record mark at the end of the last.

Then, whatever the return from XRDA, the program goes to XRDMB. The address of the end of the current record, plus one, has been found by moving it into itself with a "P" instruction. The address of the beginning of the record is subtracted from this, after both have been converted to 4-digit numbers, and the difference is stored in the three characters reserved for the length of the current record. The program then goes to XRDEND, in flow chart 204, to deal with the question of redundancy and exit from the subroutine.

Flow chart 207 shows subroutine XRDP, for reading or copying records with physical blocking. Once again, most of the work is done by subroutine XRDA (flow chart 201), the return from which is always on line 1 because every record is the first and only one in a block. The address of the end of the new record, plus one, is the same as for the block, and this has already been saved by subroutine XRDA. Using this address, the program goes to XRDMB and proceeds as for a file with record-mark blocking.

Flow chart 208 shows subroutine XSKP. For a read or copy file, the execution of this subroutine means simply that) is put

at the location named by the file name. The next time the file is read or copied, anything now in the buffer will be ignored, and a new block will be read immediately. (See the first decision box in subroutine XRDA, flow chart 201.)

For a save file, the subroutine makes sure that it is in hold status, unless it is still in neutral status. If it is in read status, this involves nothing more than changing the status character; if it is in write status, the address of the last point reached must be stored at the location named by the file name plus 6, to show the limit of readable information in the buffer.

Flow chart 209 shows subroutine XRDS, as executed for a read or copy file. Subroutine XRDA (flow chart 201) does most of the work, but some extra checks are carried out. If the return from XRDA is on line 2, the record is not the first in a block. Its first four characters are converted to 1401 3-digit form and stored in the space reserved for record length. They are also checked for being all digits, and are subtracted from the total length of the block, in the first four characters of the buffer. If this number becomes negative, something was wrong with the organization of the block.

If the return from XRDA is on line 1, the new record is the first of its block, and extra checks are made on the block, to see that its first four characters are digits that give the length of the block.

Flow chart 210 shows subroutine XRDS as it is executed for a save file. The only difference from flow chart 209 is that some tests are omitted. Since what is being unsaved must have been saved previously by the subroutine XWTS (flow chart 216) it must be well-organized.

Flow chart 211 shows subroutine XWOD, which is used by subroutines XWTA (flow chart 218), XWTP (flow chart 219), and XOUTR (flow chart 221) to execute a write instruction, and to take the necessary steps if the write instruction results in a redundancy check or an end-of-reel indication. The flow chart is self-explanatory. Note that after box XWODK has been executed, the computer hangs up until the unloaded tape has been replaced with another one, and the start button has been pressed on that tape unit.

Flow chart 212 describes subroutine XWMOV, which is used whenever a non-data instruction (write tape mark, rewind, unload, erase, or backspace) has to be applied to a tape. Its only purpose is to save a few characters in the program. A call on XWMOV requires only four characters for the branch and one for the function character, whereas an in-line method of getting the same result would require seven characters for the moving of the tape number into the tape control instruction, and five for the instruction itself.

Flow chart 213 shows subroutine XWTR as it is executed for a write file. It is called by subroutine XWTN (flow chart 215)

subroutine XWTS (flow chart 216) and subroutine XWTM (flow chart 217). The exit from each of these subroutines is at XWTX, and it may be noted that the address there is set by XWTR, using what the outer subroutine put in index register 1. Then XWTR fills up the index registers: XR1 with the address of the address of the first character in the record to be written (typically, this is the base address of a read, copy, or save file, and at that base address will be found the address of the record itself); XR2 with the base address of the write file; and XR3 with the address found at that base address, which is the address of the first available character position in the write buffer.

Flow chart 214 shows subroutine XWTR as it is executed for a save file. The difference consists of the extra things that must be done for a save file:

- (a) There is a possibility of an "end-of-file" condition, and the branch address for this has to be moved from the calling sequence to the instruction at XSAVC.
- (b) If the file is in read or hold status, the base address, showing the next available character position in the buffer, must be initialized by subroutine XSAVD (flow chart 203).

Flow chart 215 shows subroutine XWTN, for writing or saving a record in a file with normal blocking. Subroutine XWTR (flow chart 213 or 214) is used to put useful numbers in the index registers. If the name of the file is X, then locations X&7, X&8, and X&9 contain the address of the first character after the end of the

buffer. So if the address of the next available position of the buffer, found at X-2, X-1, and X, is the same as this, the buffer is full. Subroutine XWTA (flow chart 219) must be used to signal the "end-of-file" condition for a save file, or to write out the buffer on tape, in the case of a write file.

Flow chart 216 shows subroutine XWTS, for writing or saving a record in a file with standard blocking. This differs from XWTN in having some extra checks:

- (a) There is a halt if the first four characters of the record to be written or saved are not digits.
- (b) The length of the record is not constant, but depends on these digits; therefore their 3-digit equivalent is found and stored in the appropriate place. The four digits are also added to the previous length of the block, which is kept in the first four digits of the buffer. If the total is too large, the existing block is written out, and the program tries again to move the record into the buffer.
- (c) But if the address of the next available position in the buffer has its initial value, the record cannot be fitted into the buffer, and the program halts.

Flow chart 217 shows subroutine XWTM, which writes or saves records in files with record-mark blocking. After subroutine XWTR (flow chart 213 or 214) has been used to fill the index registers, XWTM counts the characters of the record, up to the nearest record mark, against the characters in the unused part of

the buffer. If the buffer is exhausted first, it is transferred to tape (in the case of a write file) or the end-of-file branch is taken (in the case of a save file) by subroutine XWTA (flow chart 218).

Then, for a write file, the character-by-character counting is repeated. If it happens that the buffer is exhausted first by this counting, while the address of the next available cell in the buffer has its initial value, the record is longer than the buffer and the program halts.

Once it is known that the buffer has room for the record, it is conveniently moved into the buffer with a P instruction.

Flow chart 218 shows subroutine XWTA, which is called by subroutines XWTN (flow chart 215), XWTS (flow chart 216), and XWTM (flow chart 217) to deal with the position when the buffer has not enough room to contain the record that is to be written or saved.

If the file is a save file, the "end-of-file" branch whose address was moved into instruction XSAVC by subroutine XWTR (flow chart 214) is taken.

Note that if there are one or more save files in the program, but the file currently being dealt with is a write file, the A-address of instruction XSAVC will be a nonsense at best, and an invalid address at worst. The branch in the instruction is not executed, however, and the 1401, at least the 8K 1401 in the Cetis building, does not hang up on account of the bad A-address.

The author did not notice this possibility in programming the subroutine, and it should cause no trouble. It may be that a different 1401, or the same machine after engineering modifications, would hang up on an invalid A-address, even when contained in a conditional branch instruction for which the condition is not met. Then it would be necessary to replace instruction XSAVC with the following two instructions:

```

      V *      &005    0011      2 1 SV
XSAVC  B 0000      SV
*****      END OF WARNING      *****

```

If the file is a write file, a group mark with word mark is put into its next available position, or into the position next after the end of the buffer if it is exactly full. The block is then written via subroutine XWOD (flow chart 211). Note that the beginning of the block is at a different position, relative to the basic address of the file, according as the file has standard blocking or not.

Then the word mark is removed from that group mark, and a group mark with word mark is put in the position next after the end of the buffer -- this is merely to correct the situation if the word mark has just been removed from that position because the whole buffer has just been written out.

The address of the next available position in the buffer is then initialized by subroutine XSAVD.

Flow chart 219 shows subroutine XWTR, for writing a record with physical blocking. This differs from the kinds of writing mentioned so far in that there is no write buffer, and no possibility of saving rather than writing.

The end of the record to be written is determined by the first group mark encountered. This is presumed to be without a word mark; in any case it will be left without a word mark after the writing.

The writing is actually accomplished by subroutine XWOD (flow chart 211).

Flow chart 220 shows subroutine XOUT, for carrying out an OUTPUT statement which writes the contents of a save file on a write tape with physical blocking. If the file is not in write status, nothing is done, but no error indication is given. XOUT determines the starting address of the block to be written, according to whether the save file has standard blocking or not.

If the starting address is also the next available address, nothing is done by the subroutine; but the author does not think this condition can arise.

The end of the block to be written is found not by looking for a group mark, but according to the next-available-location address in the save file. The program branches to subroutine XWTP (flow chart 219) at the point where the group mark with word mark is placed after the end of the block.

The writing is actually accomplished by subroutine XWOD (flow chart 211).

Flow chart 221 shows subroutine XOUTR, for carrying out an OUTPUT statement which writes the contents of a read or copy buffer on a write tape with physical blocking. It differs from the preceding subroutine because the block to be written begins not at the beginning of the buffer, but at the beginning of the most recently read record in the buffer. Also, it is important to restore the group mark with word mark at the end of the buffer, when the writing has been completed. This is not necessary in the buffer of a save file, since no tape ever gets read into it.

If the last accessing of the read or copy file by a READ or COPY statement found an end-of-file, the subroutine, and therefore the OUTPUT statement, will do nothing, but will give no special indication.

The writing is actually accomplished by subroutine XWOD (flow chart 211).

Flow chart 222 shows subroutine X9CLO, for closing a write file that has normal blocking in such a way that the last block is the full length of the buffer. If the buffer is either completely empty or completely full, the closing is done as if subroutine XCLOSE (flow chart 223) had been called in the first place.

If the buffer is partly full, the unused part is filled with 9's. A word mark is put on the first unused position of the buffer, a 9 is put into the last position, and a move instruction moves the 9 one position leftward. This results in filling the

memory with 9's down to the nearest position with a word mark, which is the first unused position of the buffer. The word mark is removed, and the file is closed as if the buffer had been full of information.

If the records of the file with normal blocking were each 1 character long, and all but the last record in the buffer contained information, this method of filling in 9's would wrongly replace the last record of information, i.e. the last record in the buffer but one, with a 9.

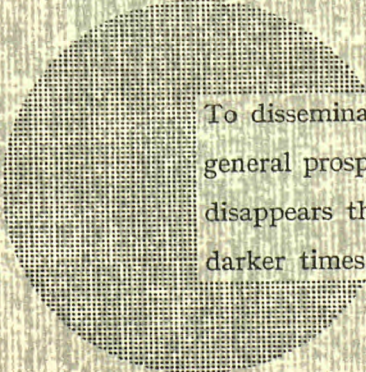
After the padding of 9's has been inserted, if necessary, the program branches into subroutine XCLOSE (flow chart 223).

Flow chart 223 shows subroutine XCLOSE, for closing a write file with standard or record-mark blocking, or with normal blocking if an incompletely full block at the end of the file need not be padded with 9's.

If the buffer is empty, the program goes directly from XCLOSA to XCLOSB; otherwise it uses subroutine XWTA (flow chart 218) to write out what there is in the buffer.

At XCLOSB, the subroutine gets the mode and number of the tape unit associated with the file (though the mode is hardly necessary) and at XCLOSD it uses subroutine XWMOV (flow chart 212) to write two tape marks on the tape and rewind it.

Flow chart 224 shows subroutine XCLSP, for closing a write file with physical blocking. Since there is no buffer, there is no question of writing out waiting records. The tape number and mode are found eleven positions to the left of where they would be found in a file with any other kind of blocking. After they have been found, the program joins subroutine XCLOSE (flow chart 223) for the tape marks and rewinding.



To disseminate knowledge is to disseminate prosperity — I mean general prosperity and not individual riches — and with prosperity disappears the greater part of the evil which is our heritage from darker times.

Alfred Nobel

SALES OFFICES

All Euratom reports are on sale at the offices listed below, at the prices given on the back of the cover (when ordering, specify clearly the EUR number and the title of the report, which are shown on the cover).

PRESSES ACADEMIQUES EUROPEENNES

98, Chaussée de Charleroi, Bruxelles 6

Banque de la Société Générale - Bruxelles
compte N° 964.558,
Banque Belgo Congolaise - Bruxelles
compte N° 2444.141,
Compte chèque postal - Bruxelles - N° 167.37,
Belgian American Bank and Trust Company - New York
compte No. 22.186,
Lloyds Bank (Europe) Ltd. - 10 Moorgate, London E.C.2,
Postcheckkonto - Köln - Nr. 160.861.

OFFICE CENTRAL DE VENTE DES PUBLICATIONS DES COMMUNAUTES EUROPEENNES

2, place de Metz, Luxembourg (Compte chèque postal N° 191-90)

BELGIQUE — BELGIË

MONITEUR BELGE
40-42, rue de Louvain - Bruxelles
BELGISCH STAATSBLAD
Leuvenseweg 40-42 - Brussel

DEUTSCHLAND

BUNDESANZEIGER
Postfach - Köln 1

FRANCE

SERVICE DE VENTE EN FRANCE
DES PUBLICATIONS DES
COMMUNAUTES EUROPEENNES
26, rue Desaix - Paris 15^e

GRAND-DUCHE DE LUXEMBOURG

OFFICE CENTRAL DE VENTE
DES PUBLICATIONS DES
COMMUNAUTES EUROPEENNES
9, rue Goethe - Luxembourg

ITALIA

LIBRERIA DELLO STATO
Piazza G. Verdi, 10 - Roma

NEDERLAND

STAATSDRUKKERIJ
Christoffel Plantijnstraat - Den Haag

EURATOM — C.I.D.
51-53, rue Belliard
Bruxelles (Belgique)