# Basic subroutine for the input of numbers, words, and special characters

*By* B. E. Cooper*

**The purpose of this paper is to describe a basic all-purpose format-free input subroutine and to show that such subroutines can be written with both efficiency and flexibility. A second purpose is to encourage the use of such subroutines to improve the often arbitrary presentation rules users normally must follow to communicate with a program. The subroutine reads one card at a time and assembles the information as a list of words, numbers, and special characters. The rules of assembly are defined in terms of two arrays of constants and may therefore be changed by program. The various decisions are taken quickly with reference to these arrays, and the programming is not machine dependent.**

(First received September 1967 and in revised form April 1968)

A great number of programs written, particularly if they pretend any generality, offer a number of different options which the user may select. The rules the user is expected to follow to make his selection often frighten potential users away. For example: if Blogg's analysis is required punch PQR in columns 14, 27 and 38 of card 4. If extra output is required punch 9 in column 54 of card 3, otherwise punch 8. If a fifth card containing a title is to be presented punch ZEBRA in columns 11 to 15 of card 2.

Rules of presentation are often more difficult to understand than the numerical method employed in the program, and much time, both users' and computers', is wasted because little thought is given to lightening the users' task. Some programs, of course, do allow flexibility in the way information is presented but I think that it is true to say that there is considerable room for improvement in this respect. Free-field format subroutines have been available for many years but the problem programmer has largely ignored them and has stuck to the standard FORTRAN format-bound instructions. This paper describes a particular subroutine of this kind which has been in use for some time as the basic input subroutine for a large statistical and data filing system called ASCOP (see Cooper, 1967) and argues that the particular organisation of this subroutine has a large number of advantages.

Use of such a subroutine enables the programmer to relax the restrictions he might otherwise insist on in the presentation of information to the program. Instead of insisting that the required information be punched in a rigid format, with a coding scheme for the selection of the required options, he can allow the user freedom of preparing information as though he was typing instructions for a subordinate. The selection of Blogg's method can be made conditional on the appearance of the word BLOGGS somewhere in the specification and a title can be introduced by the word TITLE itself. The assumption of default settings for parameters the user is not concerned with can be made more easily, and the user introduces only that subset of the specification he is concerned with when presenting his particular problem. In fact the programmer may go so far as to allow alternative means of introducing the same information. Different users often use different names for the same mathematical technique, and it is not hard to allow the use of two or more different words to refer to the same analysis.

* *Atlas Computer Laboratory, Chilton, Didcot, Berkshire.*

## General description

The name and argument list of the subroutine to be described is as follows:

SUBROUTINE CARD (FLA, IFX, IWS, NIT, IND)

Subroutine CARD reads one card, performs a left to right scan and assembles the information as a list of numbers, words, and special characters in the array FLA. Numbers are in normal floating-point form and the words and special characters are stored in the appropriate FORTRAN alphanumeric form. Integer array IFX contains indicators enabling the type of the items to be identified and integer array IWS the columns on the card on which successive items terminate. The card is read initially with format (80A1) into an array in COMMON and the text of the card is therefore available to the calling routine as well. This array is left intact during the scan and it is therefore possible to re-assemble the information contained on the last card read. The number of items read is supplied in the integer scalar NIT, and IND takes one of a number of values according to the type of card read—for example, error free, with special characters, blank.

Additional facilities include the reading of numbers to a base other than ten, and the continuation of information onto further cards by the use of a continuation character punched as the last item on each card to be continued. The continuation character is usually $.

## Organisation

The subroutine uses two arrays of integers, known as the *character integers* and the *decision integers*, in the assembly process. The use of these two sets of integers makes the program itself machine independent and to a large extent card-code independent. Each possible character in the computer's vocabulary is allocated to one of ten groups, and associated with each character is an integer in the character integer array ICHAR. In the subroutine described here we assume there are 64 different characters but this number can be easily changed if more characters are possible. The character with internal value I has ICHAR(I + 1) associated with it, and the value of ICHAR(I + 1) contains two pieces of information. The units and tens part is the number of the group to which the character is allocated, and the remaining part is 100 times the value the character is to have when used in assembling items.

For example, on Atlas the character 2 has internal code 18 so that if ICHAR(19) is set to 201 the character 2 is allocated to character group 1 and the numerical value 2 is used in assembling this character as part of a number.

The ten groups used in CARD are defined in **Table 1.**

The card is scanned from left to right and the current state of assembly plays a vital part in determining the processing of the next character in the scan. Seven assembly states are defined as follows:

1. No item started.
2. Word started.
3. Number started but before the decimal point.
4. Number started but after the decimal point.
5. Exponent indicator (usually the character E) read after number.
6. Exponent indicator passed after a number.
7. Exponent started.

The scan always begins in state 1 and if, for example, a letter is read the state moves to state 2. The state remains 2 until a character capable of changing the state is read, that is until a character which cannot form part of a word is read.

The scan section of the program is divided into 21 parts and the path through these parts is decided according to the characters read and the values of the decision integers. The decision integers are stored in a 10 × 7 array JDIS. When a new character is encountered its group is quickly determined from the character integer array. This together with the assembly state determines which decision integer is appropriate. As is seen from Table 1 the value of this integer is the number of the part of the program to be obeyed. Details of the 21 program parts are given below:

**Table 1**
**The standard values of the decision integers**

| ASSEMBLY STATES | DIGITS | LETTERS | EXPONENT INDICATOR | DECIMAL POINT | PLUS SIGN | MINUS SIGN | SPECIAL CHARACTERS | SEPARATORS | CONTINUATION CHARACTER | ILLEGAL CHARACTER |
|---|---|---|---|---|---|---|---|---|---|---|
| | CHARACTER GROUPS | | | | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 3 | 2 | 11 | 4 | 1 | 1 | 18 | 19 | 20 | 21 |
| 2 | 9 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 21 |
| 3 | 7 | 10 | 10 | 5 | 10 | 10 | 10 | 10 | 10 | 21 |
| 4 | 8 | 10 | 10 | 21 | 10 | 10 | 10 | 10 | 10 | 21 |
| 5 | 13 | 12 | 12 | 13 | 13 | 13 | 12 | 13 | 12 | 21 |
| 6 | 14 | 17 | 17 | 17 | 1 | 1 | 17 | 19 | 17 | 21 |
| 7 | 15 | 16 | 16 | 21 | 16 | 16 | 16 | 16 | 16 | 21 |

| *Part* | *State Change* | *Action* |
|---|---|---|
| 1 | | Note the reading of a sign (Character type 5 or 6). |
| 2 | 2 | Start a word—treat a previously read sign as a special character—note in IW the present column on which the word begins. |
| 3 | 3 | Start a number before the decimal point—set AIN equal to the value of the digit. (The number will be built up in AIN.) |
| 4 | 4 | Start a number with a decimal point—set AIN equal to zero—set POW = 1·0. |
| 5 | 4 | Decimal point read whilst in the middle of a number—set POW = 1·0. |
| 6 | | Continue word—do nothing. |
| 7 | | Continue number before decimal point—multiply AIN by BASE and add the value of the current digit. (This operation should be protected by a test for overflow—this test is *machine dependent*.) |
| 8 | | Continue number after decimal point—divide POW by BASE, multiply by value of current digit and add to AIN. |
| 9 | 1* | Terminate word—word begins on column IW and ends on the current column—call subroutine WORD to pack the letters appropriately. |
| 10 | 1* | Terminate number—store AIN taking account of any previously read sign. |
| 11 | 2 or 5 | Exponent indicator read—take as exponent and set state to 5 if following a number—take as beginning a word if following a sign or a word or if it is the first item and set state to 2. |
| 12 | 2 | Take exponent indicator as beginning a word. |
| 13 | 6* | Exponent indicator passed. |

*contd.*

| | | |
|---|---|---|
| 14 | 7 | Begin exponent—set JEX to value of current digit. |
| 15 | | Continue exponent—multiply JEX by IFIXF(BASE) and add the value of the current digit. |
| 16 | 1* | Terminate exponent and number. (This operation should be protected by a test for overflow based on the value of the exponent—this test is *machine dependent*.) |
| 17 | 1* | Store exponent indicator as a one-character word. |
| 18 | 1 | Assemble special character—set IND to 2. |
| 19 | | Ignore character—no action. |
| 20 | | Stop all processing. |
| 21 | | Illegal character read—output diagnostic —set IND to 4. |

Some program parts (marked *) cause the assembly state to be changed and the decision integers re-inspected with the new state. This ensures that characters encountered in certain positions can properly fulfil the two functions of terminating an item and beginning a new item. For example if we punch CAR4 in consecutive columns on a card the characters CAR will contribute to a word and the state will be 2 when the digit 4 is encountered. If the decision integers are set appropriately the action will be to terminate the word and to reset the state to one. Re-inspection of the decision integers at this point ensures that a number is started with the value 4.

The advantages of this fragmentation of the program are:

1. The structure is clear and easily changed or augmented.
2. Assembly rules can be changed by program.
3. The coding is not machine dependent nor does it assume a particular internal character code.
4. By use of an array of integers and the FORTRAN GO TO statement, decisions are taken quickly and the decisions are clearly defined.
5. The subroutine can be easily tailored to particular requirements and unwanted facilities discarded.
6. The same card may be re-processed.

### Changes in the rules

Many examples of changes in the assembly rules will have occurred to the reader already. However, a number of examples are described below to illustrate the flexibility of the subroutine. The flexibility is particularly apparent when we remember that such changes can be made during the execution of the calling program.

Changing the decision integer for state 2, character type 1 from 9 to 6 causes a number encountered immediately after a word to be taken as part of the word. That is, the change would cause the sequence CAR4 to be taken as one word rather than as a word followed by a number. Similarly, a special character can be accepted as part of a word if it immediately follows the word by changing the decision integer for state 2 character type 7 from 9 to 6.

The second example is particularly interesting. A misunderstanding of the use of continuation cards when presenting data cards to a program using CARD was responsible for the inclusion of the character $ at the end of each of a large number of records in card image form on magnetic tape. The simplest change causing these additional characters to be ignored was to re-allocate the character $ to type 8 for the duration of the reading of the data. This change, in fact, necessitated recompilation, but instead access to the decision and character integers could easily be passed on to the user of a program. The resulting gain in program flexibility makes this well worth while.

A final example concerns the use of comment cards presented with the data to a program. To achieve this the comment card must be recognised as such by CARD and the information on the card assembled in alphanumeric form. It was decided that the character * should be punched as the first item on the card to indicate a comment and this character was allocated to group 2. The calling program inspected FLA(1) for every card read by CARD and if this was * the card was printed and otherwise ignored. With these changes a card beginning with * followed by a space would be treated as a comment card and the comment printed.

### Efficiency

The reader's first impression of the efficiency of this approach might be to believe that a lot of work is performed to achieve a relatively simple result. It must be remembered that the work involved in taking the various decisions is quickly performed by a little integer arithmetic and the GO TO statement, and that this is performed instead of the work normally involved in interpreting a Format statement. The integer arrays take up a total of 134 locations but use of these involves a reduction in the size of the program itself which more than compensates. A possible source of inefficiency may be the way in which the particular FORTRAN compiler deals with the initial reading of the card in (80A1) form. Some compilers are more efficient than others in this respect and it might repay efforts to replace the single input statement with a machine-code subroutine which reads a card with this particular format. With this modification considerable improvements in reading speeds over the usual FORTRAN statements have been achieved.

### Reference

COOPER, B. E. (1967). ASCOP—A Statistical Computing Procedure. *J. Royal Stat. Soc.* Series C (Applied Statistics), Vol. 16, No. 2, pp. 100–110.