NEBULA: A Programming Language for Data Processing

By T. G. H. Braunholtz, A. G. Fraser and P. M. Hunt

NEBULA, a programming language for data processing, will be used on the Ferranti Orion Data Processing System. No previous acquaintance with automatic programming languages is assumed in this account of NEBULA. The aim is to give the reader a sense of mastery over NEBULA, rather than to give him every factual detail of the language. However, most of the basic topics are treated fully, otherwise a false impression of imprecision would be conveyed. A complete example is given in the Appendix.

Introduction

There are two stages in preparing a program for a computer. First the job has to be planned in detail, and secondly a computer program to carry out the plan must be written. These tasks demand precision of thought, and frequently a lack of such precision gives rise to teething troubles encountered when the data-processing equipment is being brought into use.

Once the program has been prepared it must be tested thoroughly and any errors removed. Then it can be used for useful work, but in all probability it will still be altered from time to time when minor changes are made in the job to be done.

In the past the cost of this work has often been comparable with the cost of the computing installation, and it has taken many months to bring the programs into satisfactory operation. Efforts by many people to reduce this cost and time have produced an assortment of programming aids, each one contributing a little to making the programmer's task easier. From all this there has emerged the technique of automatic programming, in which the key feature is simplified communication between the programmer and the machine (and, indeed, between one user and another) by the use of comprehensive special programming languages.

In the case of the Ferranti Orion Data Processing System the language used is known as NEBULA (Natural Electronic Business Language). It has a rigid syntax and employs ordinary English words, so that sentences similar to those of the English language can be formed. The use of symbols as well as words allows of brevity.

Any programming system should place a minimum of restriction upon the user, both in the way he organizes his system and in making the most efficient use of his equipment. Nebula has been designed with this in mind: the object program will be efficient, and the form in which data is held on input and output media will not be straightjacketed by presuppositions in Nebula as to the form that data will take.

The language (Nebula) used for communicating with the machine is called the *Source Language*. The user's description of the work to be carried out is written as a sequence of Nebula sentences and is called the *source program*. This is fed into the computer and automatically converted into a program of machine orders known as the *object program*; it is this object program which is eventually used time and time again for productive work. The automatic translation from source to object language is carried out by a routine called the *Nebula Compiler*. The process of compilation is quite distinct from the process to be carried out later when the object program is eventually obeyed. The conversion from source program to object program takes place once only, whereas the object program is obeyed many times, and in fact the two processes need not be carried out on the same computer.

The Advantages of Automatic Programming

Estimates and experience vary, but it may be assumed that by using automatic programming the time spent programming is reduced several fold, perhaps even by a factor of ten. This saves money, staff and time. The saving arises because the compiler solves the computer-oriented problems; the programmer has only to provide the compiler with a description of his data and of the operations to be done on it, and of the form in which he requires his results. For example, a significant proportion of any data processing job consists of programs to read from and write on magnetic tapes, and to move the data into working positions, or into positions from which they can be written on magnetic tape. All of these programs will be created by the compiler without any attention by the programmer.

Using the old methods, the description of a file was embodied in every program using that file in the form of shifts, transfers, selections and so on. Now the programmer need only describe his file once; the compiler creates a directory which it then uses over and over again to create new programs related to that file.

A machine-coded program of any size is not at all easy to understand and fully appreciate, and careful documentation is therefore necessary, even with the simplest of programs. It can in fact take months to produce the documentation required to give a full explanation of a lengthy program. By contrast, programs produced in a systematic manner from source language statements require only a fraction of this documentation. In addition, it is frequently difficult to make amendments reliably and quickly to a machine-language program, particularly if the originator is not available to make them himself. But with a compiler system it is possible

to arrange that corrections to a compiled program can be made in the source language, and that these corrections are effected by the compiler with a minimum amount of recompilation. This means that it is easier to incorporate any modifications necessitated by changing requirements. Indeed this is likely to be one of the most important advantages of automatic programming.

In any installation with a large number of reels of magnetic tape, many of which hold valuable information, a properly organized system of tape identification is essential. To achieve this every compiled object program must check the identity of every magnetic tape before using it, and must also label any newly created output reel. In this way a strict control over the use of tapes is maintained and the risk of destroying valuable information minimized. Originally these problems were considered in detail by the programmer, but now the compiler includes orders in every object program for performing this work.

Just as human beings are fallible, computers also have their sources of error, though of a different type. It is essential that after an electronic or mechanical failure it should be possible to restart the program without going back to the beginning; and preferably the restart should be from a point reached in the last five or ten minutes. Furthermore, there must be no doubt that the program is being restarted from exactly the configuration it had reached at the last restart point, including, for instance, the positioning of cards and magnetic tapes. To provide for this is a tricky task, best handled by a standard procedure. Programs for establishing these restart points are now included in all object programs by the Nebula compiler, and the programmer need no longer be concerned with this aspect of the work.

The Nebula System

Languages are intimately bound up with the ideas and the physical situations that they express and describe, and this is true of Nebula. In the case of Nebula this background has to do with the computer for which a Nebula program is written, and how the programmer conceives of it. All this is referred to as the *Nebula System*. To reiterate the terminology we use, a programmer writes a *Nebula program* in the *Nebula Language* for the *Nebula System*, and this is translated into an object program by the *Nebula Compiler*. It is assumed that the actual computer involved is an Orion, but it could be any computer with adequate facilities.

The Nebula System is to be conceived as a central processing unit with a number of input and output channels attached, as shown in Fig. 1. From this primitive starting point details can be added to the picture bit by bit.

Each input channel is a source of data, and each output channel receives data. This data may be held on punched cards, magnetic tape, or other media, but by the time it is available to the central processor it has been converted to a form that shows no trace of the

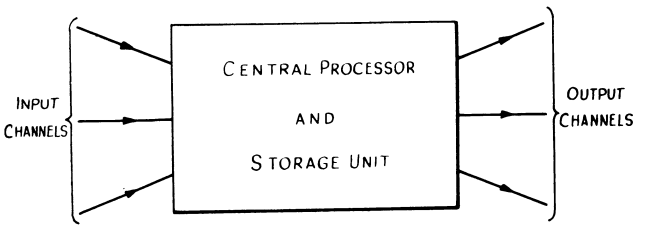


Fig. 1.—Primitive NEBULA system

physical medium on which it was recorded. Each channel is called a *file*, and data is taken in or sent out one *record* at a time. For each program, the files, records and items within records are defined and given names by filling in forms which will be described later. The names may be English words or phrases, or just labels such as "W1."

The operations carried out by the central processor consist of arithmetic manipulations, or comparisons, or transfers, or references to tables, and so on, carried out upon the data held in the central storage unit. These operations will be specified in Nebula by such sentences as

ADD QUANTITY RECEIVED INTO TOTAL STOCK ON HAND,

or

IF AMENDMENT CODE = "D" THEN WRITE MAIN FILE RECORD TO PRINTED OUTPUT FILE.

The central storage unit is used to hold not only records of input and output files but also intermediate quantities and lists of constants. All these must be defined and named in much the same way as the files of data.

The Subdivisions of a Nebula Program

A Nebula program is a description of a system, and using this information the compiler generates a machine-language program that will make Orion behave as the system. Much of this description consists of *procedure statements* such as those given above. In addition the compiler has to be provided with a good deal of purely descriptive information about the data and about the equipment available in the installation on which the compiled object program will eventually run. Consequently, a Nebula program is divided into three parts:

(1) *The Procedure Description*

This is a description of the actions to be performed on the data.

(2) *The Data Description*

This is a specification of the form of the data. As will be apparent later, the data description is subdivided into two parts:

(*a*) A description of the data as it is actually held on

the input and output media (this is called the *physical description*); and

(*b*) A description of the characteristics of the data that affect how it is stored internally in Orion (this is called the *logical description*).

(3) *The Machine Description*

The procedure description, together with the data description, is translated into an object program by the Nebula compiler using an Orion computer. This translation can be performed on any Orion computer, but the particular computer used must be described if efficient use is to be made of its facilities. Furthermore, the object program obtained as a result of translating the source program, has itself to be obeyed on a (possibly different) computer system. This *object computer* has also to be defined to the compiler so that an efficient object program can be obtained.

The description of the two computer systems is called the *machine description*.

A Comment

We have now discussed the advantages of automatic programming, and have introduced the Nebula compiler in broad terms. From now on this article will be taking up particular topics and discussing them in some detail. There is a danger that the reader new to automatic programming will not be able to see the wood for the trees—that he will feel unable to grasp the whole although he understands each individual part. We have tried to prevent this happening by being liberal with introductory remarks and comments on the nature of the problems involved. We have made no attempt to give complete information, for that is taken to be the task of a reference manual, but we have aimed instead to give full details about the most important topics, and to give an idea of what has been omitted.

File Structure and Data Names

We now return to the elaboration of the structure of the Nebula System. This section is concerned with the structure of files, and with naming files and the items of data in them. A little advance information about the procedure description will also have to be given.

It has already been stated that each input or output channel of data is a *file* and must be given a name. Each of these files will be held externally on some medium: punched cards, paper tape, printed paper, or magnetic tape. Regardless of the external medium, the data, on input, is converted and fitted into a framework which we will now describe.

Each file consists of a series of *records*. A record may contain the details of an insurance policy, or of an amendment to it; in general it will contain the data pertaining to one document or to one transaction. An input file is read one record at a time (by a procedure statement such as READ CHANGES FILE), and an output file is written one record at a time (by a procedure statement such as WRITE MAIN FILE). Files can only be worked through from the beginning to the end. Thus every READ statement reads the next record in the file, and there is no possibility of re-reading the previous record. Similarly every WRITE statement places a record in the next position along the file.

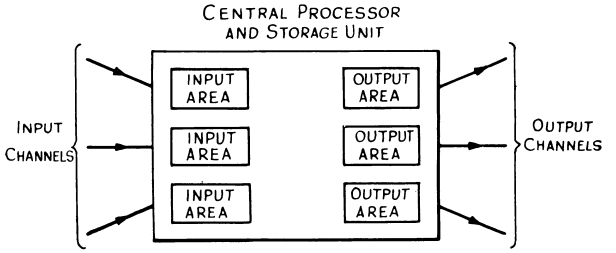


Fig. 2.—Primitive NEBULA system showing Input and Output areas

Associated with each input file is an *input area* which holds one record, and with each output file an *output area* which holds one record, so the Nebula System may now be visualized as shown in Fig. 2.

When a record is read from an input file it is placed in the input area for the file, thus covering over the previous record. The input area is unlike an ordinary computer buffer store, because it is not of a fixed size: a record may have a variable amount of data in it, or there may be more than one type of record in a file, but the input area always holds just one record.

The rules for output files are similar, and once a record is written from the output area on the file, the contents of that record are no longer available to the central processor. As with input files, the output area is not of fixed size, but holds exactly one record regardless of the record's size.

The actual handling of records in the store of Orion does not correspond very closely with this account of the Nebula System's view. The difference arises from the arrangements for efficient use of tape decks, card machines and other peripherals; therefore the reader should not let the question of just how Orion effects this record handling trouble him. However, it can be stated that records will be held in the core store of Orion, and in fact usually at least two records per file will be held. This should be borne in mind when considering how much data to include in a record, although usually in Nebula a record will conform to the usual meaning of the word, containing for instance the data pertaining to one account or transaction.

To return to the structure and naming of files: every record type must have a name, even in files containing only one record type. Every record name and every file name must be different from every other data name of any sort on any of the files in the same program. We have now introduced two kinds of *data names*, namely *file names* and *record names*, and there are two more, *group names* and *detail names*, which we will introduce

shortly. These four classes of names are needed because they have slightly different properties, either in connection with the rules for forming names or in connection with their use in procedure statements.

Every unit of data is stored internally either as a number or as a series of characters which we call a *field*. Each record contains a collection of these numbers and fields. Each number or field is a *detail* and must have a name, a *detail name*. It will often be required to refer to several of these details at a time. This is done by introducing another name, a *group name*, and stating which details it contains. For instance, the three details DAY, MONTH, and YEAR could belong to the group DATE, and then a statement such as COPY DATE INTO NEW DATE would copy all three details.

In many records the same kind of data will occur repeatedly. For instance, in an invoice record there may be the particulars of several transactions. These particulars might consist of the details ITEM NAME, UNIT PRICE, QUANTITY, and PRICE, and be given the group name ITEM DETAILS. If this group occurs several times in a record there must be some way in the procedure description of referring to a particular occurrence. This is done by using *suffices*. We will not give the full details on suffices, but simply say that the particular occurrence of a group or detail is specified by an integer enclosed in brackets after the name. For example, the successive occurrences of ITEM DETAILS are referred to in the procedure description as ITEM DETAILS (0), ITEM DETAILS (1), ITEM DETAILS (2), etc. More usefully one may write, for example, ITEM DETAILS (COUNTER) where COUNTER is a detail used for counting through the occurrences.

The structure of files has now been explained. To summarize: the structure and naming of a file is defined if the file is given a name, and then all the record types in the file are given names, and then for each record all the groups and details in it are given names, and the details contained in each group stated. This is done in the *file outline*, which is specified on the left-hand side of all the data description forms, and is described later.

We turn now to the rules for forming names. The detail names and group names within a record must all be different, but the same detail (or group) name is allowed to occur in different record types. However, if it does occur in more than one record type, then, whenever one of the details with this name is referred to, the record it is in must also be named. To illustrate this we indicate the form of an expression in a way that will be used frequently in the rest of this article. The full names of the details under discussion take the form

detail name IN record name

The parts of this expression in capitals are meant literally, i.e. these words will actually occur in the expression; whereas the phrases in small letters state what kind of thing is to appear in their place in the expression. Thus an example of the form of name just described might be

DATE IN INVOICE RECORD.

The rules for forming a data name are simple:

(1) A data name may contain any number of words:
(2) The characters in the words must be letters or digits;
(3) The first character of the first word must be a letter;
(4) No distinctions are made between upper case and lower case letters;
(5) There are nine words that may not be used in data names, these words being AND, BY, FOR, IN, INTO, OR, OTHERWISE, THEN, and TO.

Perhaps it should be added here that no distinction is made between upper case and lower case letters in any part of a Nebula program whatsoever.

Finally, before leaving the topic of files, we must mention *working files*, which provide the means of naming and describing intermediate results and working data, as well as lists of constants. These files are rather different from ordinary files, but as with ordinary files contain a number of records. They are stored on the magnetic drums or in the core store of Orion; and, in general, those records containing working data currently in use are held in the core store, and the remainder are held on the magnetic drums.

The Procedure Description

We have given a detailed explanation of the organization and naming of the data used in a Nebula program, although leaving until later the question of conversion from external media to internal form. We will now consider the procedure description, which describes the processing to be done on the data when it is in internal form.

There are about twenty *statements* that can be used in the procedure description. Every statement has a strict format or set of alternative formats. As an example, one of the alternatives for the ADD statement is described by

ADD arithmetic expression INTO detail name.

Each of the statements begins with the word that identifies the statement, such as ADD, and this is followed by several different formats for the various statements.

The term "arithmetic expression," used in the description of the format, has yet to be defined. There is a precise set of rules in Nebula for constructing an arithmetic expression, but for practical purposes it is enough to say that any formula such as

(PRICE − DISCOUNT * FACTOR) * TAX RATE + BASIC CHARGE

is an arithmetic expression. The variables in arithmetic expressions must be data names, and the symbols + − * and / must be used for the arithmetic opera-

tions. Numbers may be included, the ways in which they may be written here or elsewhere in Nebula being defined exactly. Approximately, any sensibly-written decimal number is allowed, as well as other forms. Brackets may be used in an arithmetic expression, as in the example above. Arithmetic expressions in Nebula programs are evaluated according to the normal rules.

Note that a particularly simple case of an arithmetic expression is a detail name. Thus

PRICE

is an arithmetic expression. Frequently, where an arithmetic expression is permitted in a statement, there will be in practice just a detail name.

Every word that is written in the procedure description must conform perfectly with the rules; that is, every word must be part of one of the allowed constructions, which we are about to describe, and also every word must be spelt correctly. The same is true, in fact, of every part of a Nebula program.

It is nearly true to say that the procedure description simply consists of a series of procedure statements, each one terminated by a full stop. They are obeyed (or, rather, the corresponding sequences of machine orders in the object program are obeyed) in the order in which they are written, except after the GO TO and PERFORM statements which are provided so as to be able to break this sequence.

It is quite true to say that the procedure description consists of a series of *sentences*. A sentence is always terminated by a full stop. It can consist of just one statement, or it may contain a series of statements each linked to the next by the word THEN.

A sentence can begin with a *label*, which is enclosed in square brackets, thus:

[label] statement THEN statement . . .

The label is formed in the same way as data names, but it does not have to be different from all the data names, although all the labels in a program must be different from each other. The label is used by the GO TO statement, which has the form

GO TO label

and evidently something like a label is required so that the GO TO statement can specify which sentence to obey next.

There is an important kind of sentence beginning with the word IF. Such sentences are called *conditional sentences*, and are necessary in order to be able to vary the action of the program according to circumstances. The form of a conditional sentence is

IF condition THEN ordinary sentence

and the ordinary sentence can as usual contain any number of statements linked by THEN; if the condition holds true then all the statements are obeyed. The term "condition" must be explained. There are formal rules for constructing a condition, but again for practical purposes a condition consists of any two arithmetic expressions related by one of the symbols $= \neq > \not> < \not< $ (where $\neq$ means "not equal to," etc.). Thus

PART NUMBER IN MOVEMENTS FILE
= PART NUMBER IN MAIN FILE

is a condition.

A condition is said to hold true if, when the two arithmetic expressions are evaluated, the relation in the condition is true.

Conditions may be combined into more complicated conditions by the words AND and OR. An example is

A + B > C * D AND E ≠ F

Notice that the first condition must be read as (A + B) > (C * D), and not A + (B > C) * D, which does not make sense. In other words, arithmetic expressions must be evaluated first, and then relations (i.e. = etc.), and finally ANDs followed by ORs.

In addition, the word OTHERWISE may be used to link any number of conditional sentences together, giving a complete sentence of the form

IF condition THEN statement . . . OTHERWISE
IF condition THEN statement . . . OTHERWISE
statement . . .

The meaning of this sentence will be as in ordinary English; but it is easy to construct sentences which are perfectly correct and yet quite incomprehensible, through being too long, in which case they should be broken into smaller sentences. An example of the satisfactory use of OTHERWISE clauses is the sentence

IF A = 0 THEN ADD 5 INTO D
OTHERWISE IF A = 1 THEN ADD 2 INTO D
OTHERWISE CLEAR D.

(D will only be cleared if A is not equal to 0 or 1.)

A Digression on the Appearance of Nebula Programs

It may not be clear how, given a blank piece of paper, the procedure description should be recorded on it, or just what variations are permitted, or how it is that the Nebula compiler is able to interpret the punched paper tape corresponding to the printed form of the program.

Nebula programs will be prepared on a Flexowriter which will produce a paper tape as a by-product of typing, and this paper tape has a character punched in it every time a key is depressed.

It is a principle of Nebula that the printed form of a program is what counts, and that if it appears to make sense to a human being then, so far as possible, it should make sense to the compiler too.

For most of a Nebula program the exact position of data on the page does not matter to the compiler. The compiler reads the information rather as one reads a book: the break between lines is not significant, but the order of lines and the order of information along a line is. The rule is that between words, even between a pair of words constituting a data name, one may type

any number of spaces and one new line, but a word may not be split. In Nebula, as in English, the largest unit of the language is the sentence. After the end of a sentence one may type any number of spaces and new lines. The data description forms are an exception to the rule that position across a line is not significant, as will be clear when they are explained.

Although the programmer is not compelled to lay out his program neatly, he will usually wish to do so and may arrange margins and paragraphs and headings as he wishes. He will, for instance, want to type labels at the left edge of a page, with the actual procedure statements indented a few characters.

The major sections of a Nebula program will be introduced and terminated by statements such as PROCEDURE DESCRIPTION, and END OF PROCEDURE DESCRIPTION, and DATA DESCRIPTION, and within the data description CARD DESCRIPTION, and so on. It is evident that these statements can be recognized like any other statement.

A full description of how statements and such things as IF clauses are analysed would be very lengthy, and a full statement of how incorrect forms of expression and misspellings are recognized and dealt with would be even lengthier; suffice it to state that the language is so chosen that these analyses can be done.

Procedure Statements

The number of genuinely different statements absolutely required for programming is perhaps four: one for reading or writing records to or from files, one for doing arithmetic and transferring data internally, a third to move control from one part of a program to another (such as the GO TO statement, which, with IF clauses, would also allow action dependent on the circumstances), and a fourth to stop the program. These statements would of course rely heavily on the expressions that occur in them, such as arithmetic expressions, or outside them, such as conditions and labels, to give variety of expression.

At any rate, the point is made that a small number of statements, together with a few forms of expression, are able to describe data-processing operations. The greater number of statements in Nebula are provided partly to allow efficient use of the computer and partly to provide more convenient means of expression.

We shall now discuss each of the Nebula procedure statements, mentioning most of them only briefly, but treating some of them fully in order to give an idea of how many questions they raise and how much explanation they require. The procedure statements may be classified under the headings Input and Output, Arithmetic, Control, and Miscellaneous, corresponding approximately to the four essential statements mentioned above.

Input and Output Statements

There are six input and output statements: OPEN, CLOSE, ACCEPT, DISPLAY, READ, and WRITE.

The OPEN and CLOSE statements are used, respectively, to bring a file into use and to remove it from use. They cause various routine operations to be carried out, such as checking the identity of the tapes on opening a magnetic-tape file, and placing an end-of-file marker on the tape and rewinding on closing a file. These statements are not absolutely essential, since Nebula could arrange that every file in a program is opened automatically when the program is begun and closed when it finishes. But this, though perhaps usually satisfactory, could waste time, or even require extra input or output machinery. (This latter situation would occur when two files could use the same peripheral because they were never in use simultaneously.)

The ACCEPT statement is used to receive messages typed by the operator on the monitoring Flexowriter, and the DISPLAY statement is used to print messages for the operator on this Flexowriter. For instance, the following DISPLAY statement

DISPLAY "END OF STAGE 1"

will cause the words between quotation marks to be printed on the monitoring Flexowriter.

The main features of the READ and WRITE statements were explained in the section on file structure, and the explanation will not be repeated here. But there are two problems arising with the READ statement, and we will discuss them in detail. The READ statement takes one of the forms

READ file name

or

READ record name

The second alternative may only be used when there is only one type of record in the file.

The first problem arises when there is more than one record type on a file, so that it is not known in general what type of record has been read in by a READ statement. The procedure description cannot refer to details in the record until it knows what type of record it is, and on the other hand it cannot discover the type of the record without referring to some identifying detail in the record. This dilemma is resolved by a special dispensation, which is as follows. If an input file has more than one record type, it is suggested that each record type contain a detail with the same name and properties. In this special case the detail may be referred to without its name being followed by "IN record name" as would normally be required to remove ambiguity. Then, if the detail contains a different value for each record type, the record can be identified, and the problem is solved.

The second problem is concerned with recognizing the end of a file. The difficulty is that the end of files is indicated by a marker, not a record, put on automatically by the conversion programs, and this marker cannot be read as a record by a READ order, so how is it to be sensed? The solution is provided by another special facility. A new form of conditional expression

is introduced:

IF END OF file name

which can be used to test for the end of the file before giving a READ statement. An example of its use is

IF END OF MAIN FILE THEN CLOSE THEN STOP OTHERWISE READ MAIN FILE.

Arithmetic Statements

The eight arithmetic statements are ADD, SUBTRACT, MULTIPLY, DIVIDE, COPY, CLEAR, COMPUTE, and TAKE. The first five of these all have the same construction as the ADD statement. For instance (one form of) the MULTIPLY statement is described by

MULTIPLY arithmetic expression INTO detail name.

To make the meaning of this statement clear,

MULTIPLY PRICE INTO QUANTITY

will multiply PRICE by QUANTITY and place the result in QUANTITY. Similarly

DIVIDE P INTO Q

will place Q/P in Q.

There are alternative forms for these statements, which are useful for making Nebula programs read naturally. These involve a special detail named QUANTITY ON HAND, or QH for short. Then statements such as

ADD arithmetic expression

MULTIPLY BY arithmetic expression

TAKE arithmetic expression

COMPUTE arithmetic expression

all leave the result in QH, with the ADD and MULTIPLY statements adding and multiplying what was already in the QH by the arithmetic expression. (Notice the BY in this form of the MULTIPLY statement.)

There is yet a third form of these statements, as illustrated by

ADD INTO detail name

Here the QH is added to the detail and the result placed in the detail. With these forms one can write such sentences as

TAKE GROSS CAPITAL THEN SUBTRACT DEDUCTIONS THEN MULTIPLY BY INTEREST RATE THEN COPY INTO INTEREST.

Control Statements

The four control statements are GO TO, PERFORM, EXIT, and STOP. The GO TO statement was explained when labels were introduced. The PERFORM statement is used to enter a *procedure*, which is roughly the same as a subroutine in ordinary computer terminology. Without going into the matter fully, it can be said that

File Outline			Logical Description				
Level		Name	Numeric		Non-numeric	Position	Other Details
			Min	Maximum			
1		Invoice File					
· 1		Invoice					
· ·	1	Personal Details					
· ·	· 1	Initials and Title			12		
· ·	· 2	Surname			V		
· ·	· 3	Line 1 of Address			V		
· ·	· 4	Line 2 of Address			V		
· ·	· 5	Line 3 of Address			V		
· ·	2	Invoice Details(ID)					Repeat Variable.
· ·	· 1	Quantity	0	999			
· ·	· 2	Description			V		
· ·	· 3	Price	£0	£10			
· ·	· 4	Amount	£0	£10 000			
· ·	3	Total	£0	£100 000			

Fig. 3.—The Logical Description Form, filled in for a simple Invoice File

the PERFORM statement causes the program to return to the statement following it when the procedure has been completed, whereas the GO TO statement has no such effect. The EXIT statement is used in procedures to cause this return. Lastly, the STOP statement terminates a program.

Miscellaneous Statements

The other statements of the procedure description are the LOCATE, ROUND OFF, ROUND UP, and TRUNCATE statements. The LOCATE statement is for use in searching lists or files, and the other three are for rounding or truncating numbers.

The File Outline and Logical Description

Fig. 3 shows the logical description form, filled in for an invoice file containing one record type named "Invoice." The part of the form to the left of the double line is called the File Outline. The file outline occurs on all the data description forms.

The File Outline

The file outline describes the structure and naming of a file. The entries in the Level columns indicate the structure, by means of the dots (full stops) before the numbers in the entries, which are interpreted as follows.

The level entries for files have no dots. Thus if a second file were described on the form, its level entry would be just the digit 2. The level entries for records have one dot, such as that for "Invoice." If there were

another record in the file its level entry would be .2. The entries for items within a record have at least two dots, and these entries define the group structure of a record. In the example, Personal Details and Invoice Details are group names, and all the other names are detail names. The details Initials and Title, Surname, and the three lines of address are members of the group Personal Details; and Quantity, Description, and Price are members of the group Invoice Details. The rule is thus: that an entry followed by one with a larger number of dots is a group name, and any entry followed by one with the same number of, or fewer, dots is a detail name. Groups within groups can be defined in this way, to as many levels as may be required.

Abbreviations for names can be defined in the file outline. The abbreviation can be given in brackets after the name, as in the example where the abbreviation ID for Invoice Details is defined. These abbreviations can be used whenever desired, instead of the full name, in the procedure description. More than one abbreviation may be given; and perhaps it should be stated that the "abbreviation" does not have to be shorter than the main name.

If there were not space in the Name column on the same line, the abbreviation could be given on the next line. This way of overcoming shortage of space applies to all the columns of all the data description forms: if there is not enough space, an entry may be continued on the succeeding lines of the same column. The next full entry on the form will then be entered on the line after the last of these overflow lines.

The Logical Description

The logical description contains the information required by the compiler to allocate storage space, and to keep track of scaling in arithmetic operations.

Every detail is either *numeric* or *non-numeric*. Very little has been said so far about the properties of these two classes of detail. To discuss numeric details thoroughly would involve discussing the scaling of numbers, but there is no need to do so in this article. Numeric details are numbers, and so arithmetic can be done on them. Non-numeric details correspond to what are usually called alpha or alpha-numeric fields in punched-card work, consisting of a series of characters. These are six-bit characters, and so can take 64 values, allowing a character set of 64 characters.

The columns headed NUMERIC MIN and NUMERIC MAXIMUM must be filled in for numeric details. The entry in the MIN column must be the minimum value the detail can take (the most negative value if the number can be negative), and the entry in the MAXIMUM column must be the maximum value the detail can take. These entries allow the compiler to allocate space to the detail, and also allow it to put orders into the program to check that it is within range.

The column headed NON-NUMERIC must be filled in for non-numeric details. If it is of fixed length, this length, in characters, should be entered. If its length is variable, the letter V must be entered.

All the other information required in the logical description is entered as statements in the Other Details column. These statements have a strict structure, along the same lines as those for the procedure description, although naturally a different set of statements is provided. There are about twenty of these statements, and as with the procedure statements, many of these are not essential but are provided for the programmer's convenience. We will mention the most necessary ones very briefly.

A group or detail that occurs repeatedly has entered against it the statement

REPEAT integer TIMES

if it occurs a fixed number of times, or

REPEAT VARIABLE

if the number of occurrences is variable. Naturally, even if the number of occurrences is variable, it must have a limit. Nebula contains provision for specifying the limits on record lengths. REPEAT VARIABLE includes no occurrences as a possibility.

A group or detail that may or may not occur, but is not repeated, has

OPTIONAL

entered against it.

Working files or constants files are identified as such by the statements WORKING FILE or CONSTANTS FILE entered against the file name.

The accuracy to which a numeric quantity is held is not deducible from the MAX and MIN entries. But unless otherwise specified, integers will be stored to unit accuracy, and sterling quantities to units of one penny. If other units are required, and in any case for fractions, they are specified by the UNITS statement.

The Physical Description

There are three different ways in Nebula of describing the data recorded on external media. These are the *Card Description*, the *Printing Description*, and the *Paper Tape Description*. It turns out that the description of a file is the same whether the file is an input or output file, with the exception that some information needed for input files is not needed for output. Therefore the same method of description is used for describing input and output files on each medium.

The descriptions of the files on the different media are necessarily different. For instance, comparing punched cards with printing, the natural way of describing the position of a detail on a card is to state the columns occupied by the detail, whereas the position of a detail on a page is described by line number and character position along the line. To take another example, with punched cards some provision must be made for describing single hole positions used to represent +

and —; but printing and paper tape have nothing similar. On the other hand, on printed forms but not on cards it must be possible to describe where format symbols such as £ and , are to be inserted in the print-out of details.

A form and a set of statements are provided for each of the three physical descriptions, along the same lines as for the logical description. As with the logical description, the left-hand side of the forms contains the file outline. It has been stated that the file outlines for the logical and physical descriptions of a file must be the same, but this needs a word of explanation. The file structures and names given in the two file outlines must be identical, but the exact positioning of information on the forms is not significant. Abbreviations for names must not appear on the physical description. A file might be the third file described in the logical description, but the first one in the printing description, and then the level numbers for the file would be 3 and 1 respectively: this does not matter.

The right-hand side of a physical description form contains the complete specification of how data is recorded on the external media.

It will be necessary when explaining some of the card and printing statements to introduce *literals*, which are of great use in Nebula. Literals are numbers or words—at any rate a series of characters—written in Nebula programs, and in some sense intended literally. Thus the statement

ADD 127 INTO QUANTITY

will add 127 into the detail called QUANTITY. Similarly

COPY "HENRY" INTO NAME

will place the characters HENRY in the non-numeric detail called NAME.

Literals can appear in arithmetic expressions instead of detail names, and in many other expressions in Nebula. The rules for writing literals are simple. A literal must be enclosed in quotes (e.g. "HENRY"), or, alternatively, every character of the literal must be underlined (e.g. HENRY). The only exception is that a numeric literal need not be underlined or enquoted. There is a list of the forms that a numeric literal may take, but a few examples will suffice here: £17.3.6 or £35 or + 104 or 23.7 or .001. (Sterling quantities are identified by the £ sign.)

The Card Description

Before explaining the card description it will be well to outline the nature of the card files and how they are handled.

Usually, but not necessarily, each card will contain one record. Each card will contain full identifying information. If there is more than one record type in the file, a field, in the same position on each type of card, will be punched with the identifying code. It is

FILE OUTLINE		CARD DESCRIPTION				
LEVEL	NAME	CHAR TYPE	ZERO S	ZERO NS	POSITION	OTHER DETAILS
..1	PRICE	L	0	0	16	
		L	0	0	17	
		L	0	0	18	
		S				OVERPUNCH 19/B = 10.
		S	0	0	19	
		D	0	0	20	OVERPUNCH 19/A=10;19/B=11

Fig. 4.—The Card Description Form

expected that cards will be used "fixed format," that is to say that for each type of card the meaning of the data punched in a particular column is always the same. The end of card files will be indicated by specially punched cards, it might be, for instance, by a part number of 99999. Reloading readers and emptying stackers will be done on Orion without program intervention. Misreads or mispunching, and card wrecks and other failures will be dealt with by standard procedures, so that there is no need to mention them in the card description.

Any card code can be used with Nebula and Orion. Standard card codes are given names and stored along with the compiler, so that usually the Nebula programmer will not have to describe his card codes.

Non-standard punching, in the upper or lower curtate, is assumed to be common, and it must be possible to describe it reasonably conveniently.

Fig. 4 shows the card description form and a typical entry for a detail called price. As this entry shows, each line of the form describes one digit of the detail. The most significant digit is described on the first line, and successive digits on successive lines. Similarly for alphanumeric data, one character is described on each line and the leading character is described first.

The entry in the first column, headed Char. Type (for Character Type), is always the letter A for alphanumeric characters; for numeric items the entry gives information about the significance of digits. In this latter case the possible entries are N, F, L, S or D. For sterling quantities L, S and D are used to identify the pounds, shillings and pence digits, as shown in the example. In ordinary decimal numbers N and F are used, N for digits to the left of the decimal point ("integer digits"), and F for digits to the right ("fraction digits").

The entries in the next two columns, headed Zero S and Zero NS (significant and non-significant zero), specify the punching of zeros, and are therefore only used when describing digits. The entry in these columns is usually either 0 in both, if zeros are to be punched, or the columns are left blank if zero is to be represented by an unpunched column. If any other characters are required, they should be entered in the columns.

The entry in the column headed Position gives the number of the card column holding the digit or character

being described. If several successive columns of the card have the same description, which will happen very often, they can all be described on one line by another kind of entry in the Position column. An illustration is afforded by the three L digits in the example in Fig. 4, and in this case the entry would be

$$16 \rightarrow 18$$

All the other information about a card file is given by statements in the Other Details column.

The reader may wonder why some information in the logical and data descriptions is given by entries in columns, and some by statements: all the information could in fact be provided in either way. However, the aim has been to reduce the amount of writing by providing columns for the information which is required most frequently. On the other hand, statements are preferred in general because their meaning is more self-evident, and because every extra column means more spacing or tabbing across the form. Also, some of the information varies in quantity, which does not matter with statements but does with column entries.

Before giving a detailed description of two of the card description statements, we mention all the statements briefly. Some are more essential than others. The OVERPUNCH statement, and the associated statements NO OVERPUNCH and SINGLE PUNCH are essential for describing non-standard card punching. The RECORD IDENTIFICATION statement is necessary for identifying records in files with more than one record type, and the END OF FILE statement is necessary for detecting when the end of the file has been reached. The STANDARD CODE statement is necessary for naming the card code used. The card machine to be used must be named either in the card description by the CARD MACHINE statement, or in the object machine description. The ROUND OFF, ROUND UP, TRUNCATE and USE TABLE statements serve the same purposes as in the logical description, and are convenient rather than necessary. There are other statements for handling multi-card records and variable field-length data; for saving duplication of the description of identical records or parts of records; and for the esoteric purpose of interpreting columns with deliberate double-punching in the lower curtate.

We shall describe in detail the two most complicated, and probably most used, statements, the RECORD IDENTIFICATION and OVERPUNCH statements.

Consider a card file with several different record types, with every record corresponding to a card, and with different layout of fields for the different card types. Before each card can be converted its type must be determined. Usually the type will be indicated by the punching in a field of one or two columns, these columns being in the same position on all the card types. If this is so, it is only necessary to know where on the card this field is and what punching it contains for each record type in order to identify the card type. The

FILE OUTLINE		CARD DESCRIPTION				
LEVEL	NAME	CHAR TYPE	ZERO S	ZERO NS	POSITION	OTHER DETAILS
. . . 1	MONTH	N9			38	OVERPUNCH 38/A=10;38/B=11; 38/0=12. SINGLE PUNCH.

Fig. 5.—The use of the OVERPUNCH statement to supplement a standard digit

RECORD IDENTIFICATION statement gives just this information. It takes the form

RECORD IDENTIFICATION condition

and an example might be

RECORD IDENTIFICATION CODE = "AB"

entered against the record name. Here CODE is the identifying detail, and it takes the value AB for the record type this statement is entered against.

We have described the most straightforward record identification situation, although the statement will apply under other circumstances, for example when the identifying field is not in the same columns for all record types.

All non-standard punchings of digits or characters are described by the OVERPUNCH and associated statements. The form of the OVERPUNCH statement is

OVERPUNCH hole position = literal; hole position = literal....

Thus if the category to which some article belongs is punched in positions 0 or 1 of column 7 of a card, and these categories are referred to by the procedure description as A or B, then the OVERPUNCH statement against the character holding the category will be

OVERPUNCH 7/0 = "A"; 7/1 = "B"

If in addition a category C is represented by neither of these positions being punched, then the statement

NO OVERPUNCH = "C"

would be made as well. On the other hand, if it were an error for there to be no punching in either position, then the statement

SINGLE PUNCH

would be made as well.

The OVERPUNCH statement can be used to supplement the description of a standard digit. For instance, if the month is punched in column 38 of a card with holes in the A, B, and 0 positions representing months 10, 11 and 12 respectively, then the character holding months would be described as in Fig. 5. In such cases, where there is an entry in the Position column as well as an OVERPUNCH statement, the value given to the digit formed by the OVERPUNCH statement is added to the value derived from the standard punching.

Incidentally, the 9 in the Character Type column is a special notation which states that the standard digit is contained in the bottom *nine* positions of the column, not ten as usual, with no punching in these nine positions representing zero. The SINGLE PUNCH statement has been made because there should be exactly one hole punched in column 38, taking the standard digit and the overpunch positions together.

The OVERPUNCH statement has required more explanation than one might wish, but in fact this has no more than reflected the variability in the special punchings which it has to describe. There remain a few fine points of interpretation and extension of notation which we will not discuss.

The Printing Description

Fig. 6 shows the printing description form and a typical entry for a sterling quantity called TOTAL CHARGE.

Before explaining the printing description we discuss what is required of it, and how this differs from what was required for the card description. There are two aspects of card files which do not occur with printing. One is record (i.e. card type) identification, which does not arise because printing is a type of output, and the record identification problem is one which only arises with input files. The other aspect is that nothing corresponding to overpunching occurs with printing. On the other hand, data is held in fixed format on cards, whereas the position of a detail on a printed form may vary. For instance the position of a person's surname on a form depends on his title and how many initials he has. With printing, too, more is needed for zero suppression, for inserting format characters into the print-out of details, for headings to data, and for other such refinements needed to make the printed output easily read.

The main job of describing printed output is done by the entries in the Position column which state where the details are to be printed, and by the MAP and ZERO statements in the Other Details column, which state what characters are to be printed for each detail. Notice that in the card description, each character is described on one line of the form, whereas in this description it is each detail that is described on a line.

Before describing the position entry and the MAP and ZERO statements in detail, we comment briefly on the other statements. There are the POSITIVE and NEGATIVE statements for indicating the sign of a number other than by + and −; for instance, it might be by printing CR or DR after the number. There is the TEXT statement for printing headings and other fixed words. There are the usual rounding statements. There are the FORM LENGTH and PRINTING LIMIT statements for specifying the length of the form and the number of lines available for printing, and the FORM FEED statement for stating that a form feed character sent to the printer will move the paper to the

FILE OUTLINE		PRINTING DESCRIPTION	
LEVEL	NAME	POSITION	OTHER DETAILS
2	TOTAL CHARGE	2/→60	MAP=£LLL÷SS÷D: ZERO=£$$0÷$0÷0: NEGATIVE= DR IN 2/→62.

Fig. 6.—The Printing Description Form

beginning of the next form. And there are statements for handling continuation forms, for naming the printer used, and for other purposes.

To explain the use of the position column, let us take the simplest entries first. These are for the case in which the detail is to go in a fixed position. They are best illustrated by examples:

$$4/10 \rightarrow$$

or

$$4/ \rightarrow 10$$

The first states that the detail is to be printed on line 4 with its left-hand character in character position 10; the second means the same except that the rightmost character of the detail is to be in character position 10. The first is what is usually required for alphabetic information and is called *left justification*, and the second is for numbers and is called *right justification*. The only case where no arrow is required is where the detail consists of only one digit or character.

Now consider specifying the positions of a repeated group of details, such as a list of invoice details. Here the occurrences of the group of details are to be printed consecutively down the page, each occurrence beginning on a new line, and the line on which this listing is to start is known. Usually each occurrence will occupy a fixed number of lines, but the notation to be described does not demand this. Fig. 7 shows how such situations are described. The line on which printing starts is entered against the group name: here it is line 10. The first occurrence of detail A will be printed on line $10 + 0 = 10$, and in character positions $15 \rightarrow$, and similarly B and C will be printed at $10/ \rightarrow 25$ and $11/12 \rightarrow$. Then, since this first occurrence of the group used lines 10 and 11, the next occurrence will start on

NAME	POSITION
INVOICE DETAILS	10
A	+0/15→
B	+0/→25
C	+1/12→

Fig. 7.—Use of "+" to describe positions of details of a repeated group

NAME	POSITION	OTHER DETAILS
INVOICE DETAILS	10	LAST LINE = A.
A	+0/15→	
B	+0/→25	
C	+1/12→	
TOTAL	A+1/→25	

Fig. 8.—Use of the LAST LINE statement

line 12. On this occasion detail A will be printed on line 12 + 0 = 12, and so on.

Suppose it were required to print a total on the line below the last line occupied by the listing. It must be possible to refer to the last line occupied by the listing. The LAST LINE statement is provided for this purpose. The previous example would then be augmented as shown in Fig. 8. Note that the + in the A + 1 is used in the ordinary way, as distinct from its use as a "relativizer" against details A, B and C. It should be added that the LAST LINE statement has the form

LAST LINE = label

where the label in practice will be a letter, and must differ from any other labels used in the physical description of the same record.

The MAP statement describes the printing of a detail, and takes the form

MAP = format description

This format description is a character by character representation of the printed detail. In this representation, L, S, D, N and F stand as usual for a pound, shillings, pence, integer, and fraction digit, respectively. Format characters such as decimal point or commas are underlined. Examples will make the interpretation of the MAP statement clear:

An example of

MAP = N, NNN.FF

is 4,719.32

An example of

MAP = £LL.SS.D

is £35.15.7

There are other symbols with special meanings in the MAP statement. "A" represents any character of the printing code, and is used for alphanumeric data. "+" is only used with numeric details and represents + if the number is positive and − if the number is negative. "−" has the same meaning as + except that a space is printed if the number is positive. There are a few other characters with special meanings. Lastly, the notation A... represents any number of characters and is intended for describing variable-length fields.

Any string of characters belonging to the code is an example of MAP = A....

Examples of MAP = + NN are +37 and − 04.
Examples of MAP = − NN are 37 and − 04.

These details of the MAP statement suffice for its use in the printing description. It is also used for the paper tape description, where a little more is required of it.

The ZERO statement supplements the MAP statement if any zero suppression is required. The ZERO statement evidently can only be made against numeric details. It takes the form

ZERO = format description

This format description must contain the same number of characters as the format description in the MAP statement it goes with, and each character of the ZERO statement is simply the character to be printed if the corresponding character of a number being printed is not significant. Usually there is no difficulty in deciding whether a digit is significant in the ordinary sense of the term, but there is a precise rule in Nebula for deciding in special cases. Again, examples will clarify the situation.

Examples of

MAP = N , N N N . F F
ZERO = $ $ $ $ 0 . 0 $

where $ represents the space character, are

13.7 rather than 0, 013.70

and 0.0 rather than 0,000.00

An example of

MAP = £ L L . S S . D
ZERO = £ − 0 . $ 0 . 0

is £−3.2.0 rather than £03.02.0

Conclusion

The paper tape description is similar to the card and printing descriptions, many facilities being very like those provided in one or the other.

We have given no description of a great many of the facilities provided in Nebula. Some of these are: how to describe card, printer, and paper tape codes; how to cope with faulty data on cards and paper tape; the Object Machine Description, which states such things as the amount of core store and drum store available, the number of tape decks and the number and names of the other peripherals; the specification of restart procedures; the specification of tables; the use of subroutines; and any specification that may be desirable concerning the sizes of the files, or the maximum length of records. But these facilities all use the same kind of statements and principles of construction as those parts of Nebula that have been explained.

The purpose of this article has been to make clear how programming with the Nebula language is done, and in parallel to give some understanding of how the language has been constructed; and for those who wish to use Nebula the aim has been that they should understand enough to be able to write programs in Nebula with the aid of the programming manual and the summarized procedure sheet.

The authors are indebted to Ferranti Ltd. for permission to publish this paper.

Appendix

A Stock Control Application

A warehouse has in stock a large number of different items. Each item is given a part number. Each time a quantity of an item is issued from stock an issue card is made out. Similarly, each time a new batch of items is brought in a receipt card is made out. These cards are punched cards to be used by an Orion computer system. The fields on the cards are as follows:

(*a*) A single character I for issue card and R for receipt.
(*b*) A part number.
(*c*) The quantity of stock issued or received.

A master file is kept on magnetic tape which contains a record of all the items in stock. For each item is recorded against its part number, the quantity currently on hand. Also kept with each record is a minimum below which the quantity on hand should never fall.

Each day it is intended to pass the cards, sorted into part number order, and the master file, through the computer and to produce as a result a new, updated master file. Some printed information is also required:

Transaction Records

These are summaries for each stock for which there was a receipt or issue card. Against the part number is printed the total quantity received and the total quantity issued.

Requirement Slips

These are print-outs giving the number of any stock for which the quantity on hand is below the specified minimum for that stock. Also printed is the deficiency.

The receipt and issue cards will constitute a file called the Movements File and the cards themselves will generally be referred to as Movement Records. The Total fields in the transaction records will be used to contain the intermediate totals of issues and receipts.

The part number will occupy the first twelve binary digits of the forty-eight digits in one Orion word.

One possible flow diagram for the required program is given below, and it is followed by the appropriate Logical Data Description and Procedure description.

The layout of items within the store of the computer will only be specified in the case of the stock records on the master file. The layout of the remaining items will be left to the compiler. A stock record will be contained in two computer words as indicated below:

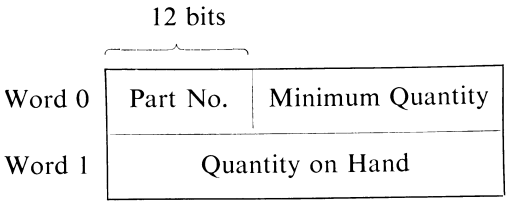
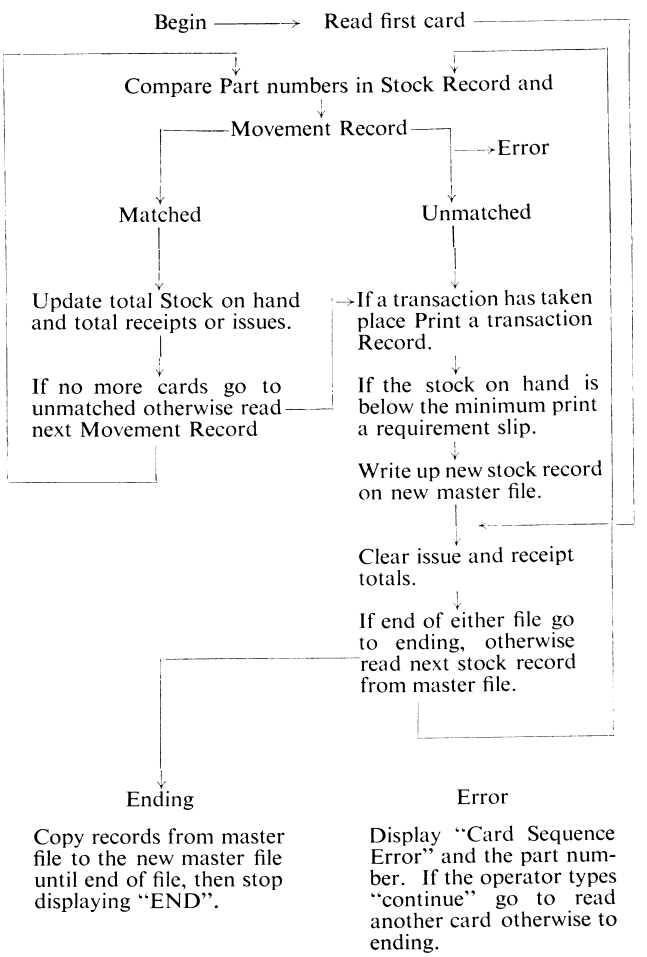


Fig. 9.—Arrangement of stock record words

Flow Diagram



FILE OUTLINE			LOGICAL DESCRIPTION				
LEVEL		NAME	NUMERIC MIN.	NUMERIC MAX.	NON-NUMERIC LENGTH	POSITION	OTHER DETAILS
1		Master File (MF)					
. 1		Stock Record					
. .	1	Part No.	0	4000		0/0	
. .	2	Quantity	0	100000		1	
. .	3	Minimum	0	1000		0/12	
2		New Master File (New MF)					
. 1		New Stock Record					Identical to Stock Record.
3		Movements File					
. 1		Movement Record (MR)					
. .	1	Code			1		Code "I" = Issue Card.
. .	2	Part No.	0	4000			
. .	3	Quantity	1	9999			
4		Printed Records					
. 1		Transaction Record					
. .	1	Part No.	0	4000			Value = Part No. in MF.
. .	2	Totals					
. .	. 1	Total Issue	0	10000			
. .	. 2	Total Receipt	0	10000			
. 2		Requirement Slip					
. .	1	Part No.	0	4000			Value = Part No. in Stock Record.
. .	2	Quantity Required	1	1000			Value = Minimum Quantity in MF.
5							Working File.
. 1		Error Direction			16		

FILE OUTLINE			CARD DESCRIPTION				
LEVEL		NAME	TYPE OF CHARACTER	ZERO ACTION SIGNIFICANT	ZERO ACTION NON-SIGNIFICANT	CARD POSITION	OTHER DETAILS
3		Movements File					Card Machine = ICT 581. End of File Part No. = 4000.
. 1		Movement Record					
. .	1	Code	A10			20	
. .	2	Part No.	N	0		21→23	
			N	0	0	24	
. .	3	Quantity	N	0		30→32	
			N	0	0	33	

FILE OUTLINE			PRINTING DESCRIPTION	
LEVEL		NAME	POSITION	OTHER DETAILS
4		Printed Records		Printer = Bull BZ.
. 1		Transaction Record		Form Length = 8: Text = "Transaction" in 3/10→.
. .	1	Part No.	5/10→	Map = D D D D: Zero = * * * 0.
. .	2	Totals		
. .	. 1	Total Issue	5/20→	Map = D D , D D D: Zero = * * * * * 0.
. .	. 2	Total Receipt	5/30→	Map = D D , D D D: Zero = * * * * * 0.
. 2		Requirement Slip		Form Length = 8: Text = "Requirement" in 3/10→.
. .	1	Part No.	5/10→	Map = D D D D: Zero = * * * 0.
. .	2	Quantity Required	5/25→	Map = D D D D: Zero = * * * 0.

Procedure Description

[BEGIN]	Read Movement File then Go to X.
[MATCHING]	If Part No. in MR > Part No. in Stock Record then go to unmatched action. If Part No. in MR ≠ Part No. in Stock Record then go to Error action.
[MATCHED ACTION]	If Code = Issue Card then take MR, Quantity then subtract into quantity Stock Record then Add into Total Issue. Otherwise Take MR, Quantity then Add into Quantity in Stock Record then add into total Receipt.
[Y]	Unless End of Movement File then read Movement file then go to Matching.
[UNMATCHED ACTION]	Unless Totals is Clear: Write Transaction Record. If Quantity in Stock Record < Minimum then write Requirement Slip. Write Stock Record to New Master File.
[X]	Clear Totals. If end of Movement File or End of Master File then go to ending Otherwise Read Master File then go to Matching.
[ENDING]	If end of Master File then Close then Stop "END". Otherwise Locate Stock Record for End of Master File; Writing to New Master File then Write Stock Record to New Master File then close then stop "END".
[ERROR ACTION]	Display "CARD SEQUENCE ERROR"; "PART NUMBER IS"; Part No. in MR then Accept into Error Direction.
[Z]	If Error Direction = "CONTINUE" then go to Y. Otherwise If Error Direction = "STOP" then go to Ending. Otherwise Display "INVALID MESSAGE" then go to Z.

References

The NEBULA Programming Manual. Ferranti Publication List CS 282.
NEBULA Programming Examples and Solutions. Ferranti Publication List CS 283.
NEBULA Summarized Procedure Description. Ferranti Publication List CS 284.

Annual Prizes: Result of 1960–61 Competition

As announced in *The Computer Journal*, Volume 3, page 163, the Editorial Board have considered the papers which were published between June 1960 and April 1961 in this *Journal* and *The Computer Bulletin*.

Awards of twenty guineas each were made in respect of two papers, and these were presented to the authors at the Annual General Meeting of The British Computer Society in London on 26 September 1961.

The winning papers were:

H. H. Rosenbrock (*Constructors John Brown Limited*).
"An Automatic Method for Finding the Greatest or Least Value of a Function."
Published in *The Computer Journal*, Volume 3, page 175.

A. J. Platt (*Pilkington Brothers Limited*).
"The Experience of Applying a Commercial Computer in a British Organization."
Published in *The Computer Journal*, Volume 3, page 185.

The Editorial Board highly commended the paper by Miss Daphne E. Kilner (*British Transport Commission*) entitled "The Characteristics of Computers of the Second Decade" (*The Computer Bulletin*, Volume 4, p. 88); also the paper by F. G. Duncan and D. H. R. Huxtable (*The English Electric Co. Ltd.*) entitled "The Deuce Alphacode Translator" (*The Computer Journal*, Volume 3, page 98.

A further competition will be held on the papers published between June 1961 and April 1962.